MATH 270B Notes

Ryan Anderson

2024-01-18

Lecture 1/17: QR Decomposition

Want to solve basic matrix equations $Ax = b$. When A is square and invertible, this is trivial - $x = A^{-1}b$. Otherwise, need to solve $x : min_x \|Ax - b\|_2$.

By way of motivation note that triangular matrices and orthogonal matrices are non square but still very easy to solve. For orthogonal matrices, the inverse is the adjoint, so $Qx = b \to x = Q^* b$.

The idea then is to write $A = QR$ where Q is orthogonal and R is upper triangular. Then $Ax = b \to QRx = b \to Rx = Q^* b$.

To actually obtain this factorization we began as follows: we seek orthonormal vectors $q_1, q_2, \dots, q_n$ such that $span(q_1, q_2, \dots, q_k) = span(a_1, a_2, \dots, a_k)$. This is done with the classical Gram-Schmidt algorithm:

```
$v_j = a_j$
for$i = 1, 2, \dots, j-1$
    $r_{ij} = q_i^* a_j$
    $v_j = v_j - r_{ij} q_i$
$r_{jj} = \|v_j\|_2$
$q_j = v_j / r_{jj}$
```

Problem is that we can't use classical Gram-Schmidt due to numerical instability. This means that floating point precision problems could ruin the QR factorization. The major change in modified Gram-Schmidt is that we normalize then project away, rather than projecting away and then normalizing.

We want to show existence and uniqueness of the QR decomposition for general matrices.

Theorem: Every $A \in \mathbb{C}^{m \times n}$ has a full (also reduced) QR factorization.

Proof: By the Gram-Schmidt process, we can generate a set of orthogonal vectors $q_1, q_2, \dots, q_k$ with $span(q_1, q_2, \dots, q_k) = span(a_1, a_2, \dots, a_k)$. We can also calculate the corresponding entries of R via the G-S process.

Only problem occurs if $r_{jj} = 0$ for some j. In this case, we can just drop the jth column of Q and R and continue the process.

If $A \in \mathbb{C}^{m \times n}$ is full-rank, then we can get a unique reduced QR decomposition.

Example:

```r
A <-  matrix(c(1,1,0,1,0,1,0,1,1),nrow=3,byrow=T)
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    0
## [2,]    1    0    1
## [3,]    0    1    1
```

```r
u_1 <- A[,1]
e_1 <- u_1/norm(u_1,type="2")

u_2 <- A[,2] - (A[,2] %*% e_1)*e_1
e_2 <- u_2/norm(u_2,type="2")

u_3 <- A[,3] - (A[,3] %*% e_1)*e_1 - (A[,3] %*% e_2)*e_2
e_3 <- u_3/norm(u_3,type="2")

Q <- matrix(cbind(e_1,e_2,e_3),nrow=3,byrow=F)
R <- matrix(c(A[,1] %*% e_1,A[,2] %*% e_1, A[,3] %*% e_1,0,A[,2] %*% e_2, A[,3] %*% e_2, 0, 0, A[,3] %*

A - Q %*% R
```

```
##               [,1]          [,2]          [,3]
## [1,] 2.220446e-16  1.110223e-16 -2.966377e-16
## [2,] 2.220446e-16 -6.816043e-17  1.110223e-16
## [3,] 0.000000e+00  2.220446e-16  1.110223e-16
```

Other uses of the QR inverse include calculating the pseudo-inverse. Recall the Moore-Penrose pseudo-inverse is given as $A^{\dagger} = (A^TA)^{-1}A^T$.

With the QR decomposition we can write

$$\begin{aligned}
A^{\dagger} = (QR)^{\dagger} &= ((QR)^T(QR))^{-1}(QR)^T \\
&= (R^TQ^TQR)^{-1}R^TQ^T \\
&= (R^TR)^{-1}R^TQ^T \\
&= R^{-1}R^{-T}R^TQ^T \\
&= R^{-1}Q^T
\end{aligned}$$

If A is square and nonsingular then $A^{-1} = R^{-1}Q^T$.

We can get the absolute value of the determinant of a square matrix out via QR factorization. $det(A) = det(Q) * det(R)$ but Q is unitary so $|det(Q)| = 1$. Thus $|det(A)| = |det(R)| = |\prod_i r_{ii}|$.

We can do experiments to see the use of QR. The pseudoinverse implementation is pretty quick.

```r
A <- matrix(rnorm(3000000),nrow=3000,ncol=1000)
x <- rnorm(1000)
b <- A %*% x

start <- Sys.time()
A_pseudo_inv <- solve(t(A) %*% A) %*% t(A)
x_pseudo <- A_pseudo_inv %*% b
end <- Sys.time()
end-start
```

```
## Time difference of 2.777099 secs
```

```r
norm(x - x_pseudo,type="2")
```

```
## [1] 1.581696e-13
```

The QR implementation is less quick.

```r
start <- Sys.time()
A_qr_factor <- qr(A)
A_Q <- qr.Q(A_qr_factor)
A_R <- qr.R(A_qr_factor)
y <- t(A_Q) %*% b
x_QR <- solve(A_R) %*% y
end <- Sys.time()
end-start
```

