

# Propagation of sparse

- Matlab
  - $\text{op}(\text{sparse}) \rightarrow \text{sparse}$
  - $\text{op}(\text{sparse}, \text{sparse}) \rightarrow \text{sparse}$
  - $\text{op}(\text{sparse}, \text{dense}) \rightarrow \text{dense}$
- SpMV – In general,  $V$  is dense. Returns dense
  - Well developed algorithm.
- SpMM – Sparsity of return is variable.
  - Algorithms still developing.
- Sparse BLAS: <http://math.nist.gov/spblas/>
  - Incomplete compared to dense

# Next: Conjugate Gradient Descent

- We want a method to solve  $Ax = b$  which converges quickly.
- The problem with gradient descent is that the algorithm usually chooses a bad direction vector, leading to zig-zag convergence.

An Introduction to  
the Conjugate Gradient Method  
Without the Agonizing Pain  
Edition 1 $\frac{1}{4}$

Jonathan Richard Shewchuk  
August 4, 1994

## Conjugate Gradient Notes

Dec 2018 – Stuart Brorson, [s.brorson@northeastern.edu](mailto:s.brorson@northeastern.edu)

### Preliminaries

Gradient descent and conjugate gradient are algorithms used to solve a system of linear equations,

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\&\vdots \\a_{N1}x_1 + a_{N2}x_2 + a_{N3}x_3 + \cdots + a_{NN}x_N &= b_N\end{aligned}\tag{1a}$$

We usually abbreviate this linear system using matrix notation as

$$Ax = b\tag{1b}$$

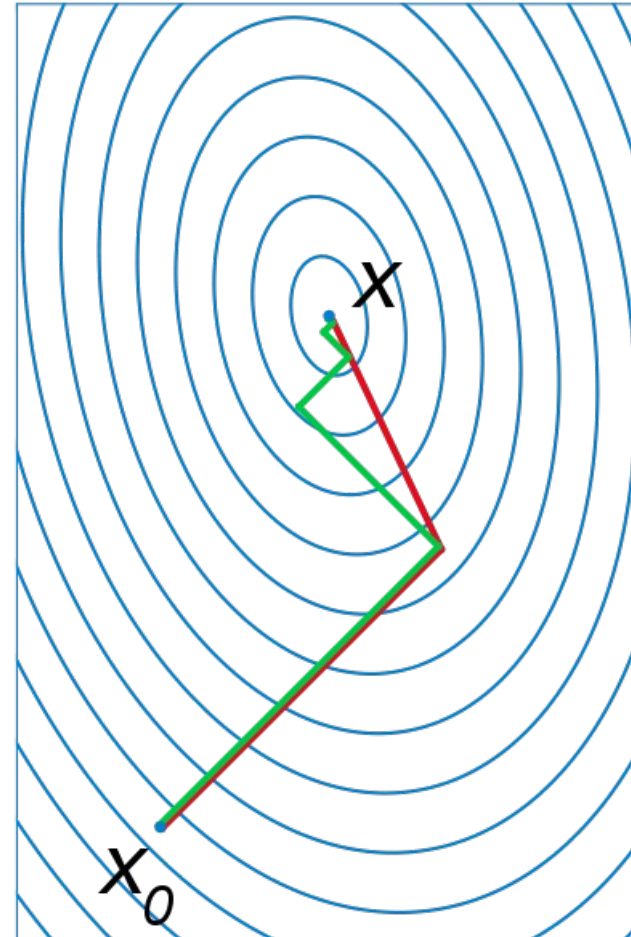
where  $A$  is an  $N \times N$  matrix and  $x$  and  $b$  are (column) vectors.

Both gradient descent and conjugate gradient are applicable in the case where the matrix  $A$  has the following properties:

- Goal: Find a better method to choose direction vector. Do this by modifying the choice of step direction.
- This is departure point for real-world methods.

# Steepest descent vs. conjugate gradient

- Steepest descent zig-zags to solution. This is because we demand each step is perpendicular to the last step.
- CG goes directly to the center (of its search subspace) with each step.



It's all about finding the best step direction

# Conjugate gradient goal

- Find a method which gives us better direction vectors than gradient descent.
- The method will use the residual as a starting point, but then add correction factors to give a better direction.

Gradient descent

$$\vec{r}_n = -\nabla f(\vec{x}_n)$$

$$\vec{x}_{n+1} = \vec{x}_n + \alpha_n \vec{r}_n$$

Conjugate gradient

$$\vec{d}_n = \vec{r}_n + \text{something}$$

$$\vec{x}_{n+1} = \vec{x}_n + \alpha_n \vec{d}_n$$

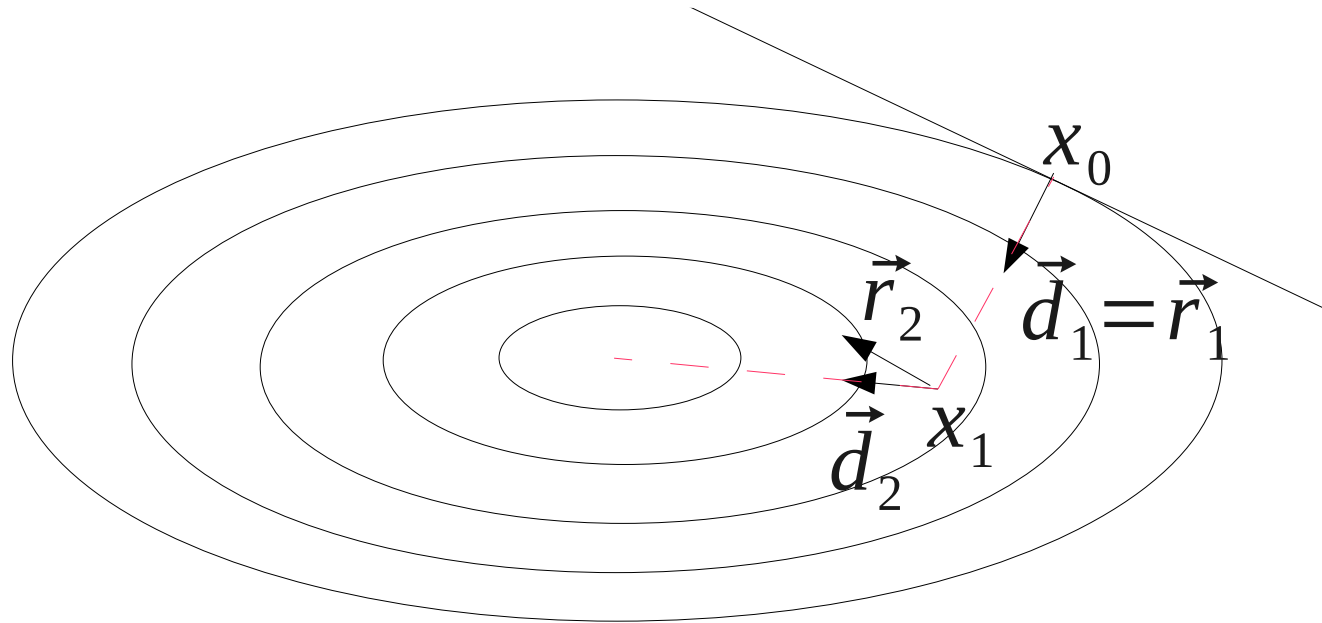
# Necessary concept: conjugate vectors

- Consider two vectors  $u, v$  satisfying

$$u^T A v = 0$$

- These vectors are called “conjugate” with respect to each other under  $A$ , or “ $A$ -orthogonal” .
- You can consider this product to define an inner product between  $u$  and  $v$ .
  - Usual inner product:  $\langle u|v \rangle = u^T v$
  - The  $A$ -inner product:  $\langle u|v \rangle_A = u^T A v$

