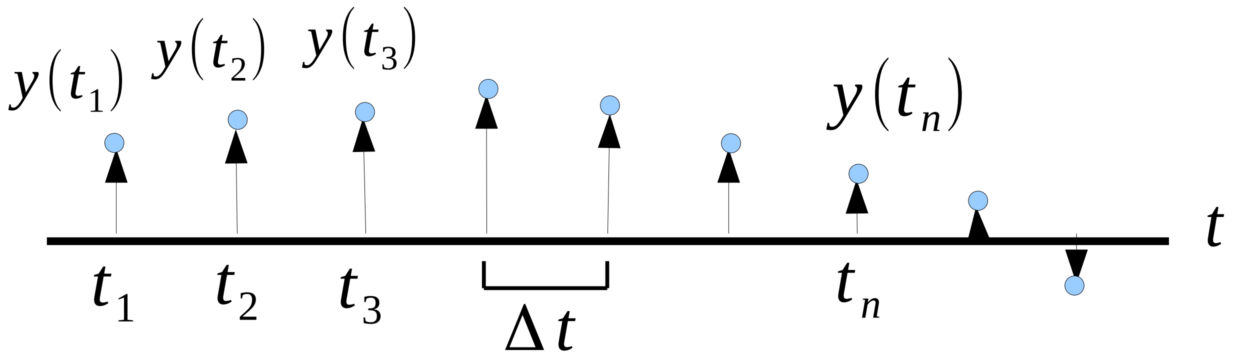
Quick aside: Sampled data and numerical derivatives



- Use Taylor's series to derive:
  - Forward difference
  - Backward difference
  - Two-sided difference (symmetric difference)
- Note truncation error from each

Computing the first derivative

- Derive on blackboard:
  - Forward difference

Drop

$$\left.\frac{df}{dx}\right|_x = \frac{f(x+h)-f(x)}{h} \cancel{- \frac{h}{2} f''(x) - \frac{h^2}{6} f'''(x) + \cdots}$$

  - Backward difference

Drop

$$\left.\frac{df}{dx}\right|_x = \frac{f(x)-f(x-h)}{h} \cancel{- \frac{h}{2} f''(x) + \frac{h^2}{6} f'''(x) + \cdots}$$

  - Two-sided difference

Drop

$$\left.\frac{df}{dx}\right|_x = \frac{f(x+h)-f(x-h)}{2\text{h}} \cancel{- \frac{h^2}{6} f'''(x) + \cdots}$$