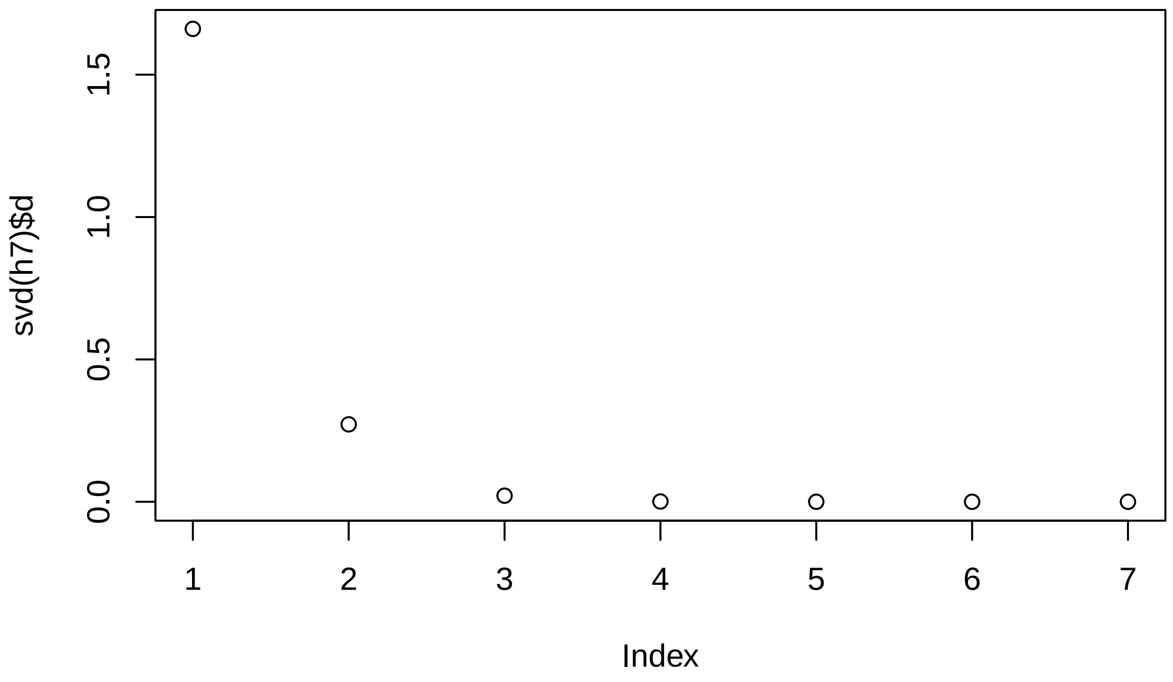
```
## Time difference of 6.515091 secs
```

```r
norm(x - x_QR,type="2")
```

```
## [1] 1.098989e-13
```

If we try to generate a Hilbert matrix we run into problems with precision. Hilbert matrices are cool because they are full-rank but have extremely large condition numbers.

```r
hilbert <- function(n) { i <- 1:n; 1 / outer(i - 1, i, "+") }
h7 <- hilbert(7)
plot(svd(h7)$d)
```



Lecture 1/22: Givens Rotations

Givens Rotations

Consider a matrix $G(i,j,\theta)$ given by

$$G(i,j,\theta) = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix}$$

As an example consider the expression

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sqrt{x}^2 + y^2 \\ 0 \end{bmatrix}$$

This indicates that we can perform Givens rotations to zero out the lower triangular portion of a matrix. This is useful for QR factorization.

In particular, we want to perform sequential Givens rotations $G_1, G_2, \ldots$ of our initial matrix to yield an upper triangular R and then take the product of their adjoints, which will yield our Q matrix.

In the below example, we want to zero out the $2, 1$ entry of the matrix, the 5.

$$\begin{matrix} 6 & 5 & 0 \\ 5 & 1 & 4 \\ 0 & 4 & 3 \end{matrix}$$

We start by calculating G_1, with $r = \sqrt{6}^2 + 5^2 = 7.81, c = 6/7.81 = 0.768, s = -5/7.81 = -0.64$.

$$G_1 = \begin{bmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This simplifies to

```
orig_mat <- matrix(c(6,5,0,5,1,4,0,4,3),nrow=3,byrow=T)
G_1 <- matrix(c(0.768,0.64,0,-0.64,0.768,0,0,0,1),nrow=3,byrow=T)
G_1
```

```
##        [,1]  [,2] [,3]
## [1,]  0.768 0.640    0
## [2,] -0.640 0.768    0
## [3,]  0.000 0.000    1
```

```
G_1 %*% orig_mat
```

```
##              [,1]   [,2]  [,3]
## [1,] 7.808000e+00  4.480 2.560
## [2,] 2.220446e-16 -2.432 3.072
## [3,] 0.000000e+00  4.000 3.000
```

We then calculate G_2.

$$G_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c & -s \\ 0 & s & c \end{bmatrix}, r = \sqrt{-2.43^2 + 4^2} = 4.68, c = -2.43/r = -0.52, s = -4/r = -0.85$$

Then we get the final matrix.

```
G_2 <- matrix(c(1,0,0,0,-0.52,0.85,0,-0.85,-0.52),nrow=3,byrow=T)
G_2
```

```
##      [,1]  [,2]  [,3]
## [1,]    1  0.00  0.00
## [2,]    0 -0.52  0.85
## [3,]    0 -0.85 -0.52
```

```
G_2 %*% (G_1 %*% orig_mat)
```

```
##               [,1]     [,2]     [,3]
## [1,]  7.808000e+00  4.48000  2.56000
## [2,] -1.154632e-16  4.66464  0.95256
## [3,] -1.887379e-16 -0.01280 -4.17120
```

Then we calculate the QR decomposition as

```
Q <- t(G_1) %*% t(G_2)
Q
```

```
##       [,1]     [,2]    [,3]
## [1,] 0.768  0.33280  0.5440
## [2,] 0.640 -0.39936 -0.6528
## [3,] 0.000  0.85000 -0.5200
```

```
R <- G_2 %*% (G_1 %*% orig_mat)
R
```

```
##               [,1]     [,2]     [,3]
## [1,]  7.808000e+00  4.48000  2.56000
## [2,] -1.154632e-16  4.66464  0.95256
## [3,] -1.887379e-16 -0.01280 -4.17120
```

```
round(Q %*% R,1)
```

```
##      [,1] [,2] [,3]
## [1,]    6    5    0
## [2,]    5    1    4
## [3,]    0    4    3
```

And we can see that $QR = A$ as desired.

Householder Reflections

Householder reflections are another way to perform QR decomposition. A *Householder reflection* is a reflection about a plane that contains the origin.

We can project a vector x onto a hyperplane H by calculating $P_x = (I - \frac{vv^*}{\|v\|^2})x$ where v is a unit vector normal to the hyperplane H.

Since that gets you onto the hyperplane, reflecting across it is given by simply doubling the distance: $F_x = (I - 2\frac{vv^*}{\|v\|^2})x$.

It turns out that reflecting over a particular hyperplane is equivalent to obtaining a vector in the direction of the unit basis vector with magnitude equal to the norm of the vector you want to reflect over. That is, $Fx = \|x\|e_1$. This lets us calculate v as $v = \|x\|e_1 - x$.

Note there are two possible such projections, one that results in a vector $Fx = \|x\|e_1$, and one that gives $F'x = -\|x\|e_1$. For stability reasons we want to choose the reflection that moves x the furthest.