# CG idea in 2D....



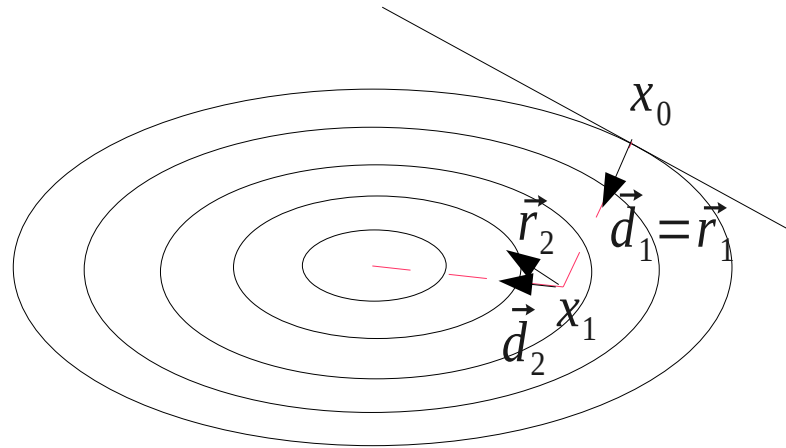
- Idea: make step directions  $d_2, d_1$  A-perpendicular (conjugate).
- Start with  $d_2 = r_2$ . Then remove components from  $d_2$  which are A-parallel to  $d_1$
- **Note that  $d_2$  points to ellipse's center!**

$$d_1 = r_1$$

$$d_2 = r_2 - \frac{\langle r_2 | d_1 \rangle_A}{\langle d_1 | d_1 \rangle_A} d_1$$

Looks like  
Gram-Schmidt

# Why does $d_2$ point to ellipse's center?



- Start at point  $x_1$  is where  $d_1$  is tangent to ellipse.
- Construct ray  $u$  which points to the center.
- Move  $u$  to ellipse center.  $u$  is a radius vector,  $d_1$  is tangent vector.
- Claim: In  $A$ -inner product space,  $u$  is conjugate to  $d_1$ :

$$\langle u | d_1 \rangle_A = u^T A d_1 = 0$$



# Why does $d_2$ point to ellipse's center?

- First: what are these ellipses again?

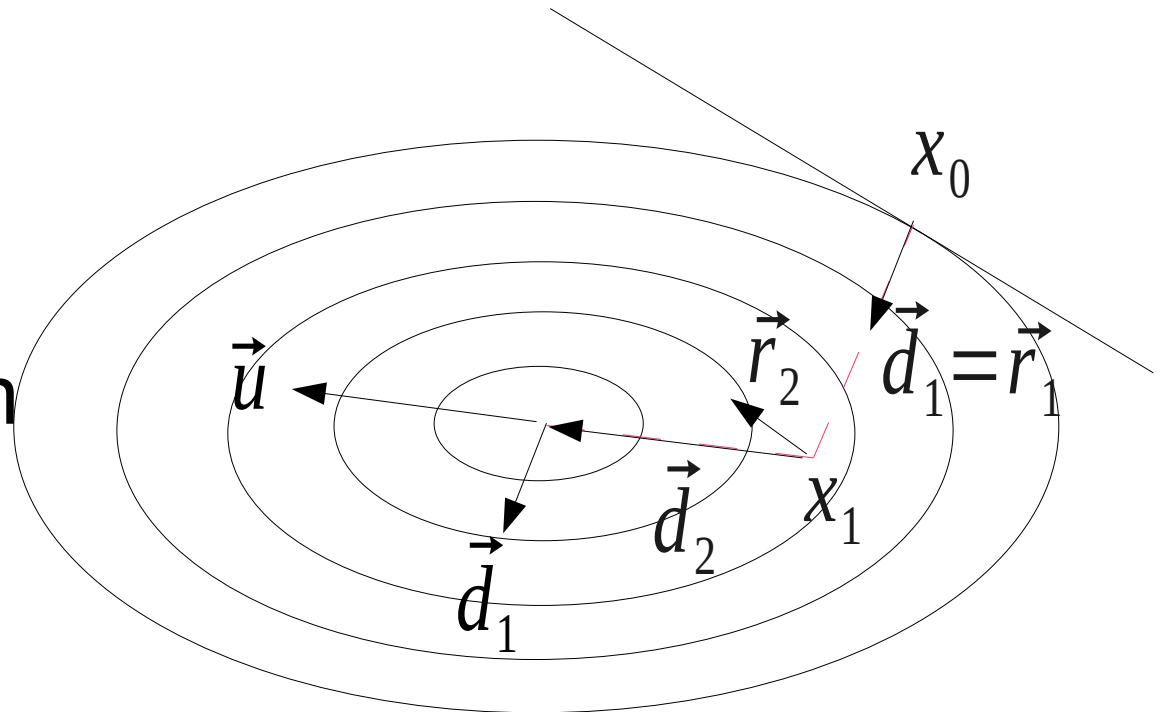
- Problem at hand:  $Ax=b$

- Levelsets of quadratic form:

$$u^T A u = C$$

- Linear transform of unit ball:

$$u = Bx; x^T x = C$$



- Claim:  $\langle u | d_1 \rangle_A = u^T A d_1 = 0$



# Proof that $\langle u | d_1 \rangle_A = 0$

- First, recall two ways to visualize a matrix.

Matrix as transform

$$u = Bx; x^T x = C$$

Quadratic form

$$u^T A u = C$$

Assert

$$A = (B^{-1})^T (B^{-1})$$

With this assertion

$$u^T A u = x^T B^T (B^{-1})^T (B^{-1}) B x = C$$

Assertion:  
relationship  
between two  
visualizations

For SPD,  $(B^{-1})^T = (B^T)^{-1}$

$$u^T A u = x^T B^T (B^T)^{-1} (B^{-1}) B x = C$$

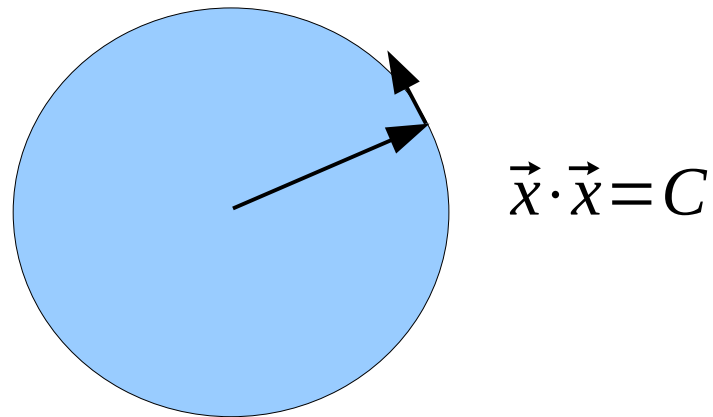
1

1

Therefore,  $u^T A u = x^T x = C$

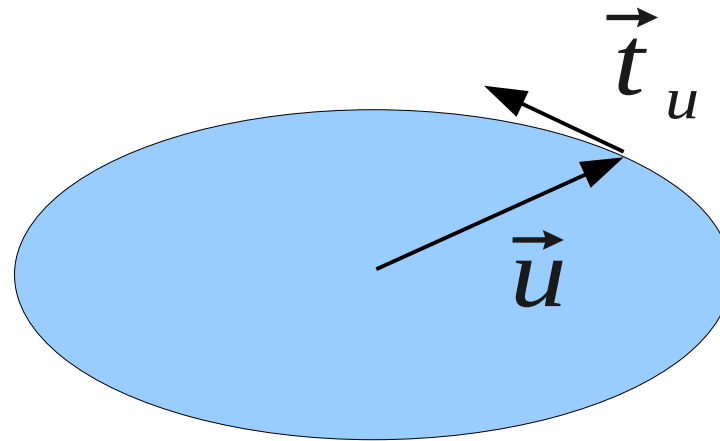
Check!