Approximations using more points

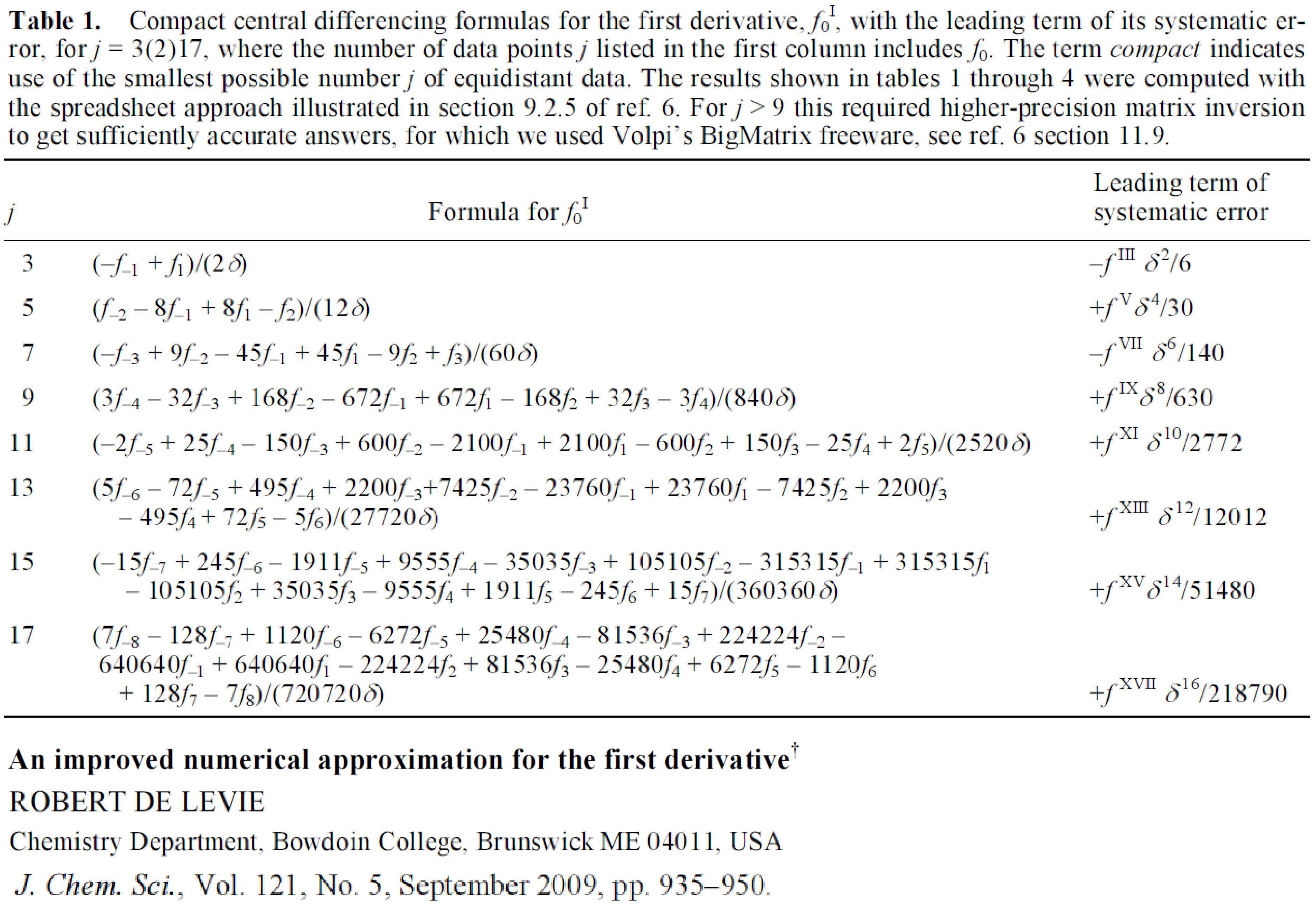


Table 1. Compact central differencing formulas for the first derivative, f_0^{I}, with the leading term of its systematic error, for $j = 3(2)17$, where the number of data points j listed in the first column includes f_0. The term *compact* indicates use of the smallest possible number j of equidistant data. The results shown in tables 1 through 4 were computed with the spreadsheet approach illustrated in section 9.2.5 of ref. 6. For $j > 9$ this required higher-precision matrix inversion to get sufficiently accurate answers, for which we used Volpi's BigMatrix freeware, see ref. 6 section 11.9.

j	Formula for f_0^{I}	Leading term of systematic error
3	$(-f_{-1}+f_1)/(2\delta)$	$-f^{\mathrm{III}}\,\delta^2/6$
5	$(f_{-2}-8f_{-1}+8f_1-f_2)/(12\delta)$	$+f^{\mathrm{V}}\,\delta^4/30$
7	$(-f_{-3}+9f_{-2}-45f_{-1}+45f_1-9f_2+f_3)/(60\delta)$	$-f^{\mathrm{VII}}\,\delta^6/140$
9	$(3f_{-4}-32f_{-3}+168f_{-2}-672f_{-1}+672f_1-168f_2+32f_3-3f_4)/(840\delta)$	$+f^{\mathrm{IX}}\,\delta^8/630$
11	$(-2f_{-5}+25f_{-4}-150f_{-3}+600f_{-2}-2100f_{-1}+2100f_1-600f_2+150f_3-25f_4+2f_5)/(2520\delta)$	$+f^{\mathrm{XI}}\,\delta^{10}/2772$
13	$(5f_{-6}-72f_{-5}+495f_{-4}+2200f_{-3}+7425f_{-2}-23760f_{-1}+23760f_1-7425f_2+2200f_3-495f_4+72f_5-5f_6)/(27720\delta)$	$+f^{\mathrm{XIII}}\,\delta^{12}/12012$
15	$(-15f_{-7}+245f_{-6}-1911f_{-5}+9555f_{-4}-35035f_{-3}+105105f_{-2}-315315f_{-1}+315315f_1-105105f_2+35035f_3-9555f_4+1911f_5-245f_6+15f_7)/(360360\delta)$	$+f^{\mathrm{XV}}\,\delta^{14}/51480$
17	$(7f_{-8}-128f_{-7}+1120f_{-6}-6272f_{-5}+25480f_{-4}-81536f_{-3}+224224f_{-2}-640640f_{-1}+640640f_1-224224f_2+81536f_3-25480f_4+6272f_5-1120f_6+128f_7-7f_8)/(720720\delta)$	$+f^{\mathrm{XVII}}\,\delta^{16}/218790$