This leads to our algorithm: we sequentially perform Householder reflections to zero out the lower triangular portion of our matrix, moving column-by-column. We can then take the product of the adjoints of the Householder reflections to get our Q matrix. Let $A \in R^{m \times n}$. Then we have

$$\begin{aligned}
&\text{for } k = 1, \dots, n \\
&\qquad x = A_{k:m,k} \\
&\qquad v_k = sign(x)\|x\|e_1 + x \\
&\qquad v_k = \frac{v_k}{\|v_k\|} \\
&\qquad A_{k:m,k:n} = A_{k:m,k:n} - 2v_k(v_k^* A_{k:m,k:n})
\end{aligned}$$

Our R then becomes the final A matrix after the algorithm completes. We don't explicitly compute Q while performing this, but can obtain it. Let $Q_1, Q_2, \dots$ be the reflection operators we generate at each iteration. Then $R = Q_n \dots Q_1 A$, which means $Q_n \dots Q_1 = Q^*$.

Lecture 1/24: Least Squares

Consider the least squares problem $Ax = b, A \in \mathbb{C}^{m\times n}, m \geq n$ where A is a tall matrix. Clearly there is only an exact solution if $b \in Rng(A)$, which is uncommon in practice. Instead we solve by taking

$$x_{LS} = \text{argmin}_{x\in\mathbb{C}^n} \|Ax - b\|_2$$

We define the *residual* $r = Ax_{LS} - b$ to be the error using the least squares solution. The geometry of the situation tells us that Ax_{LS} is given by the orthogonal projection of b into $Rng(A)$.

Theorem: Least Squares Solution Let $A \in \mathbb{C}^{m\times n}, m \geq n, b \in \mathbb{C}^n$. A vector $x \in \mathbb{C}^n$ minimizes $\|Ax - b\|_2^2 = \|r\|_2^2$ iff $r \perp Rng(A)$, that is $A^*r = 0$ or $A^*Ax = A^*b$ or $proj_{Rng(A)}(b) = Ax$.

The system defined by $A^*Ax = A^*b$ is called the *normal equations*, which are nonsingular iff A is full rank.

Proof For the equivalence of $A^*r = 0$ and $A^*Ax = A^*b$, start by noting $r = b - Ax$. That means $A^*r = 0$ iff $A * b = A * Ax$.

For the equivalence of $A^*Ax = A^*b$ and $proj_{Rng(A)}(b) = Ax$, we start by noting that $proj_{Rng(A)}(b) - Ax = 0$ implies $AA^\dagger b - Ax = 0$. Multiplying through by A^* we get

$$\begin{aligned} A^*AA^\dagger b - A^*Ax &= 0 \\ \Rightarrow A^*b - A^*Ax &= 0 \\ \Rightarrow A^*r &= 0 \end{aligned}$$

We can show that all solutions to the least squares problem are of the form $A^\dagger b + ker(A)$.

Implementing Least Squares via QR Factorization

For implementation's sake, let A be full-rank. Consider the normal equations $A^*Ax = A^*b$.

The Gram matrix A^*A is square, nonsingular, and positive definite. That means we can take $(A^*A)^{-1}$, which means the solution to the normal equations can be obtained simply by taking $x = (A^*A)^{-1}A^*b = A^\dagger b$. That is, the solution to the normal equations is the pseudoinverse of A when A has full column rank.

To get the pseudoinverse we will proceed by QR factorization.

$$\begin{aligned} A = QR \Rightarrow A^\dagger &= [(QR)^*(QR)]^{-1}(QR)^* \\ &= [R^*Q^*QR]^{-1}R^*Q^* \\ &= [R^*R]^{-1}R^*Q^* \\ &= R^{-1}(R^*)^{-1}R^*Q^* = R^{-1}Q^*. \end{aligned}$$

Thus we can restate the solution to the normal equations as $x = A^\dagger b \Rightarrow x = R^{-1}Q^*b \Rightarrow Rx = Q^*b$, which is now an upper triangular system.

This gives the least squares algorithm via QR factorization:

1) Compute $A = QR$.
2) Compute Q^*b.
3) Solve the upper triangular system $Rx = Q^*b$.

Note this computation is dominated by the first step, taking the factorization. With Householder reflections this takes about mn steps.

Implementing Least Squares via SVD

We can also obtain an algorithm via SVD.

$$
\begin{aligned}
A = U\Sigma V^* \Rightarrow A^\dagger &= [(U\Sigma V^*)^* U\Sigma V^*]^{-1}(U\Sigma V^*)^* \\
&= [V\Sigma^* U^* U\Sigma V^*]^{-1} V\Sigma^* U^* \\
&= [V\Sigma^2 V^*]^{-1} V\Sigma^* U^* \\
&= V\Sigma^{-1}U^*
\end{aligned}
$$

This yields $x = A^\dagger b \Rightarrow x = V\Sigma^{-1}U^* b \Rightarrow \Sigma V^* x = U^* b$.

Then our algorithm for least squares via SVD is:

1) Compute (reduced) SVD
2) Compute U^*b
3) Solve $\Sigma w = U^* b$ for w
4) Set $x = Vw$

Step 1) dominates the computational runtime here, giving overall $2mn^2 + n^3$. Note for $m >> n$, SVD and QR are similarly time expensive. When $m \simeq n$, SVD is way more expensive. However, SVD does give a diagonal system, which is trivial, compared to the upper triagonal system from QR.

Cholesky factorization is better than both!

Experimenting with Factorizations

We will now experiment with the different factorizations.

```r
M = 10000
N = 1000
A = matrix(rnorm(M*N),nrow=M,ncol=N)
x = rnorm(N)
b = A %*% x

start <- Sys.time()
A_qr_factor <- qr(A)
A_Q <- qr.Q(A_qr_factor)
A_R <- qr.R(A_qr_factor)
y <- t(A_Q) %*% b
x_QR <- solve(A_R) %*% y
end <- Sys.time()
end-start
```

```
## Time difference of 21.77861 secs
```

```r
norm(x - x_QR,type="2")
```

```
## [1] 1.457455e-13
```

```r
start <- Sys.time()
A_SVD <- svd(A)
u_b <- t(A_SVD$u) %*% b
w <- solve(diag(A_SVD$d)) %*% u_b
x_SVD <- A_SVD$v %*% w
end <- Sys.time()
end-start
```

```
## Time difference of 23.40624 secs
```

```r
norm(x - x_SVD,type="2")
```

```
## [1] 2.294841e-13
```

Lecture 1/29: Conditioning

Consider a problem $f : X \to Y$ where we have X data and Y solution. We want to describe *well-conditioned problems*, where small changes in the dataset lead to small changes in the solution.