# Consider radius vector of circle and its tangent



- Radius vectors:  $\vec{x} = \sqrt{C} \begin{pmatrix} \cos \theta \\ \sin \theta \end{pmatrix}$
- Tangent vectors:  $\vec{t}_x = \frac{d\vec{x}}{d\theta} = \sqrt{C} \begin{pmatrix} -\sin \theta \\ \cos \theta \end{pmatrix}$
- Obviously,  $\vec{x} \cdot \vec{t}_x = 0$

From this, we get radius vector  
and its tangent for ellipse



- Radius vector:  $\vec{u} = B \vec{x}$
- Tangent vector:  $\vec{t}_u = \frac{d\vec{u}}{d\theta} = B \frac{d\vec{x}}{d\theta}$   
 $= B \vec{t}_x$

# Compute $\langle u | t_u \rangle_A$

$$\langle u | t_u \rangle_A = u^T A t_u$$

$$= x^T B^T A B t_x$$

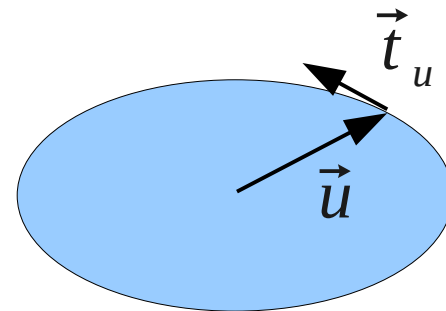
Recall relationship between  
two matrix visualizations

$$A = (B^{-1})^T (B^{-1})$$

$$= x^T B^T (B^T)^{-1} (B^{-1}) B t_x$$

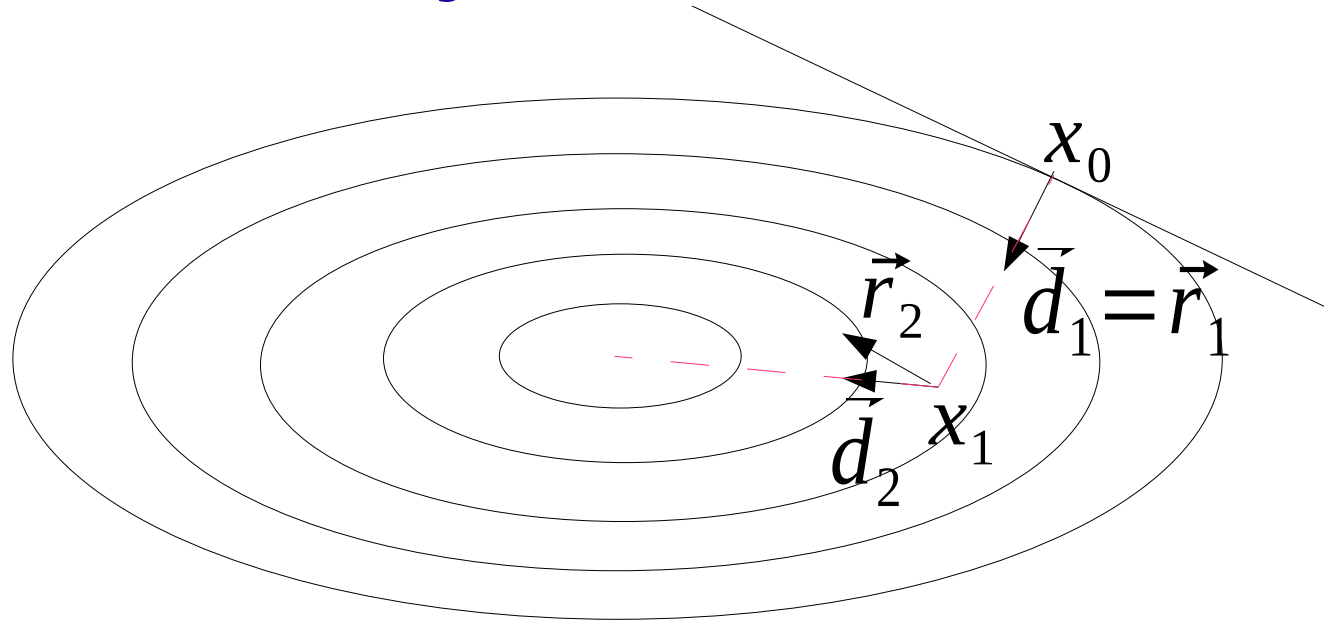
$$= x^T t_x$$

$$= 0$$



- Therefore, in  $A$ -inner product space, radius vector  $u$  is conjugate ( $A$ -perpendicular) to tangent  $t_x$

# What I just showed...



- Idea: make  $d_2$ ,  $d_1$  A-perpendicular (conjugate).
- Start with  $d_2 = r_2$ . Then remove components from  $d_2$  which are A-parallel to  $d_1$
- **$d_2$  points to ellipse's center! It the best next step.**

$$d_1 = r_1$$
$$d_2 = r_2 - \frac{\langle r_2 | d_1 \rangle_A}{\langle d_1 | d_1 \rangle_A} d_1$$

Looks like  
Gram-Schmidt

## Another way to see it: Think about metric of product space

- Consider using basis vectors in directions of eigenvalues of  $A$ .

$$A \hat{e}_n = \lambda_n \hat{e}_n$$

- Expand vectors  $u$  and  $v$  in this basis

$$\vec{u} = \sum_n u_n \hat{e}_n \quad \vec{v} = \sum_m v_m \hat{e}_m$$

- Normal dot product

$$\langle u | v \rangle = u^T v$$

$$= \sum_n \sum_m u_n v_m \hat{e}_n \cdot \hat{e}_m$$

$$= \sum_n u_n v_n$$

- A-dot product

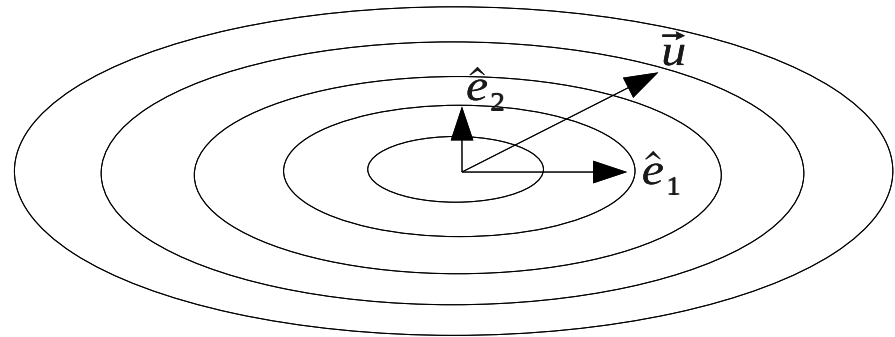
$$\langle u | v \rangle_A = u^T A v$$

$$= \sum_n \sum_m u_n \hat{e}_n \cdot v_m A \hat{e}_m$$

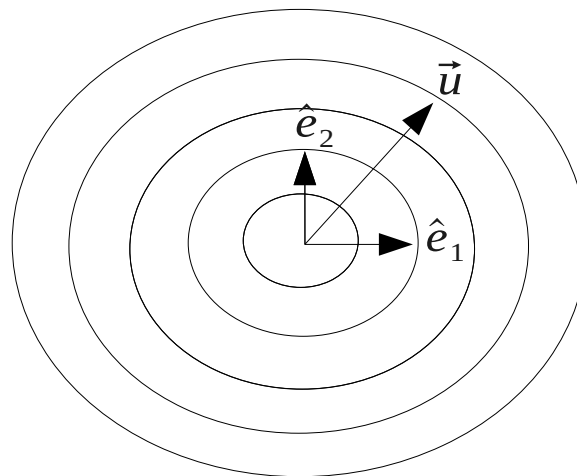
$$= \sum_n \lambda_n u_n v_n$$

# A-dot product weighs certain directions more than others

$$\begin{aligned}\langle u|v\rangle_A &= u^T A v \\ &= \sum_n \lambda_n u_n v_n\end{aligned}$$



