Integer Programming
ISE 418

Lecture 24

Dr. Ted Ralphs

Reading for This Lecture

- "Computer Solution of Linear Algebraic Systems", Forsythe and Moler
- "Computer Methods for Mathematical Computations", Forsythe, Malcolm, and Moler

Numerical Analysis

Numerical Analysis

- *Numerical analysis* is the study of algorithms for problems from continuous mathematics.
- A *problem* is a map from $f : X \to Y$, where X and Y are normal vector spaces.
- A *numerical algorithm* is a procedure which calculates $F(x) \in Y$, an approximation of $f(x)$.
- As we have already discussed, we can define these algorithms in terms of an algorithmic map.
- Because we have to use floating point arithmetic and other approximations, our answers will not be exact.

Conditioning

- A problem is *well-conditioned* if $x' \approx x \Rightarrow f(x') \approx f(x)$.
- Otherwise, it is *ill-conditioned*.
- Notice that well-conditioned requires all small perturbations to have a small effect.
- Ill-conditioned only requires some small perturbations to have a large effect.
- Condition number of a problem
  - Absolute
  - Relative

Stability

- An algorithm is *stable* if $F(x) \approx f(x')$ for some $x' \approx x$.
- This says that a stable algorithm computes "nearly the right answer" to "nearly the right question."
- Notice the contrast between conditioning and stability:
  - Conditioning applies to problems.
  - Stability applies to algorithms.

Accuracy

- Stability plus good conditioning implies *accuracy*.
- If a stable algorithm is applied to a well-conditioned problem, then $F(x) \approx f(x)$.
- Conversely, if a problem is ill-conditioned, an accurate solution may not be possible or even meaningful.
- We cannot ask more of an algorithm than stability.

Examples

- Addition, subtraction, multiplication, division.
  - Addition, multiplication, division with positive numbers are well-conditioned problems.
  - Subtraction is not.
- Zeros of a quadratic equation.
  - The problem of computing the two roots is well-conditioned.
  - However, the quadratic formula is not a stable algorithm.
- Solving systems of linear equations $Ax = b$.
  - Conditioning depends on the matrix A.

Floating-point Arithmetic

- The floating-point numbers F are a subset of the real numbers.
- For a real number x, let $fl(x) \in F$ denote the floating point approximation to x.
- Let $\odot$ and $\cdot$ represent the four floating point and exact arithmetic operations.
- Typically, there is a number $u << l$ called *machine epsilon* such that
  - $fl(x) = x(1+\varepsilon)$ for some ε with $|\varepsilon| \leq u$.
  - $\forall a, b \in F, a \odot b = (a \cdot b)(1+\varepsilon)$ for some $|\varepsilon|$ with $\varepsilon \leq u$.

Stability of Floating Point Arithmetic

- Floating point arithmetic is stable for computing sums, products, quotients, and differences of two numbers.
- Sequences of these operation can be unstable however.
- Example
  - Assume 10 digit precision
  - $(10^{-10} + 1) - 1 = 0$
  - $10^{-10} + (1 - 1) = 10^{-10}$
- Floating point operations are not always associative.

Floating Point Weirdness

- The fact that numbers are stored using binary representations has vast implications and can have surprising results.
- In Python, we have

```
>>> round (2.675, 2)
2.67
>>> 0.1 + 0.2
0.30000000000000004
```

- This because the floating point representations of these seemingly innocuous numbers are not exact.