Let δX denote a small change in the dataset, and let $\delta f = f(X + \delta X) - f(X)$. Then the *absolute condition number* of f at X is given by

$$\hat{\kappa} = \lim_{\delta \to 0} \sup_{\|\delta X\| \le \delta} \frac{\|\delta f\|}{\|X\|} = \sup_{\|\delta X\| \le \delta} \frac{\|\delta f\|}{\|X\|}$$

We also define the *relative condition number*:

$$\kappa = \lim_{\delta \to 0} \sup_{\|\delta X\| \le \delta} \frac{\|\delta f\|/\|f\|}{\|\delta X\|/\|X\|} = \sup_{\|\delta X\| \le \delta} \frac{\|\delta f\|/\|f\|}{\|\delta X\|/\|X\|}$$

Note the similarities to the definition of the derivative. Indeed we can formally describe the relationship between the absolute condition number and the Jacobian of f at X as

$$\delta f \simeq J(X)\delta X \Rightarrow \hat{\kappa} = \|J(x)\|$$

Ex. 1: Computing Eigenvalues Consider two matrices as below

$$\begin{bmatrix} 1 & 1000 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1000 \\ .001 & 1 \end{bmatrix}$$

The first matrix has eigenvalues 1 and 1, while the second has eigenvalues 0 and 2. This points to the fact that in general, computing eigenvalues is a very poorly conditioned problem.

Ex. 2: Square Root Problems Consider $f : X \to X$ where $f(x) = \sqrt{X}$. We compute the condition numbers as below by actually taking the Jacobian:

$$\kappa = \frac{\|J(X)\|}{\|f(X)\|/\|X\|} = \frac{\frac{1}{2\sqrt{X}}}{\frac{\sqrt{X}}{|X|}} = \frac{1}{2}$$

Ex. 3: Solving for Polynomial Roots Consider the problem of solving for roots of general polynomials, $p(x) = \sum_i^n c_i x^i$. We have the following theorem:

Let r be a root of $p(x)$, and let $p'(x) = \sum_i^{n-1} (i+1) c_i x^i$. Then

Proof: let $r(c_j) = r, r(c_j + \delta c_j) = \hat{r}$. Then

$$\kappa = \lim_{\delta \to 0, \delta c_j \le \delta} \sup \frac{}{\frac{|\delta c_j|}{|c_j|}}$$

A good example of extremely poorly conditioned polynomials for root solving is *Wilkinson's polynomial*, $w(x) = (x-1)(x-2)\dots(x-20)$. It turns out that if the coefficient on the x^{19} term is perturbed by $\simeq 2^{-23}$, the root will change by $\simeq 1$.

Ex. 4: Matrix-vector Multiplication We calculate κ for matrix-vector multiplication as

$$\kappa = \sup_{\delta x} \frac{\|A(x+\delta x) - Ax\|/\|Ax\|}{\|\delta x\|/\|x\|} = \sup_{\delta x} \frac{\|A\delta x\|/\|Ax\|}{\|\delta x\|/\|x\|} = \frac{\|A\|}{\|Ax\|/\|x\|} \leq \|A\|\|A^{\dagger}\|$$

The final quantity $\|A\|\|A^{\dagger}\|$ is actually the definition of the condition number of A.

Accuracy and Stability

Consider a problem of computing $f(X)$ from X, with a result in floating point $\tilde{f}(X)$. An algorithm is *accurate* if

$$\frac{\|\tilde{f}(X) - f(X)\|}{\|f(X)\|} = O(\epsilon_{machine})$$

An algorithm is *stable* if $\forall X$ we have

$$\frac{\|\tilde{f}(X) - f(\tilde{X})\|}{\|f(\tilde{X})\|} = O(\epsilon_{machine})$$

for some $\tilde{X}$ such that $\frac{\|\tilde{X}-X\|}{\|X\|} = O(\epsilon_{machine})$.

An algorithm is *backwards stable* if $\forall X$ there exists $\tilde{X}$ with $\tilde{f}(X) = f(\tilde{X})$.

Reading Notes on Stability

Stable algorithms give nearly the right answer to nearly the right question.

Backward stable algorithms give exactly the right answer to nearly the right question.

Stability of Floating Point Arithmetic

We can find the stability of subtraction as follows. We have data $(x_1, x_2) \in \mathbb{C}^2$ and algorithm $\tilde{f}(x_1, x_2) = fl(x_1) -_{FL} fl(x_2)$.

By the axiom of floating point arithmetic, $fl(x_1) = x_1(1+\epsilon_1)$, $fl(x_2) = x_2(1+\epsilon_2)$, and $fl(x_1) -_{FL} fl(x_2) = (fl(x_1) - fl(x_2))(1+\epsilon_3)$ for some $\epsilon_1, \epsilon_2, \epsilon_3 \leq \epsilon_{machine}$.

Combining, we get

$$\begin{aligned} fl(x_1) -_{FL} fl(x_2) &= (fl(x_1) - fl(x_2))(1+\epsilon_3) \\ &= (x_1(1+\epsilon_1) - x_2(1+\epsilon_2))(1+\epsilon_3) \\ &= x_1(1+\epsilon_1)(1+\epsilon_3) - x_2(1+\epsilon_2)(1+\epsilon_3) \\ &= x_1(1+\epsilon_4) - x_2(1+\epsilon_5) \end{aligned}$$

where $\epsilon_4, \epsilon_5 \leq 2\epsilon_{machine} + O(\epsilon_{machine}^2$.

This is enough to imply subtraction is backward stable.

Accuracy and Stability

Theorem - let a backward stable algorithm be applied to a problem $f : X \to Y$ with condition number κ. Then the relative error satisfies

$$\frac{\|\tilde{f}(X) - f(X)\|}{\|f(X)\|} \leq O(\kappa(x)\epsilon_{machine})$$

Stability of QR Factorization QR factorization via Householder reflections is backward stable.

Consider QR factorization via reflections to solve a matrix equation $Ax = b$. We compute $A = QR, y = Q^*b, x = R^{-1}y$.

The first step is backward stable, as we noted above.

The second step is also backward stable, and so we get $(\tilde{Q} + \delta Q)\tilde{y} = b, \|\delta Q\| = \epsilon_{machine}$.

The third step is also backward stable, as we have $(\tilde{R} + \delta R)\tilde{x} = \tilde{y}, \|\delta R\|/\|\tilde{R}\| = O(\epsilon_{machine})$.