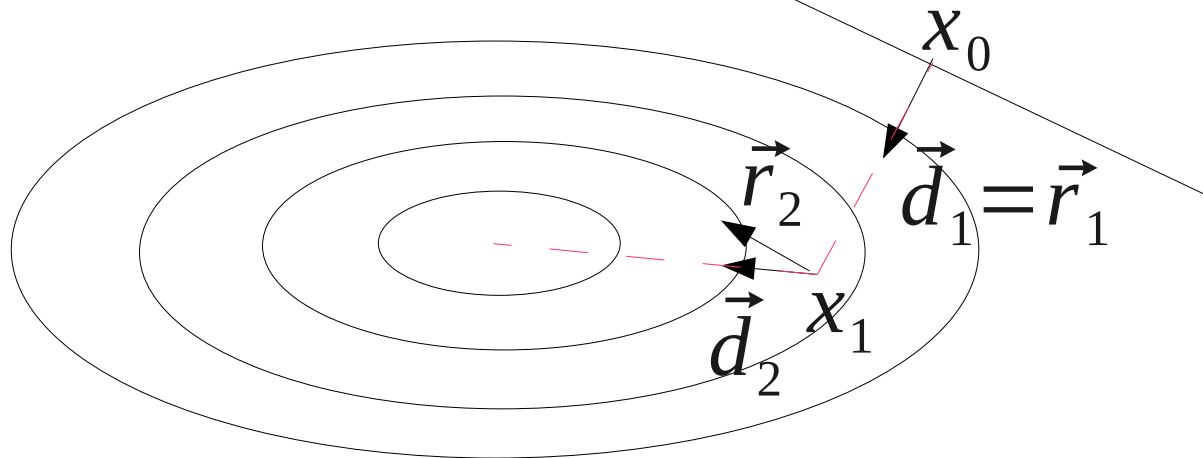
- Therefore, the ellipses of  $u^T A v$  look like circles in A-product space



Recall doing gradient descent on circles is fast.



# Conjugate gradient goal



- We want to find a way to construct step direction  $d$  (instead of  $r$ ) at every step of the method.
- But so far I have only said that if you have  $d$  you should use it. I haven't said how to construct it.

# Next concept: Krylov subspaces

Consider gradient descent.

- Step 1: direction vector and new point:

$$d_0 = r_0 = b - A x_0 \qquad x_1 = x_0 + \alpha_0 d_0$$

- Step 2: direction vector and new point:

$$\begin{aligned} d_1 &= r_1 = b - A x_1 & x_2 &= x_1 + \alpha_1 d_1 \\ &= b - A (x_0 + \alpha_0 d_0) \end{aligned}$$

- Step 3: direction vector and new point:

$$\begin{aligned} d_2 &= r_2 = b - A x_2 & x_3 &= x_2 + \alpha_2 d_2 \\ &= b - A (x_1 + \alpha_1 d_1) \\ &= b - A (x_1 + \alpha_1 (b - A (x_0 + \alpha_0 d_0))) \end{aligned}$$

# Krylov subspace

- Note that each successive iteration introduces a new factor of  $A^n$

$$d_2 = b - A(x_1 + \alpha_1(b - A(x_0 + \alpha_0 d_0)))$$

- It can be shown that the sequence of vectors

$$[d, A d, A^2 d, A^3 d, \dots]$$

spans the N dimensional column space of A.

- This is the space in which the solution  $x^*$  lives.
- This space is called a “Krylov space”.

```
octave:50> d1 = randn(4, 1)
d1 =
```

```
-0.960278
 0.579511
-0.605153
-0.043011
```

Sequence of 4 vectors

$$[d, Ad, A^2d, A^3d]$$

spans a 4D space.

```
octave:51> A = randn(4)
A =
```

```
 0.3057866 -0.3041886  0.4102302  0.0321047
-0.5584887  1.9949554  2.3966330  0.6288215
-0.6211852  1.0898562  0.0018969 -0.1795250
 1.2526069 -0.5013742 -1.4480352  0.0908180
```

```
octave:52> d2 = A*d1;
octave:53> d3 = A*d2;
octave:54> d4 = A*d3;
octave:55> B = [d1, d2, d3, d4]
B =
```

```
-0.960278 -0.719554  0.201122 -0.737956
 0.579511  0.215026  3.399360  6.780682
-0.605153  1.234667  0.795155  4.093639
-0.043011 -0.621025 -2.853368 -2.862974
```

```
octave:56> rank(B)
ans =  4
```

# Krylov and direction vectors

- Every iteration opens a new dimension in which to search.
- Each successive direction vector has components pointing in a new direction.
- Conjugate gradient exploits this fact because each iteration finds the minimum point in the newly-opened dimension in one step.
  - One step because of the A-product space property.

# Krylov subspaces

The CG iteration generates residuals which increase the dimensionality of the search space by 1 with each iteration.

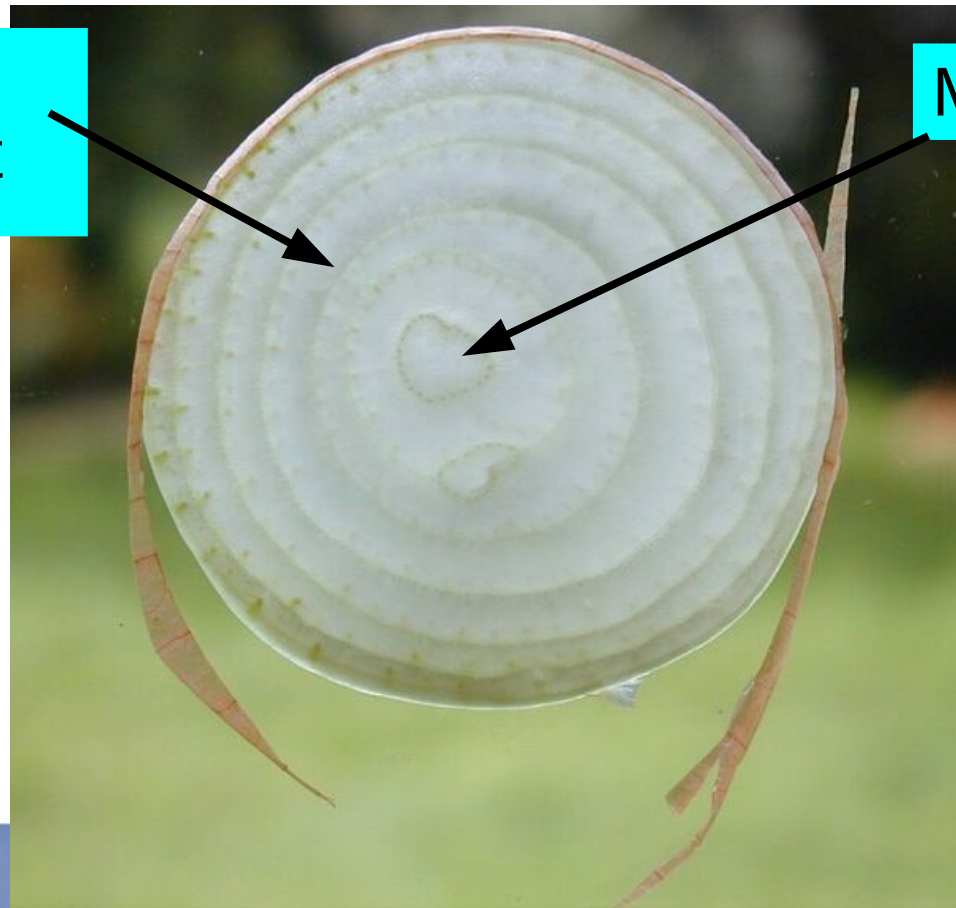
- Therefore, the first step optimizes  $x$  in a 1D space...
- The next step optimizes  $x$  in a 2D space...
- The next step optimizes  $x$  in a 3D space...
- etc.