An improved numerical approximation for the first derivative[†]

ROBERT DE LEVIE

Chemistry Department, Bowdoin College, Brunswick ME 04011, USA

J. Chem. Sci., Vol. 121, No. 5, September 2009, pp. 935–950.

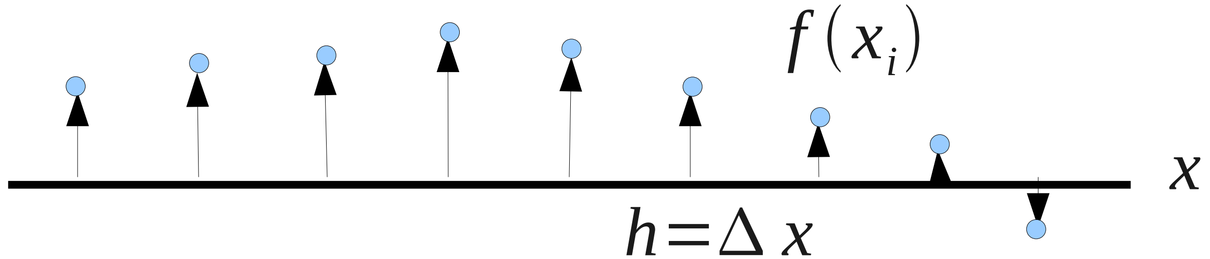
Second derivative

- Derived on blackboard

$$\left.\frac{d^2 f}{dx^2}\right|_x = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + \cancelto{\text{Drop}}{\frac{h^2}{12} f^{(4)}(x)} + \cdots$$

- Truncation error is of order h^2

Derivatives as Matrix Multiplications



- f(x) is a vector of values evaluated at each x$_i$: $[f_{0,} f_{1,} f_{2,} f_{3,} \cdots]$
- Derivative (one-sided): $\frac{1}{h}[f_1 - f_0, f_2 - f_1, f_3 - f_2, \cdots]$

$$\frac{\partial f}{\partial x} = \frac{1}{h}\begin{pmatrix} -1 & 1 & 0 & 0 & \cdots \\ 0 & -1 & 1 & 0 & \cdots \\ 0 & 0 & -1 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}\begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \end{pmatrix}$$

Second derivative

$$\frac{\partial^2 f}{\partial x^2} = \frac{1}{2h} \begin{pmatrix} -2 & 1 & 0 & 0 & \cdots \\ 1 & -2 & 1 & 0 & \cdots \\ 0 & 1 & -2 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \end{pmatrix}$$

$$\frac{1}{2h}[f_1 - 2f_0, f_2 - 2f_1 + f_0, f_3 - 2f_2 + f_1, \cdots]$$

- We will see this again shortly ....
- Note issue with boundary.
- A matrix is a linear operator, so is a derivative.

Main topic: Iterative Matrix Solvers

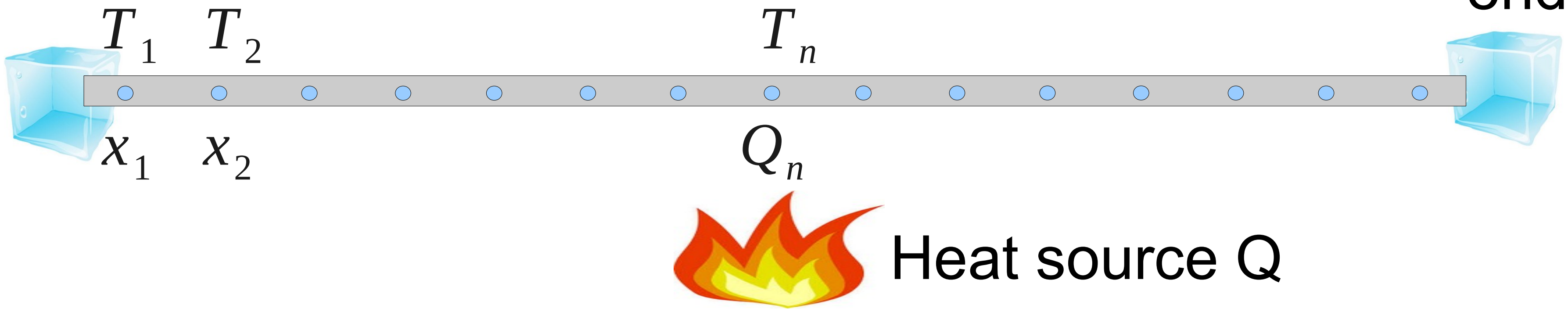
- Our equation to solve: $Ax = b$
  - We know A, b. Want to find x.
- We have done a few “direct methods”:
  - Gaussian elimination
  - LU
  - Cholesky
- Direct methods are typically used for dense systems.
- Iterative solvers are really good for sparse systems.