Then we can show that the algorithm for solving the matrix equation is backward stable, with $(A + \Delta A)\tilde{x} = b, \|\Delta A\|/\|A\| = \epsilon_{machine}$.

Start by concatenating to get

$$\begin{aligned} b &= (\tilde{Q} + \delta Q)(\tilde{R} + \delta R)\tilde{x} \\ &= (\tilde{Q}\tilde{R} + \tilde{Q}\delta R + \delta Q\tilde{R} + \delta Q\delta R)\tilde{x} \\ &= (A + \delta A + \tilde{Q}\delta R + \delta Q\tilde{R} + \delta Q\delta R)\tilde{x} \end{aligned}$$

where the last equality follows from backward stability of QR factorization (hence, $\tilde{Q}\tilde{R} = A + \delta A$). Now we just need to show that $\delta A + \tilde{Q}\delta R + \delta Q\tilde{R} + \delta Q\delta R = \Delta A$ is small relative to A.

We do one trick - using $\tilde{Q}\tilde{R} = A + \delta A$ and unitarity of $\tilde{Q}$, we get

$$\frac{\|\tilde{R}\|}{\|A\|} \leq \|\tilde{Q}^*\|\frac{\|A + \delta A\|}{\|A\|} = O(1 + \epsilon_{machine})$$

Now we can examine each term:

$$\begin{aligned} \frac{\|\tilde{Q}\delta R\|}{\|A\|} &\leq \|\tilde{Q}\|\frac{\|\delta R\|}{\|\tilde{R}\|}\frac{\|\tilde{R}\|}{\|A\|} = O(\epsilon_{machine})O(1 + \epsilon_{machine}) = O(\epsilon_{machine}) \\ \frac{\|\delta Q\tilde{R}\|}{\|A\|} &\leq \|\delta Q\|\frac{\|\tilde{R}\|}{\|A\|} = \epsilon_{machine}O(1 + \epsilon_{machine}) = O(\epsilon_{machine}) \\ \frac{\|\delta Q\delta R\|}{\|A\|} &\leq \|\delta Q\|\frac{\|\delta R\|}{\|A\|} = O(\epsilon_{machine})O(\epsilon_{machine}) = O(\epsilon_{machine}^2) \end{aligned}$$

Putting this all together, we get $\|\Delta A\|/\|A\| = O(\epsilon_{machine})$.

Lecture 2/5: Stability and Conditioning of Least Squares

Given the least squares problem $Ax = b$, we want to ask the question of how x, y change as A, b change?

Recall the definition of the condition number of a problem $f : X \to Y$:

$$\kappa(x) = \sup_{\delta x} \frac{\|\delta f(x)\|}{\|f(x)\|}\frac{\|\delta x\|}{\|x\|}$$

Theorem: Let $A \in \mathbb{C}^{m\times n}$ be full rank and consider the solution x to the least squares problem $Ax = b$ and $y = Ax$. Then perturbations in A yield condition on x, y as follows:

$$\begin{bmatrix} b \text{ vs } A & y & x \\ b & \frac{1}{\cos(\theta)} & \frac{\kappa(A)}{\gamma\cos(\theta)} \\ A & \frac{\kappa(A)}{\cos(\theta)} & \kappa(A) + \frac{\kappa(A)^2\tan(\theta)}{\gamma} \end{bmatrix}$$

Proof: the first few sensitivities are easy to pull out.

$$\kappa_{b\to y} = \frac{\|J(b)\|}{\|y\|/\|b\|} = \frac{\|P\|}{\|y\|/\|b\|} = \frac{1}{\cos(\theta)}$$
$$\kappa_{b\to x} = \frac{\|J(b)\|}{\|x\|/\|b\|} = \|A^\dagger\|\frac{\|b\|}{\|y\|}\frac{\|y\|}{\|x\|} = \|A^\dagger\|\|A\|\frac{1}{\cos(\theta)} = \frac{\kappa(A)}{\gamma\cos(\theta)}$$

To calculate the sensitivities with respect to changes in A we have to understand how changes in the colspace of A affect our x, y.

Lecture 2/7: LU Decompositions

Triangular systems are easy to deal with - you can invert via back substitution.

Def: A matrix is *unit lower (upper) triangular* if it is lower (upper) triangular with 1's on the diagonal.

We are concerned with matrices that admit an *LDU decomposition* where we write $A = LDU$ where L is unit lower triangular, D is diagonal, and U is upper triangular.

An *LU decomposition* is a special case of an LDU decomposition where $D = I$.

Given $A = LDU$, we can solve $Ax = b$ by solving $LDUx = b$. We can solve $Ly = b$ by forward substitution, $Dz = y$ by scaling, and $Ux = z$ by back substitution.

We can compute the LU decomposition by Gaussian elimination.

Ex. Consider the matrix
$$\begin{bmatrix}1&1&1\\1&2&4\\3&9&27\end{bmatrix}$$

Proceed one step at a time. First let's try to zero out the (3,1) entry, so we want to take $R_3 - 3R_1 \to R_3$. We can write that in matrix form as
$$\begin{bmatrix}1&0&0\\0&1&0\\-3&0&1\end{bmatrix}\begin{bmatrix}1&1&1\\1&2&4\\3&9&27\end{bmatrix} = \begin{bmatrix}1&1&1\\1&2&4\\0&6&24\end{bmatrix}$$

Note that our 1st elimination matrix was unit lower triangular. In fact its inverse will be as well, as the inverse is given by just flipping the sign on the (3,1) entry. Hence we also have
$$\begin{bmatrix}1&1&1\\1&2&4\\3&9&27\end{bmatrix} = \begin{bmatrix}1&0&0\\0&1&0\\3&0&1\end{bmatrix}\begin{bmatrix}1&1&1\\1&2&4\\0&6&24\end{bmatrix}$$

Now say we want to zero out the (2,1) entry. We can do this by taking $R_2 - R_1 \to R_2$. We can write this as
$$\begin{bmatrix}1&1&1\\1&2&4\\0&6&24\end{bmatrix} = \begin{bmatrix}1&0&0\\1&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}1&1&1\\0&1&3\\0&6&24\end{bmatrix}$$