Think about iterative process to find center of an onion (in 3D)

$$f(x) = \frac{1}{2} x^T A x - x^T b$$

Contours of  
equal height

Minimum  $x^*$





## Next: Spanning the column space of A

- We have an inner product  $\langle u|v\rangle_A = u^T A v$ .
- We will generate a set of direction vectors  $\vec{d}_n$ .
- The direction vectors obey  $\langle \vec{d}_m | \vec{d}_n \rangle_A = d_m^T A d_n = \delta_{mn}$
- Therefore, we have a basis set for a vector space, and can expand any vector in that set.
- Call the solution to  $A \cdot x = b$  the variable  $x^*$ .

Then

$$x^* = \sum_n \alpha_n \vec{d}_n$$

$$b = A x^* = \sum_n \alpha_n A \vec{d}_n$$

Remember this – I'll use it in a second...

# Consider an iteration to find $x^*$

- Take step direction  $d$

$$x^* = \sum_n \alpha_n d_n = \alpha_1 d_1 + \alpha_2 d_2 + \alpha_3 d_3 + \alpha_4 d_4 + \dots$$


The diagram illustrates the iterative construction of  $x^*$  from a set of conjugate directions. It shows three equations stacked vertically, connected by V-shaped lines that indicate the cumulative addition of terms:

$$x_1$$
$$x_2 = x_1 + \alpha_2 d_2$$
$$x_3 = x_2 + \alpha_3 d_3$$

This expresses  $x^*$  as set of steps – all conjugate to each other.

- To make this work, we need to find  $\alpha$  and  $d$  at each step.
- We want successive  $d_n$  to be A-orthogonal to all previous  $d$ . This will avoid the zig-zags.

# Derive step direction $d_i$ from residual $r_i$

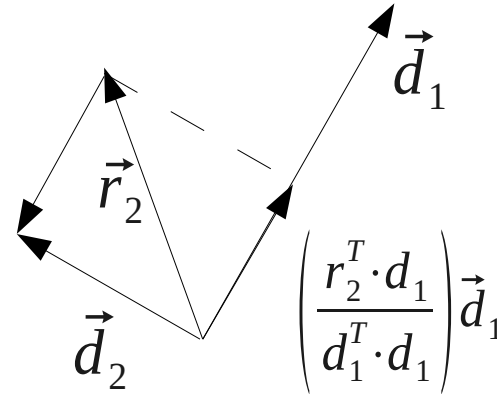
- For any  $n$ , the residual is  $r_n = b - Ax_n$
- Derive new step  $d_i$  from gradient (residual)  $r_i$  at point  $i$ .  
 Start with residual, but modify it....
- Choose every new step  $d_i$  to be A-orthogonal to all previous  $d_j$ :

$$d_i^T A d_j = 0 \quad \text{For all} \quad i > j$$

- That is, the steps  $d_i$  are derived from the residuals  $r_i$ , and are all conjugate to each other.
- How to do this?

# Detour: Recall Gram-Schmidt

$$\begin{aligned}\vec{d}_2 &= \vec{r}_2 - \left( \frac{r_2^T d_1}{d_1^T d_1} \right) \vec{d}_1 \\ &= r_2 - \frac{\langle r_2 | d_1 \rangle}{\langle d_1 | d_1 \rangle} d_1\end{aligned}$$



**Goal:** Find new  $d_2$  perpendicular to  $d_1$  using  $r_2$  as “seed”.

- Start with  $d_1, r_2$ .
- Project  $r_2$  onto  $d_1$ .
- Take projection vector and subtract from  $r_2$ .
- Result is  $d_2$ .

In ND: Repeat for all other vectors  $d_n$

# How to find next step direction $d_{n+1}$ ?

- We derive the new search direction from the gradient (residual).
- We want every step  $d_{n+1}$  to be conjugate (A-orthogonal) to all the previous steps  $d_i$ .
- This expression will accomplish that:

$$d_{n+1} = r_{n+1} - \sum_{i=1}^n \frac{r_{n+1}^T A d_i}{d_i^T A d_i} d_i$$

Looks like Gram-Schmidt, in A-inner product space

$$= r_{n+1} + \sum_{i=1}^n \beta_{n+1,i} d_i \quad \beta_{n+1,i} = -\frac{r_{n+1}^T A d_i}{d_i^T A d_i}$$

Remember this – I'll use it in a second...

## Next: get step length $\alpha$

- Start with  $b = A x^* = \sum_i \alpha_i A d_i$
- Form product  $d_n^T b = d_n^T A x^* = \sum_i \alpha_i d_n^T A d_i$
- Conjugate property gives  $= \alpha_n d_n^T A d_n$
- Therefore, for any set of basis vectors  $d_n$  we have

$$\alpha_n = \frac{d_n^T b}{d_n^T A d_n}$$

- Note that the  $d$  are a set of step vectors, all A-orthogonal to each other

# Update rule for $\alpha$

- At every iteration,  $\alpha_n = \frac{d_n^T b}{d_n^T A d_n}$
- For any  $n$ , the residual is  $r_n = b - A x_n$

- So  $\alpha_n = \frac{d_n^T (r_n + A x_n)}{d_n^T A d_n}$

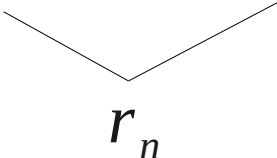
$$= \frac{d_n^T r_n}{d_n^T A d_n}$$

By A-orthogonality –  $x_m$  was composed of previous steps in the  $d_n$  direction

Remember this – I'll use it in a second...



# Update the residual at each step

- In general,  $r_{n+1} = b - A x_{n+1}$   
$$r_{n+1} = b - A(x_n + \alpha_n d_n)$$

$$r_n$$

- So, residual update rule

$$r_{n+1} = r_n - \alpha_n A d_n$$

- One can also write

$$A d_n = \frac{1}{\alpha_n} (r_n - r_{n+1})$$

Remember this – I'll use it in a second...