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.674999999999999822364316059974953532218933105468 75')
```

More Bad Examples

- Calculating e^{-a} with $a > 0$ by Taylor Series.
  - The round-off error is approximately u times the largest partial sum.
  - Calculating e^{a} and then taking its inverse gives a full-precision answer.
- Roots of a quadratic $(ax^2 + bx + c)$
  - If $x_1 \approx 0$ and $x_2 >> 0$, then the quadratic formula is unstable.
  - Computing x_2 by the quadratic formula and then setting $x_1 = cx_2/a$ is stable.

Backward Error Analysis

- *Backward error analysis* is a method of analyzing round-off error and assessing stability.
- We want to show that the result of a floating-point operation has the same effect as if the original data had been perturbed by an amount in $O(u)$.
- If we can show this, then the algorithm is stable.

More Examples

- Matrix factorization.
  - Generally ill-conditioned.
  - There are stable algorithms, however.
- Zeros of a polynomial.
  - Generally ill-conditioned.
- Eigenvalues of a matrix.
  - For a symmetric matrix, finding eigenvalues is well-conditioned, finding eigenvectors is ill-conditioned.
  - For non-symmetric matrices, both are ill-conditioned.
  - In all cases, there are stable algorithms.

Solving Systems of Equations

Solving Systems of Equations

- Problem: Given a matrix $A \in \mathbb{R}^{n \times n}$ and a vector $b \in \mathbb{R}^n$, we wish to find $x \in \mathbb{R}^n$ such that $Ax = b$.
- Question: Is this problem well-conditioned?
- Answer: It depends...

Review: Real Vector Spaces

A *real vector space* is a set V along with

- an addition operator that is commutative and associative;
- an element $0 \in V$ such that $a + 0 = a \; \forall a \in V$;
- an additive inverse operation such that $\forall a \in A, \exists a'$ such that $a + a' = 0$; and
- a scalar multiplication operator such that $\forall \lambda, \mu \in \mathbb{R}, a, b \in V$:
  - $\lambda(a + b) = \lambda a + \lambda b$
  - $(\lambda + \mu)a = \lambda a + \mu a$
  - $\lambda(\mu a) = (\lambda\mu)a$
  - $1a = a$

Review: Norms on Vector Spaces

- A *norm* on a vector space is a function $\|\cdot\| : V \to \mathbb{R}$ satisfying
  - $\|v\| \geq 0 \; \forall v \in V$
  - $\|v\| = 0 \Leftrightarrow v = 0$
  - $\|v + w\| \leq \|v\| + \|w\| \; \forall v, w \in V$
  - $\|\lambda v\| = |\lambda| \|v\| \forall \lambda \in \mathbb{R}, v \in V$
- Norms are a measure of the "distance" between two objects in a vector space.
- These are the properties you would expect such a measure to have.

Review: Examples of Vector Spaces

- $\mathbb{R}^n$
- $\mathbb{Z}^n$
- $\mathbb{R}^{n \times n}$
- $\{y \in \mathbb{R}^n \mid Ax = y, \exists x \in \mathbb{R}^n\}$

Review: Norms on Matrices and Vectors

- Unless otherwise indicated, we will use the Euclidean norm ($\ell^2 - norm$) for vectors, defined as

$$\|x\| = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

- The matrix norm corresponding to the $\ell^2 - norm$ on vectors is the *spectral norm*:

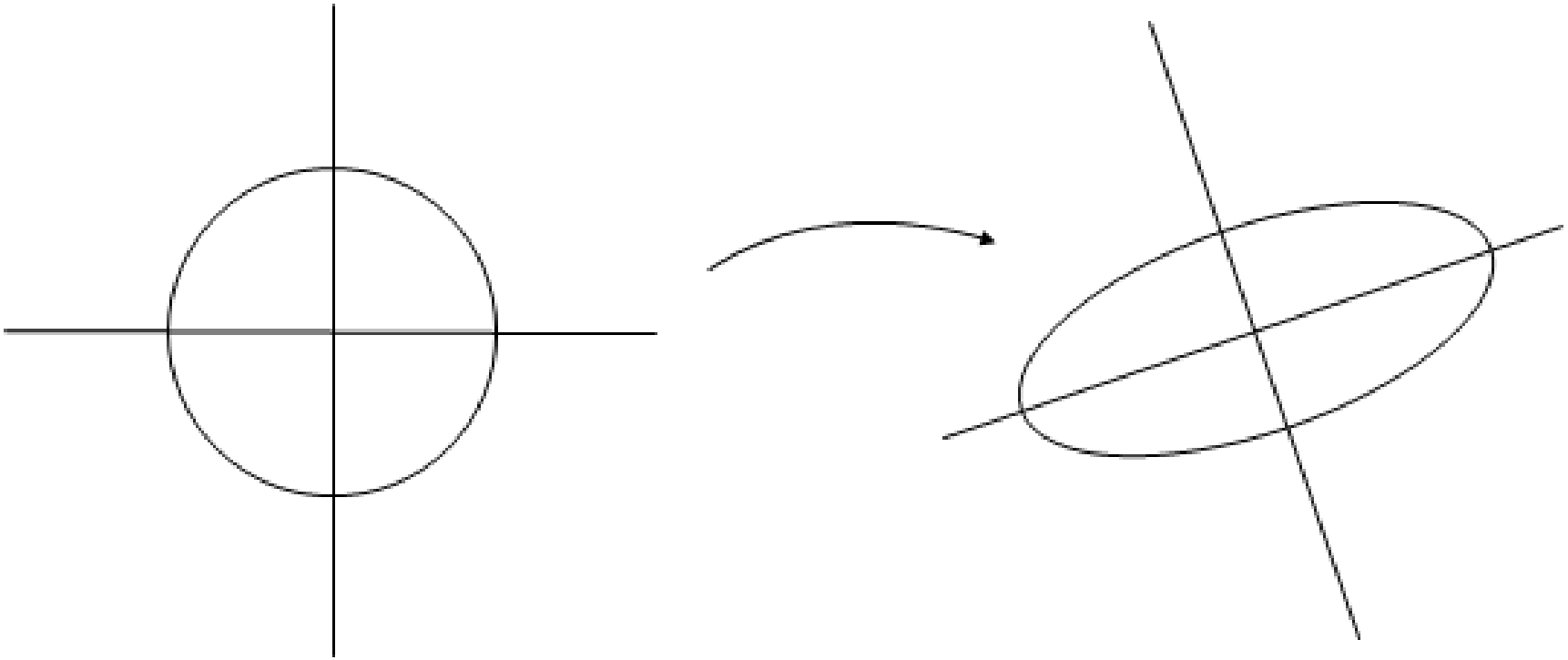
$$\|A\| = \max\{\|Ax\|/\|x\|, x \neq 0\} \; \forall A \in \mathbb{R}^{n \times n}$$

- Note the following definitions and properties.
  - $|x^\top y| \leq \|x\|\|y\| \; \forall x, y \in \mathbb{R}^n$
  - $\|Ax\| \leq \|A\|\|x\| \; \forall A \in \mathbb{R}^{n \times n}, x \in \mathbb{R}^n$
  - $\|AB\| \leq \|A\|\|B\| \; \forall A, B \in \mathbb{R}^{n \times n}$

Singular Values

- The conditioning of the problem of solving a system of equations depends roughly on properties of the matrix A.
- Recall that an orthogonal matrix U has the property $U^TU = UU^T = I$.
- Given $A \in \mathbb{R}^{m\times n}$, there exist orthogonal matrices U, V such that $U^TAV = D$, where
  - D is a diagonal matrix for which diagonal elements are $\mu_1 \geq \mu_2 \geq \ldots \geq \mu_r > \mu_{r+1} = \ldots = \mu_n = 0$,
  - r is the rank of A, and
  - μ_i is the non-negative square root of the i^{th} eigenvalue of A^TA.
- This is called the *singular value decomposition*.

Importance of the SVD



Implications

- Multiplying by A represents a rotation and a scaling of axes to get from one space to the other.
- μ_i is the non-negative square root of the i^{th} eigenvalue.
- Notice that $\|A\| = \|D\| = \mu_1$.
- So the norm of A is the maximum amount any axis gets magnified by A.
- If $r = n$, then we can easily derive the inverse of A.
- Also, $\|A^{-1}\| = \|A\|^{-1} = 1/\mu_n$.

Condition of a Linear System

- Consider the problem of solving $Ax = b$.
- If we perturb b, how much does the x change?

$$\begin{aligned} x + \delta x = A^{-1}(b + \delta b) \quad &\Rightarrow \quad \delta x = A^{-1}\delta b \\ &\Rightarrow \quad \|\delta x\| \leq \|A^{-1}\|\|\delta b\| \\ &\Rightarrow \quad \|\delta x\|\|b\| \leq \|A\|\|A^{-1}\|\|x\|\|b\| \\ &\Rightarrow \quad \|\delta x\|/\|x\| \leq \|A\|\|A^{-1}\|(\|\delta b\|/\|b\|) \end{aligned}$$

- The condition number of a matrix is the quantity $\mathrm{cond}(A) = \|A\|\|A^{-1}\|$

Condition Number

- Note that $\mathrm{cond}(A) = \mu_1/\mu_n$.
- Hence it is a relative measure of how much distortion A causes to its input.
- It is also a measure of how much the inaccuracies in b get multiplied in x when solving systems $Ax = b$.
- If b is the result of a previous calculation, then $\|\delta b\|/\|b\|$ is at best equal to u (machine epsilon).
- The inaccuracies in x will then be at best $u\,\mathrm{cond}(A)$.

Interpretation

- Orthogonal matrices have a norm of 1 and hence don't cause any scaling or distortion.
- Singular matrices have at least one singular value equal to 0 and hence have a norm of "infinity".
- "Nearly singular" matrices are the ones that cause problems.
- These are ones that have singular values "relatively close" to zero.

Gaussian Elimination

- Standard row operations
  - Interchange rows
  - Multiply rows by a scalar
  - Subtract a multiple of row j from row i
- Standard algorithm
  - Elimination Phase
  - Back-substitution Phase

Gaussian Elimination