Now we can see where this is going. The product of all these individual elimination matrices will give us our L matrix, and the product of all the modified matrices will give us our U matrix. That is, we now have
$$\begin{aligned}\begin{bmatrix}1&1&1\\1&2&4\\3&9&27\end{bmatrix} &= \begin{bmatrix}1&0&0\\0&1&0\\3&0&1\end{bmatrix}\begin{bmatrix}1&0&0\\1&1&0\\0&0&1\end{bmatrix}\begin{bmatrix}1&1&1\\0&1&3\\0&6&24\end{bmatrix}\\ &= \begin{bmatrix}1&0&0\\1&1&0\\3&0&1\end{bmatrix}\begin{bmatrix}1&1&1\\0&1&3\\0&6&24\end{bmatrix}\end{aligned}$$

Lastly, we just need to eliminate the (3,2) entry. We can do this by taking $R_3 - 6R_2 \to R_3$. We can write this as

$$\begin{bmatrix}1 & 1 & 1\\0 & 1 & 3\\0 & 6 & 24\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 6 & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 1\\0 & 1 & 3\\0 & 0 & 6\end{bmatrix}$$

Combining all the above, we get

$$\begin{aligned}\begin{bmatrix}1 & 1 & 1\\1 & 2 & 4\\3 & 9 & 27\end{bmatrix} &= \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\3 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\1 & 1 & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 6 & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 1\\0 & 1 & 3\\0 & 0 & 6\end{bmatrix}\\ &= \begin{bmatrix}1 & 0 & 0\\1 & 1 & 0\\3 & 6 & 1\end{bmatrix}\begin{bmatrix}1 & 1 & 1\\0 & 1 & 3\\0 & 0 & 6\end{bmatrix}\\ A &= LU\end{aligned}$$

Algorithmically, we can write this as follows. In Gaussian elimination for LU, let A be $m \times m$. Then we have

```
$U = A, L = I$
for $k = 1$ to $m - 1$
  for $j = k + 1$ to $m$
    $L_{j,k} = U_{j,k}/U_{k,k}$
    $U_{j,k:m} = U_{j,k:m} - L_{j,k}U_{k,k:m}$
```

Stability of LU Decomposition

Consider the matrix

$$A = \begin{bmatrix}0 & 1\\1 & 1\end{bmatrix} \simeq \begin{bmatrix}10^{-20} & 1\\1 & 1\end{bmatrix}$$

The LU decomposition of A is

$$L = \begin{bmatrix}1 & 0\\10^{20} & 1\end{bmatrix}, U = \begin{bmatrix}10^{-20} & 1\\0 & 1 - 10^{20}\end{bmatrix}$$

Consider perturbations of the decomposition matrices so we get

$$\tilde{L} = \begin{bmatrix}1 & 0\\10^{20} & 1\end{bmatrix}, \tilde{U} = \begin{bmatrix}10^{-20} & 1\\0 & -10^{20}\end{bmatrix} \Rightarrow \tilde{L}\tilde{U} = \begin{bmatrix}10^{-20} & 1\\1 & 0\end{bmatrix}$$

Hence even if we can produce L, U stably, may not mean $Ax = b$ is solved stably. This points to the fact that the LU factorization for solving $Ax = b$ is not backward stable.

We can get around this by modifying the algorithm to better choose the way we eliminated entries. The choice of such eliminations is called choosing a *pivot*.

Lecture 2/14: Cholesky Factorization

Recall that a real symmetric matrix M is *positive definite* if for all non-zero vectors x, we have $x^T Mx > 0$. A real symmetric matrix is *positive semi-definite* if for all non-zero vectors x, we have $x^T Mx \geq 0$.

Note that we can write $x^T Ax$ as

$$x^T Ax = \sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}x_i x_j = \sum_{i=1}^{n} A_{ii}x_i^2 + 2\sum_{i>j} A_{ij}x_i x_j$$

Equivalently, a real symmetric matrix M is positive definite if all its eigenvalues are positive, and positive semi-definite if all its eigenvalues are non-negative.

Recall the *Gram matrix* of a matrix B is B^TB. Every Gram matrix is PSD, and a Gram matrix $A = B^TB$ is positive definite if $x^TAx = \|Bx\|^2 > 0$ which is true when B is non-singular.

Cholesky Factorization

Every positive definite matrix A has a *Cholesky factorization* of the form $A = R^TR$ where R is upper triangular with positive diagonal elements. R is the *Cholesky factor* of A.

The complexity of computing R is $O(n^3/3)$. R is sort of like the "square root" of A.

We can write the Cholesky factorization as follows. Let A be a positive definite matrix. Then we have

$$\begin{aligned}
A &= R^TR \\
\begin{bmatrix} A_{11} & A_{1,2:n} \\ A_{2:n,1} & A_{2:n,2:n} \end{bmatrix} &= \begin{bmatrix} R_{11} & 0 \\ R_{1,2:n}^T & R_{2:n,2:n}^T \end{bmatrix} \begin{bmatrix} R_{11} & R_{1,2:n} \\ 0 & R_{2:n,2:n} \end{bmatrix} \\
&= \begin{bmatrix} R_{11}^2 & R_{11}R_{1,2:n} \\ R_{1,2:n}^T R_{11} & R_{1,2:n}^T R_{1,2:n} + R_{2:n,2:n}^T R_{2:n,2:n} \end{bmatrix}
\end{aligned}$$

Hence, we get $R_{11} = \sqrt{A}_{11}$ and $R_{1,2:n} = \frac{A_{1,2:n}}{R_{11}}$ and $A_{2:n,2:n} - R_{1,2:n}^T R_{1,2:n} = R_{2:n,2:n}^T R_{2:n,2:n}$, and we can recursively apply this to the bottom right submatrix.

Ex: Let $A = \begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix}$. Then we have

$$\begin{aligned}
\begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix} &= \begin{bmatrix} R_{11} & 0 & 0 \\ R_{21} & R_{22} & 0 \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} R_{11} & R_{21} & R_{31} \\ 0 & R_{22} & R_{32} \\ 0 & 0 & R_{33} \end{bmatrix} \\
&= \begin{bmatrix} 5 & 0 & 0 \\ 3 & R_{22} & 0 \\ -1 & R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} 5 & 3 & -1 \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix}
\end{aligned}$$

After completing the first row/column, we start with the bottom right submatrix.

$$\begin{aligned}
A_{2:n,2:n} - R_{1,2:n}^T R_{1,2:n} &= R_{2:n,2:n}^T R_{2:n,2:n} \\
\begin{bmatrix} 18 & 0 \\ 0 & 11 \end{bmatrix} - \begin{bmatrix} 3 & -1 \end{bmatrix} \begin{bmatrix} 3 \\ -1 \end{bmatrix} &= \begin{bmatrix} R_{22} & 0 \\ R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} R_{22} & R_{23} \\ 0 & R_{33} \end{bmatrix} \\
\begin{bmatrix} 9 & 3 \\ 3 & 10 \end{bmatrix} &= \begin{bmatrix} R_{22}^2 & R_{22}R_{23} \\ R_{22}R_{23} & R_{23}^2 + R_{33}^2 \end{bmatrix} \\
&\Rightarrow R_{22} = 3, R_{23} = 1, R_{33} = \sqrt{10-1} = 3
\end{aligned}$$

Thus we have $R = \begin{bmatrix} 5 & 3 & -1 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{bmatrix}$.