# Compute beta (to update d)

- From last pages

$$\beta_{n+1,i} = -\frac{r_{n+1}^T A d_i}{d_i^T A d_i} \quad A d_i = \frac{1}{\alpha_i} (r_i - r_{i+1})$$

- Substituting,

$$\beta_{n+1,i} = -\frac{r_{n+1}^T (r_i - r_{i+1})}{\alpha_i (d_i^T A d_i)}$$

- Recall sum for  $d_{n+1}$  runs to  $n$ , and use orthogonality

$$\beta_{n+1,i} = -\frac{r_{n+1}^T r_{i+1}}{\alpha_i (d_i^T A d_i)} \quad r_i^T r_j = 0 \text{ if } i \neq j$$

- Finally, substitute in expression for alpha

$$\alpha_n = \frac{d_n^T r_n}{d_n^T A d_n} \quad \beta_{n+1} = -\frac{r_{n+1}^T r_{n+1}}{d_n^T r_n}$$

# Some cleanup algebra

- An identity  $r_n^T d_n = r_n^T r_n + r_n^T (d_n - r_n)$

- Expression for beta:

$$d_n = r_n + \beta_{n-1} d_{n-1} \quad d_n - r_n = \beta_{n-1} d_{n-1}$$

- Substituting into identity

$$r_n^T d_n = r_n^T r_n + r_n^T \beta_{n-1} d_{n-1}$$

$$= r_n^T r_n$$

$$d_n^T r_n = r_n^T r_n$$

Desired  
results

- Therefore, new expressions for alpha & beta:

$$\alpha_n = \frac{r_n^T r_n}{d_n^T A d_n}$$

$$\beta_{n+1} = -\frac{r_{n+1}^T r_{n+1}}{r_n^T r_n}$$

# CG moving parts

## Compare to gradient descent

- Step distance

$$\alpha_n = \frac{r_n^T r_n}{d_n^T A d_n}$$

Compute  $A d_n$  once per iteration

- Position

$$x_{n+1} = x_n + \alpha_n d_n$$

- Residual

$$r_{n+1} = r_n - \alpha_n A \cdot d_n$$

- Step direction

$$\beta_{n+1} = -\frac{r_{n+1}^T r_{n+1}}{r_n^T r_n}$$


$$d_{n+1} = r_{n+1} + \beta_{n+1} d_n$$

Step direction uses “conjugate” to direction of gradient (residual)

# Conjugate Gradient Algorithm

1. Start with initial guess  $x_0$
2. Compute gradient (residual) at  $x_0$ :  $r_0 = b - Ax_0$
3. Generate initial direction vector  $d_0$ :  $d_0 = r_0$
4. Start main loop here
5. Compute  $Ad$  product:  $z = Ad_n$
6. Compute step length:  $\alpha_n = \frac{r_n^T r_n}{d_n^T z}$
7. Step to next point:  $x_{n+1} = x_n + \alpha_n d_n$

Temporary  
convenience  
variable



# Conjugate Gradient Algorithm cont'd...

7. Compute new residual:

$$r_{n+1} = r_n - \alpha_n z$$

8. Check for convergence,  
and exit loop if satisfactory

$$\|r_{n+1}\| < tolerance$$

9. Compute new beta:

$$\beta_{n+1} = -\frac{r_{n+1}^T r_{n+1}}{r_n^T r_n}$$

10. Compute next step direction:

$$d_{n+1} = r_{n+1} + \beta_n d_n$$

11. Loop back to start of loop (step 4)

```

function xnp1 = mycg( A, b, tol )
    % This is my implementation of the basic CG algorithm.

    % Initialize calc
    xn = zeros(size(b)); % Initial first guess
    rn = b;              % b - A*x with x0 = zeros.
    dn = rn;
    for idx = 1:200
        z = A*dn;          % Compute A*d product -- do only one matmul
        alphan = (rn'*rn)/(dn'*z); % Update alpha
        xnp1 = xn + alphan*dn; % Take step
        rnp1 = rn - alphan*z; % Update residual

        % Check for convergence: residual < tol
        if (norm(rnp1) < tol)
            fprintf(' Niter = %d ', idx)
            return
        end

        betan = (rnp1'*rnp1)/(rn'*rn); % Compute updated beta
        dnp1 = rnp1 + betan*dn;        % Compute new step direction

        % Now push all n+1 values to n, then loop again.
        dn = dnp1;
        rn = rnp1;
        xn = xnp1;

    end
    error('Failed to converge!')
end

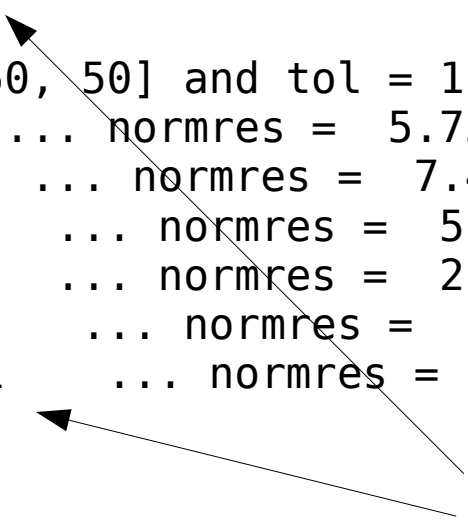
```



```

>> test_mysd
---- Test gradient descent with matrix [50, 50] and tol = 1.000000e-05 ---
K = 4.000000... Niter = 26      ... normres = 3.223149e-05.
K = 25.000000... Niter = 178    ... normres = 2.565673e-05.
K = 100.000000... Niter = 674   ... normres = 2.015559e-05.
K = 400.000000... Niter = 2034  ... normres = 2.164351e-05.
K = 2500.000000... Niter = 16162 ... normres = 2.232956e-05.
K = 10000.000000... Niter = 59902 ... normres = 2.976023e-05.
>> test_mycg
----- Test cg with matrix [50, 50] and tol = 1.000000e-05 -----
K = 4.000000... Niter = 15      ... normres = 5.758712e-06.
K = 25.000000... Niter = 35     ... normres = 7.470778e-06.
K = 100.000000... Niter = 51    ... normres = 5.633881e-06.
K = 400.000000... Niter = 56    ... normres = 2.536377e-07.
K = 2500.000000... Niter = 59   ... normres = 1.792248e-06.
K = 10000.000000... Niter = 61  ... normres = 1.932195e-06.

```



For systems with equivalent conditioning, SD converges much more slowly than CG.

- $K$  = matrix condition number
- Niter = iterations to convergence
- Normres = norm of residual =  $\|b - Ax\|$

# Remarks

- Many variations of this algorithm exist out there.
- Convergence is very fast in general.
- CG is used by Matlab. Look at PCG, BiCG, and other variants which are part of the standard Matlab solver library.
- CG is one of several algorithms referred to as “Krylov methods”. All involve working in vector space generated by  $\{d, Ad, A^2d, A^3d, \dots\}$

# Comments about CG mathematics

- A-orthogonality means you are operating in a space where the ellipses defined by  $u^T A u$  look like circles.
- CG derives step direction from gradient (residual) via Gram-Schmidt process using A-dot product.
- In A-product space, circle center is found in one step upon finding  $d_n$ , meaning fast convergence *per dimension*.
- Successive iterations increase the dimension of the search space by 1.
- Therefore, CG is guaranteed to find  $x^*$  in N steps (assuming no round-off errors).
- However, the magic of CG is that it finds an excellent approximation to  $x^*$  in only a few steps.

# Final remarks on CG

- Good for solving sparse systems – most expensive operation is single sparse matmul per iteration.
- Matlab: pcg
- Many important solvers are derived from CG, such as bicg, cgs, bicgstab, etc.

## Conjugate Gradient Notes

Dec 2018 – Stuart Brorson, [s.brorson@northeastern.edu](mailto:s.brorson@northeastern.edu)

### Preliminaries

Gradient descent and conjugate gradient are algorithms used to solve a system of linear equations,

$$\begin{aligned}a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\&\vdots \\a_{N1}x_1 + a_{N2}x_2 + a_{N3}x_3 + \cdots + a_{NN}x_N &= b_N\end{aligned}\tag{1a}$$

We usually abbreviate this linear system using matrix notation as

$$Ax = b\tag{1b}$$

where  $A$  is an  $N \times N$  matrix and  $x$  and  $b$  are (column) vectors.