```
#Elimination Phase
for k in range(n-1):
    for row in range(k+1, n):
        multiplier = A[row,k]/A[k,k]
        A[row, k:] = A[row, k:] - multiplier*A[k, k:]
        b[row] = b[row] - multiplier*b[k]
#Back Substitution Phase
x = np.zeros(n)
for k in range(n-1, -1, -1):
    x[k] = (b[k] - np.dot(A[k,k+1:],x[k+1:]))/A[k,k]
```

- Note that this implementation uses the Python package `numpy`.
- Is this algorithm correct? What can go wrong?

Example:Numerical Issues

Consider solving the following system:

$$.0001x_1 + x_2 = 1$$
$$x_1 + x_2 = 2$$

The true solution is

$$x_1 = \frac{10000}{9999} \qquad x_2 = \frac{9998}{9999}$$

Executing the algorithm from the previous slide with 3-digit precision, we obtain

$$x_1 = 0.00 \qquad x_2 = 1.00$$

Pivoting

- If at some point in the previous algorithm, we have `A[k,k] = 0`, then the algorithm fails.
- The example shows that even when this is not the case, a very small pivot element (`A[k, k]` in the algorithm) can cause bad numerical errors,
- Essentially, the problem arises from the difference in magnitude of the coefficients on x_1 in the two equations.
- We can take care of this problem by *pivoting*, which is to do the elimination in a different order.
- We swap the current row `k` for row `j > n` in which element `k` is non-zero (and usually of large magnitude).
- In our example, using this strategy results in a full-precision answer.
- Note that if element `k` is zero in all rows greater than or equal to `k`, there is no pivot available and the matrix is singular.

Pivoting Strategies

- Partial Pivoting Strategy
  - Take the pivot element to be the largest element (in absolute value) in the column
- Complete Pivoting Strategy
  - Take the pivot element to be the largest element (in absolute value) in the whole matrix
- Using these strategies, we can limit round-off error
- Roughly, we will obtain x such that $(A + \delta A)x = b$ and the entries of δA are in $O(nu)$.

Gaussian Elimination with Partial Pivoting