Lecture 2/21: Eigenvalue Problems

To start note that the matrix $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ is not diagonalizable. That means, there does not exist B, D such that $A = BDB^{-1}$.

A series of definitions is useful. For a matrix A, a pair (x, λ) is an *eigenpair* of A if $Ax = \lambda x$ for some scalar λ. The scalar λ is called the *eigenvalue* of A corresponding to the eigenvector x. The *eigenspace* is the space on which A acts like scalar multiplication: the set of eigenvectors corresponding to an eigenvalue λ is denoted E_λ. Note $AE_\lambda \subset E_\lambda$. The *spectrum* of A is the set of all eigenvalues of A.

Thm: The determinant of a matrix A is the product of its eigenvalues, and the trace of A is the sum of its eigenvalues.

A factorization of a square matrix A into a product $A = X\Lambda X^{-1}$ where Λ is diagonal and X is invertible is called an *eigenvalue factorization* of A. The columns of X are the eigenvectors of A, and the diagonal elements of Λ are the eigenvalues of A.

Every matrix has an SVD, but not every matrix has an eigenvalue factorization. For example, the matrix $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ does not have an eigenvalue factorization. However, it does have an SVD.

Characteristic Polynomials and Eigenvalues

The characteristic polynomial of a matrix $A \in \mathbb{C}^{m\times m}$ is defined as $p_A(z) = \det(zI - A)$. The characteristic polynomial is a monic polynomial of degree n in λ.

Thm: λ is an eigenvalue of A iff it is a root of the characteristic polynomial. If $A \in \mathbb{C}^{m\times m}$ and you count with algebraic multiplicity, then there are m eigenvalues not necessarily distinct.

We distinguish two notions of multiplicity. The *algebraic multiplicity* of an eigenvalue λ is the number of times λ appears as a root of the characteristic polynomial. The *geometric multiplicity* of an eigenvalue λ is the dimension of the eigenspace E_λ.

Ex: let $A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}, B = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix}$. $p_A(z) = (z-2)^3, p_B(z) = (z-2)^3$. The algebraic multiplicity of 2 is 3 for both A and B.

However, note that the eigenvectors of A are just the coordinate vectors e_1, e_2, e_3. This means that E_λ has dimension 3, and hence λ has geometric multiplicity 3.

For B the only eigenvector is e_1, so the geometric multiplicity of λ is 1.

Thm: The geometric multiplicity of an eigenvalue is less than or equal to its algebraic multiplicity. Moreover, if X is nonsingular then A and XAX^{-1} have the same eigenvalues and the same multiplicities.

Diagonalization and Unitary Diagonalization

An eigenvalue whose algebraic and geometric multiplicities are equal is called *simple*. An eigenvalue whose algebraic multiplicity is greater than its geometric multiplicity is called *defective*. A matrix with any defective eigenvalues is called *defective* (otherwise *non-defective*).

Theorem: A matrix $A \in \mathbb{C}^{m\times m}$ is non-defective iff it has an eigendecomposition $A = X\Lambda X^{-1}$ where X is nonsingular and Λ is diagonal.

Furthermore, we might get lucky and have X be unitary, in which case the eigendecomposition is given as $A = X\Lambda X^*$. Then we say that A is *unitarily diagonalizable*.

A *Schur factorization* of A is given by $A = QTQ^*$ where Q is unitary and T is upper triangular. Since A, T are similar, and since upper triangular matrices contain their eigenvalues on the diagonal, the diagonal elements of T are the eigenvalues of A.

Theorem: Every square matrix has a Schur factorization.

Now note that if $A = QTQ^*$ is normal, i.e. $AA^* = A^*A$, then T is also normal. Since T is normal and upper triangular, it is diagonal. Thus, if A is normal, then the Schur form gives us unitary diagonalization!

Theorem: A matrix A is normal iff it is unitarily diagonalizable.

To summarize:

1. Every matrix A has an SVD $A = U\Sigma V^*$.
2. Every square matrix A has a unitary triangularization (Schur factorization) $A = QTQ^*$.
3. Every non-defective matrix A has a diagonalization (resp. eigendecomposition) $A = X\Lambda X^{-1}$.
4. Every normal matrix A has a unitary diagonalization $A = Q\Lambda Q^*$.

These diagonalizations are useful for finding eigenvalues. Recall that root-finding is ill-conditioned, so we want to avoid calculating eigenvalues by solving the characteristic polynomial.

Iterative Methods for Eigenvalue Computation

When computing Schur form iteratively we end up with intermediate matrices whose product is our final matrix. This resembles compiling Householder reflections into a final QR factorization.

Because we need to be careful about pivoting, we want to have our intermediate steps end in *Hessenberg form*, with zeroes below the first subdiagonal.

Consider $A = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mm} \end{bmatrix}$. Then apply our Householder reflections to get

$$\begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mm} \end{bmatrix} \to_{Q_1^*} \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a'_{21} & a'_{22} & \dots & a'_{2m} \\ 0 & a_{32} & \dots & a_{3m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{mm} \end{bmatrix} \to_{Q_1} \begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1m} \\ a'_{21} & a'_{22} & \dots & a'_{2m} \\ 0 & a_{32} & \dots & a_{3m} \\ 0 & 0 & a_{43} & \dots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{mm} \end{bmatrix}$$

Then we iterate on the sub-blocks. That is, we first compute the Householder $\hat{P}_1$ which sends $(a_{2:m,1})$ to e_1 and let $P_1 = \begin{bmatrix} 1 & 0 \\ 0 & \hat{P}_1 \end{bmatrix}$. Then we compute $P_1AP_1^*$ and get

$$\begin{bmatrix} a'_{11} & a'_{12} & \dots & a'_{1m} \\ a'_{21} & a'_{22} & \dots & a'_{2m} \\ 0 & a_{32} & \dots & a_{3m} \\ 0 & a_{42} & a_{43} & \dots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{m3} & \dots & a_{mm} \end{bmatrix}$$

Then compute $\hat{P}_2$ to send $(a_{3:m,2})$ to e_2 and let $P_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \hat{P}_2 \end{bmatrix}$.