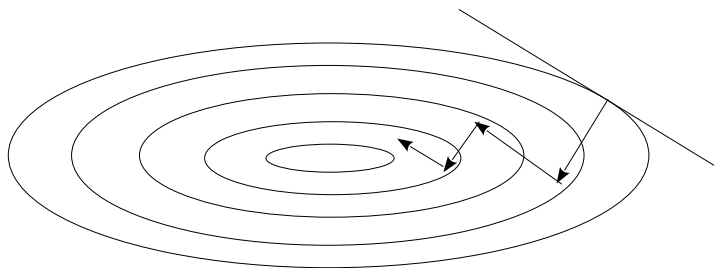
Both gradient descent and conjugate gradient are applicable in the case where the matrix  $A$  has the following properties:

An Introduction to  
the Conjugate Gradient Method  
Without the Agonizing Pain  
Edition 1  $\frac{1}{4}$

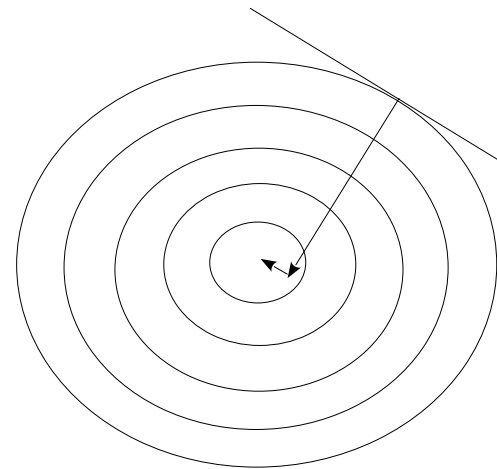
Jonathan Richard Shewchuk  
August 4, 1994

# Next topic: Preconditioners for iterative solvers

- Iterative solvers: Convergence rate depends upon matrix condition number.
  - Visualization: Finding the minimum of a squeezed ellipse is more difficult than finding the minimum of a near-circle.



High  $K$



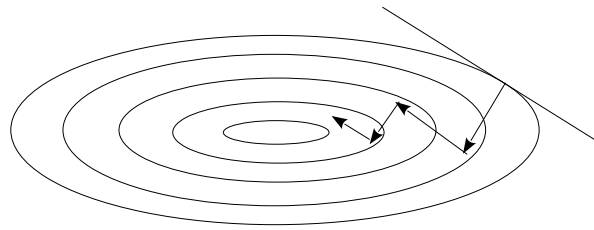
Low  $K$

# Idea: Solve $Ax = b$ using a better matrix

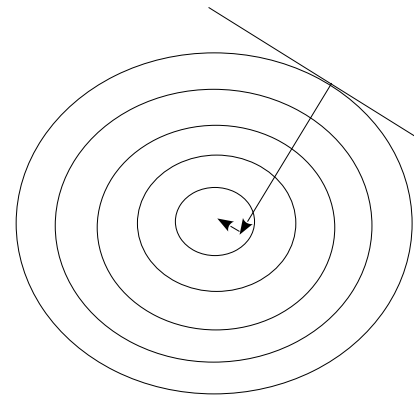
- Modify problem so the objective function looks more like a circle.

$$Ax = b \rightarrow (P^{-1}A)x = P^{-1}b$$

Replace                      Equivalent equation with better conditioning



High K



Low K

- Even conjugate gradient is sensitive to round-off errors for badly-conditioned matrices.

# Preconditioning

Replace  $Ax=b$  with equivalent equation with better conditioning for faster convergence.

$$Ax=b \rightarrow (P^{-1}A)x=P^{-1}b$$

- Choose matrix  $P$  so  $P^{-1}A$  is better conditioned than  $A$  alone.
- Choose matrix  $P$  which is easy to invert.
- Note that solution  $x$  is the same for both equations.
- What to choose for  $P$ ? A few examples:
  - Jacobi preconditioner:  $P = \text{diag}(A)$
  - Gauss-Seidel
  - Incomplete LU (ilu)

# Jacobi Preconditioner

- Use diagonal elements of  $A$

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \quad P = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{pmatrix}$$

- Inverse of  $P$  is trivial to compute

$$P^{-1} = \begin{pmatrix} 1/a_{11} & 0 & 0 & 0 \\ 0 & 1/a_{22} & 0 & 0 \\ 0 & 0 & 1/a_{33} & 0 \\ 0 & 0 & 0 & 1/a_{44} \end{pmatrix}$$



# Jacobi Preconditioner

- Equation to solve:

$$P^{-1} \cdot A x = P^{-1} b$$

- Written out:

$$\begin{pmatrix} 1/a_{11} & 0 & 0 & 0 \\ 0 & 1/a_{22} & 0 & 0 \\ 0 & 0 & 1/a_{33} & 0 \\ 0 & 0 & 0 & 1/a_{44} \end{pmatrix} \cdot \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1/a_{11} \\ b_2/a_{22} \\ b_3/a_{33} \\ b_4/a_{44} \end{pmatrix}$$

- Each row is normalized by diag.
- Diagonal element = 1, off-diags are < 1.

# Preconditioning code

```
% Generate symmetric positive def matrix from A
A = sprandn_svdcond(n, m, testcond);

S = A'*A;
condS = cond(S);

Pm1 = diag(1./diag(S));

b = 10*randn(n, 1);

Pm1S = Pm1*S;
Pm1b = Pm1*b;

fprintf('Testing matrix with condition number %f...', condS);

x = mysd(Pm1S, Pm1b, tol);

Nres = norm(b - S*x);
fprintf('    ... norm of residual = %f.\n', Nres);
```

# Jacobi preconditioner vs. no preconditioner

```
>> test_mysd
--- Test gradient descent with matrix [25, 25] and tol = 1.000000e-05 ---
K = 4.000000... Niter = 28 ... normres = 6.735709e-06.
K = 25.000000... Niter = 177 ... normres = 9.871077e-06.
K = 100.000000... Niter = 595 ... normres = 9.860751e-06.
K = 400.000000... Niter = 2506 ... normres = 9.930844e-06.
K = 2500.000000... Niter = 16611 ... normres = 9.992518e-06.
K = 10000.000000... Niter = 70000 ... normres = 9.999191e-06.
K = 40000.000000... Niter = 282900 ... normres = 9.999642e-06.

>> test_mysd_JacobiPreconditioner
- Test preconditioned gradient descent with matrix [25, 25] and tol = 1.000000e-05 -
K = 4.000000... Niter = 22 ... normres = 5.681206e-06.
K = 25.000000... Niter = 119 ... normres = 9.338814e-06.
K = 100.000000... Niter = 267 ... normres = 9.647971e-06.
K = 400.000000... Niter = 2387 ... normres = 9.986772e-06.
K = 2500.000000... Niter = 4492 ... normres = 9.995395e-06.
K = 10000.000000... Niter = 20103 ... normres = 9.998301e-06.
K = 40000.000000... Niter = 49822 ... normres = 9.999556e-06.
```

```
>> test_ilu
```

Af =

4	-1	0	-1	0	0
-1	4	-1	0	-1	0
0	-1	4	0	0	-1
-1	0	0	4	-1	0
0	-1	0	-1	4	-1
0	0	-1	0	-1	4

K2D



# ILU(0)

- Do LU decomposition, but keep only nonzero elements at nonzero positions in A

L =

1.0000	0	0	0	0	0
-0.2500	1.0000	0	0	0	0
0	-0.2667	1.0000	0	0	0
-0.2500	-0.0667	-0.0179	1.0000	0	0
0	-0.2667	-0.0714	-0.2871	1.0000	0
0	0	-0.2679	-0.0048	-0.3160	1.0000

U =

4.0000	-1.0000	0	-1.0000	0	0
0	3.7500	-1.0000	-0.2500	-1.0000	0
0	0	3.7333	-0.0667	-0.2667	-1.0000
0	0	0	3.7321	-1.0714	-0.0179
0	0	0	0	3.4067	-1.0766
0	0	0	0	0	3.3919