```
#Elimination Phase
for k in range(n-1):
    #Choose largest pivot element below (and including) k
    maxindex = abs(A[k:,k]).argmax() + k
    if A[maxindex, k] == 0:
        raise ValueError("Matrix is singular.")
    #Swap rows
    if maxindex != k:
        A[[k,maxindex]] = A[[maxindex, k]]
        b[[k,maxindex]] = b[[maxindex, k]]
    for row in range(k+1, n):
        multiplier = A[row,k]/A[k,k]
        A[row, k:] = A[row, k:] - multiplier*A[k, k:]
        b[row] = b[row] - multiplier*b[k]
#Back Substitution Phase
x = np.zeros(n)
for k in range(n-1, -1, -1):
    x[k] = (b[k] - np.dot(A[k,k+1:],x[k+1:]))/A[k,k]
```

Ordinarily, we also keep track of the pivots so that we can report the answer in the original order.

The LU Factorization

- The LU decomposition
  - Let us assume $\det(A_k) \neq 0 \forall k$, where A_k is the sub-matrix consisting of the first k rows and columns of A.
  - $\exists$ a lower triangular matrix L with 1's on the diagonal, and an upper triangular matrix U such that $A = LU$.
- With an LU factorization, can solve the system $Ax = b$
  - Solve $Ly = b$ (elimination phase)
  - Solve $Ux = y$ (back substitution phase)
- In our Gaussian elimination algorithm, we actually transform the matrix A into the matrix U in the elimination phase.
- All that remains is to construct the matrix L.
- This can be done easily.

Interchanges in LU Factorization

- Just as in Gaussian elimination, we also need to worry about pivoting for numerical stability here.
- Row interchanges are captured by constructing a permutation matrix P.
- Taking into account row interchange, we can state more generally that for any matrix A, there exists a permutation matrix P, a lower triangular matrix L with unit diagonal, and an upper triangular matrix U such that

$$LU = PA$$

Simple LU Factorization: Pre-computing the Permutation Matrix