Once complete, we get $P_n \dots P_1AP_1^{-1} \dots P_n^{-1} = H$, where H is Hessenberg. Then let $Q = P_n \dots P_1$ and $A = QHQ^*$.

Lecture 2/26: Hessenberg Reductions

The algorithm for Hessenberg reduction via Householder reflections is as follows:

```
A <- matrix(rnorm(n*n), n,n)
H <- A
```

```r
V <- vector(mode = "list", length = n-2)

# Householder transformation
for (k in 1:(n-2)) {
    v <- H[(k+1):n, k]
    sgn <- sign(v[1])
    if (sgn == 0) sgn <- 1
    v[1] <- v[1] + sgn * norm(v,type="2")
    if (norm(v,type="2") != 0) v <- v / norm(v,type="2")

    H[(k+1):n,k:n] <- H[(k+1):n, k:n] - 2 * v %*% (t(v) %*% H[(k+1):n,k:n])
    H[ ,(k+1):n] <- H[ , (k+1):n] - (2 * (H[ , (k+1):n] %*% v)) %*% t(v)
    V[[k]] <- v
}
Q <- diag(nrow=n)
for (j in (n-2):1) {
    Q[(j+1):n, ] <- Q[(j+1):n, ] - (2 * V[[j]]) %*% (t(V[[j]]) %*% Q[(j+1):n, ])
}

list(H = H, P = Q)
```

This Householder method reduces any matrix to Hessenberg form which is nearly triangular in $\frac{10}{3}n^3$ flops. If A is Hermitian, then the Hessenberg form is tridiagonal, and this allows us to do computations in only $\frac{4}{3}n^3$ flops.

What's critical here is that the Hessenberg form of a matrix has the same eigenvalues as the original matrix.

Other Eigenvalue methods

Other method for eigenvalue computation include the Jacobi algorithm, the divide-and-conquer algorithm, and the bisection algorithm.

Jacobi Algorithm For symmetric matrices, we want to repeatedly diagonalize small submatrices of A until A is diagonal, which obviously gives you the eigenvalues! The trick here is to use Givens rotations to diagonalize 2×2 submatrices. That is, for a submatrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, we want to find a Givens rotation G such that $G^T \begin{bmatrix} a & b \\ c & d \end{bmatrix} G = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$.

We start by trying $G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$, where $c = \cos\theta$ and $s = \sin\theta$, $\tan\theta = \frac{2d}{b-a}$. We repeatedly apply this to the 2×2 submatrices, usually beginning with the submatrix with largest off-diagonal entries.

Lecture 2/28: QR Algorithm

Let A be real and symmetric. Hence it has real eigenvalues and orthonormal eigenvectors, and is diagonalizable by its Schur form.

The (reduced) QR factorization of A is $A = QR$, where Q is orthogonal and R is upper triangular.

Power Iteration as a Motivation

One jumping off point is the power iteration method for calculating eigenvalues. We start with a random unit vector $v^{(0)}$ and repeatedly apply A and renormalize. In the limit, $v^{(k)} \to v_1$, the eigenvector corresponding to the largest eigenvalue.

Consider an extension of power iteration where we begin with a matrix $V^{(0)} = (v_1^{(0)}, \dots, v_n^{(0)})$ and repeatedly apply A and renormalize. In the limit, $V^{(k)} \to V$, where the columns of V are the eigenvectors of A. We assume that there is a gap between the first and second eigenvalues, and the rate of convergence is determined by the max of the eigengaps.

QR Algorithm

The basic idea going forward is that we want to use the similarity transform enabled by the QR factorization. We start with $A = QR$, then triangularize via $R = Q^*A$, then right multiply by Q again to get $RQ = Q^*AQ$. We then iterate on this process.

In particular, we get the QR algorithm via the following steps:

1. Start with $A_0 = A$.
2. Compute the QR factorization $A_k = Q_k R_k$.
3. Set $A_{k+1} = R_k Q_k$.
4. Repeat until A_k is upper triangular.

Since A_k is always similar to A, it has the same eigenvalues.

```
A <- matrix(c(1,2,3,4),nrow=2,byrow=T)
for(i in 1:100) {
    Q <- qr.Q(qr(A))
    R <- qr.R(qr(A))
    A <- R %*% Q
}
A
```

```
##                [,1]       [,2]
## [1,]  5.372281e+00 -1.0000000
## [2,] 1.071841e-115 -0.3722813
```

```
Q
```

```
##                 [,1]          [,2]
## [1,]  -1.000000e+00 -2.879117e-115
## [2,] -2.879117e-115   1.000000e+00
```

Practical Modifications to the QR Algorithm

The QR algorithm is not practical for large matrices, but there are some modifications that make it more practical.

We first reduce A to tridiagonal form via Householder reflections. Then we apply the QR algorithm to the tridiagonal matrix. We also pick a shift μ_k to accelerate convergence.

With these modifications in mind, the QR algorithm becomes:

1. Start with the tridiagonalization of A, $Q_0 A Q_0^T = T_0$.
2. Pick a shift μ_k (one easy choice is the final diagonal element of our current iterate $\mu_k = A_{mm}^{(k-1)}$.
3. Compute the QR factorization of $A^{(k-1)} - \mu_k I$: $A^{(k)} - \mu_k I = Q_k R_k$.
4. Recombine the factors to get $A^{(k)} = R_k Q_k + \mu_k I$.
5. If any off-diagonal elements are small, we can set them to zero and continue.
6. Reapply the QR algorithm to the resulting submatrices.

Analysis of QR Algorithm

QR works because it is secretly just a simultaneous iteration of the power iteration method.

Thm: Let the pure QR algorithm be applied to a real symmetric matrix A with distinct eigenvalues. Then the iterates $A^{(k)}$ converge to a diagonal matrix with the eigenvalues of A on the diagonal at a rate given by $\max(\lambda_{k+1}/\lambda_k)$, and the iterates Q_k converge to the matrix of eigenvectors of A.