# ILU(0)

`[Li, Ui, Pi] = ilu(A);`

`Li =`

1.0000	0	0	0	0	0
-0.2500	1.0000	0	0	0	0
0	-0.2667	1.0000	0	0	0
-0.2500	0	0	1.0000	0	0
0	-0.2667	0	-0.2667	1.0000	0
0	0	-0.2679	0	-0.2885	1.0000

`Ui =`

4.0000	-1.0000	0	-1.0000	0	0
0	3.7500	-1.0000	0	-1.0000	0
0	0	3.7333	0	0	-1.0000
0	0	0	3.7500	-1.0000	0
0	0	0	0	3.4667	-1.0000
0	0	0	0	0	3.4437

Af =

4	-1	0	-1	0	0
-1	4	-1	0	-1	0
0	-1	4	0	0	-1
-1	0	0	4	-1	0
0	-1	0	-1	4	-1
0	0	-1	0	-1	4

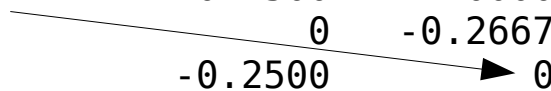
L =

1.0000	0	0	0	0	0
-0.2500	1.0000	0	0	0	0
0	-0.2667	1.0000	0	0	0
-0.2500	-0.0667	-0.0179	1.0000	0	0
0	-0.2667	-0.0714	-0.2871	1.0000	0
0	0	-0.2679	-0.0048	-0.3160	1.0000

Li =

1.0000	0	0	0	0	0
-0.2500	1.0000	0	0	0	0
0	-0.2667	1.0000	0	0	0
-0.2500	0	0	1.0000	0	0
0	-0.2667	0	-0.2667	1.0000	0
0	0	-0.2679	0	-0.2885	1.0000

Zero  
when A is  
zero



# ILU(0)

- L, U nonzero only when A is nonzero.
- Decomposition is only approximate.

$$LU \approx A$$

- $L^{-1}$ ,  $U^{-1}$  easy to compute by backsubstitution since they are triangular.
- Preconditioned matrix ( $U^{-1}L^{-1} A$ ) has better conditioning than A alone.

# Preconditioner remarks

- What preconditioner to use? No simple answer, depend upon nature of matrix  $A$ .....
- Usually applied to equation as part of advanced method, e.g. conjugate gradient.
  - Example: Matlab's pcg has the preconditioner built into the code.
- Becomes important for large systems.



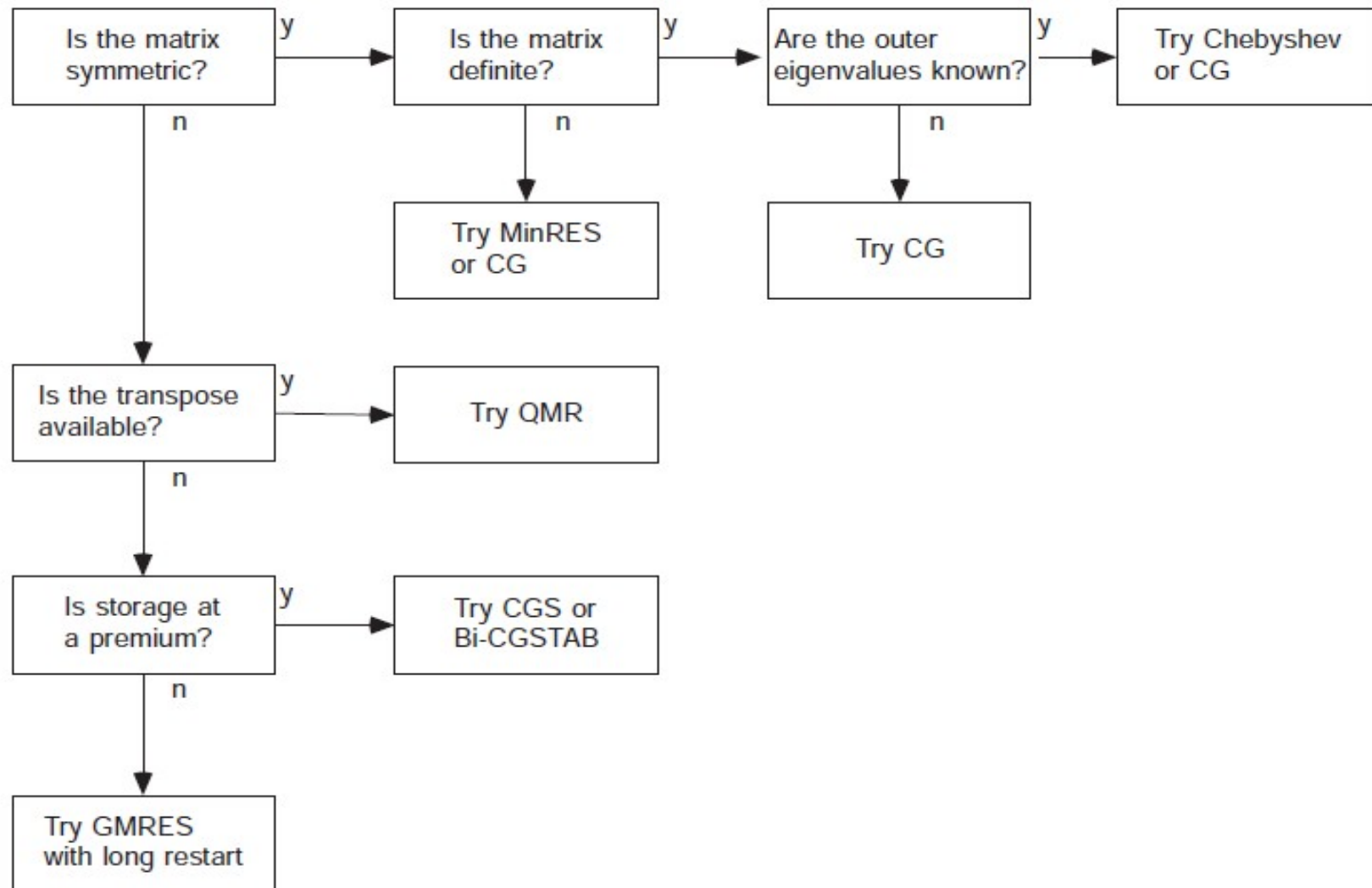
# Iterative solvers you should know

Method	Lapack name(s)	When to use
Conjugate Gradient	CG	Use for symmetric positive definite.
Minimum Residual Symmetric LQ	MINRES SYMMLQ	Symmetric positive indefinite.
Conjugate Gradient on the Normal Equations	CGNE CGNR	Used for $(A^T A)x = b$ .
Generalized Minimum Residual	GMRES	General non-symmetric matrices. Uses lots of memory.
BiConjugate Gradient	BiCG	Non-symmetric matrices. Uses limited memory. Unstable.
Quasi-Minimal Residual	QMR	Similar to BiCG.
Conjugate Gradient Square	CGS	Variant of BiCG
BiConjugate Gradient Stabilized	BiCGSTAB	Variant of BiCG with better convergence properties.

These are in LAPACK

# Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods<sup>1</sup>

<http://www.netlib.org/templates/templates.pdf>



# This session's topics

- Conjugate gradient algorithm
  - Concepts: Conjugate vectors & inner product spaces.
  - Algorithm is complicated.
  - “Stretches” axes of vector space so the step direction points directly to the center of the subspace you are working in.
  - CG and its derivatives are used in Matlab.
- Preconditioners for linear solvers.