```
def pivotize(m):
    n1,n2 = m.shape
    if n1!=n2:
        raise Exception("Matrix is not square!")
    n = n1
    ID = numpy.eye(n)
    for j in xrange(n):
        row = max(xrange(j, n), key=lambda i: m[i,j])
        if j != row:
            ID[j], ID[row] = ID[row], ID[j]
    return ID
```

Simple LU Factorization

```
def lu(A):
    n = len(A)
    L = numpy.zeros(shape=(n,n), dtype=float)
    U = numpy.zeros(shape=(n,n), dtype=float)
    P = pivotize(A)
    A2 = P.dot(A)
    for j in xrange(n):
        L[j,j] = 1.0
        for i in xrange(j+1):
            s1 = sum(U[k,j] * L[i,k] for k in xrange(i))
            U[i,j] = A2[i,j] - s1
        for i in xrange(j, n):
            s2 = sum(U[k,j] * L[i,k] for k in xrange(j))
            L[i,j] = (A2[i,j] - s2) / U[j,j]
    return (L, U, P)
```

Calculating an LU Factorization "In Place"

- The LU factorization can be computed "in-place".
- Row interchanges can be represented by permutation matrices.
- Elimination operations can be represented by *eta matrices*.
- The eta matrices can be stored compactly as elimination proceeds.

Eta Matrices

- Performing an elimination step is the same as left-multiplying by a specially constructed matrix.
- To multiply row j by β and add it to row i, take I and change the entry (i, j) to β.
- A sequence of row operations can similarly be represented as a matrix.
- Hence, step `k` of the elimination phase is the same as multiplying by

$$M_k = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -m_{k+1,k} & \ddots & \\ & & \vdots & & \\ & & -m_{n,k} & & 1 \end{bmatrix}$$

where $m_{i,j} = A_{i,j}^{(k-1)} / A_{k,k}^{(k-1)}$ and $A^{(k)} = M_k M_{k-1} \cdots M_1 A$.

Storing the Eta Matrices

- Note that after elimination step `k`, the entries of $A^{(k)}$ are all zero below the diagonal.
- We can store the entries of M_k in place of these zeros.
- At the end of the elimination phase, we have that

$$U = A^{(k)} = M_{n-1} \cdots M_1 A.$$

and hence that

$$L = (M_{n-1} \cdots M_1)^{-1} = M_1^{-1} M_2^{-1} \cdots M_{n-1}^{-1}$$

Inverting the Eta Matrices

It is easy to see that

$$M_k^{-1} = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & m_{k+1,k} & \ddots & \\ & & \vdots & & \\ & & m_{n,k} & & 1 \end{bmatrix}$$

and that

$$L = \sum_{k=1}^{n-1} -M_k$$

We can easily store the entries of L within the transformed matrix $A^{(k)}$ as we go and at the end have both L and U available with no auxiliary storage required.

Solving with Multiple RHS's

- Suppose we wish to solve the system $Ax = b$ with multiple RHS vectors.
- Calculate an LU factorization.
- Use it to solve the system with various RHS's.
- Avoid computing A^{-1}
  - Takes more computation (takes longer)
  - More round-off error
  - Usually completely dense