Example sparse equation: steady-state heat equation in 1D

$$-k\frac{\partial^2 T}{\partial x^2}=Q(x)$$

Consider iron bar with fire under middle.
What is temperature profile T(x)?

BCs on end



Discretize in x:

$$-k\frac{(T_{n+1}-2\mathrm{T}_n+T_{n-1})}{h^2}=Q(x_n)$$

Written in matrix format

$$\frac{-1}{h^2}\begin{bmatrix} -2 & 1 & 0 & 0 & 0 & \cdots \\ 1 & -2 & 1 & 0 & 0 & \cdots \\ 0 & 1 & -2 & 1 & 0 & \cdots \\ 0 & 0 & 1 & -2 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} T = \begin{bmatrix} Q_1/k \\ Q_2/k \\ Q_3/k \\ Q_4/k \\ \vdots \end{bmatrix}$$

- This is linear system of form Ax = b.
- We know Q and k, want to find T.
- We might want to simulate 1000s of points --> This creates sparse matrices of dimension 1000s x 1000s
- Dense solvers can run out of memory!

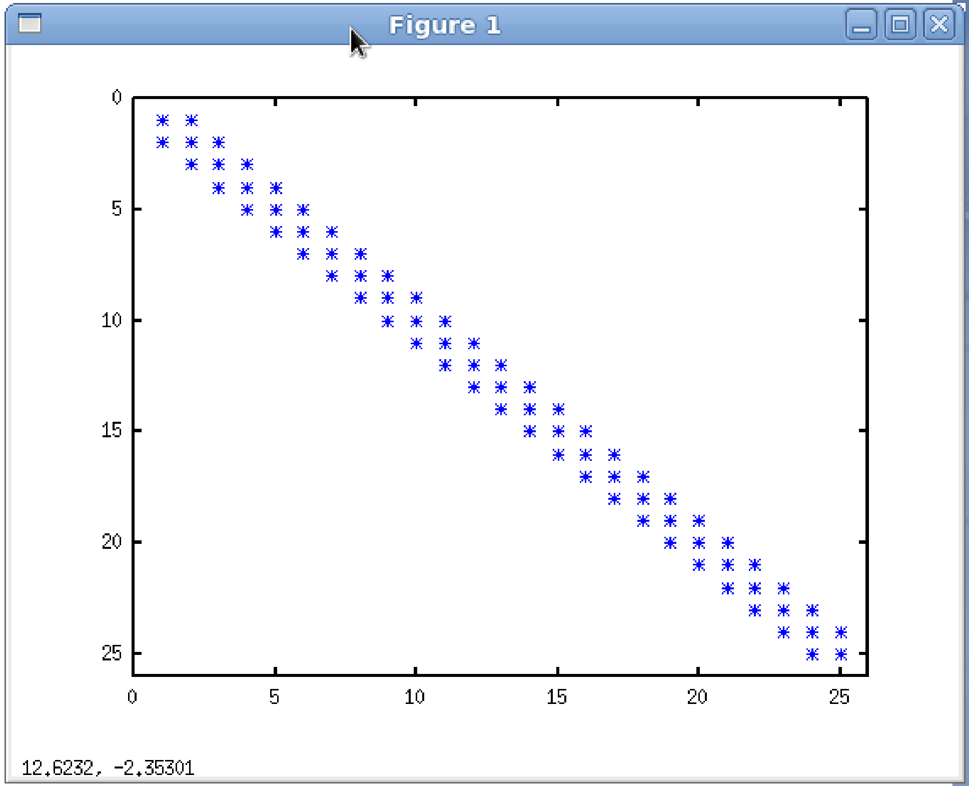
1D Laplacian operator and Matlab spy() function

```
% Create tridiagonal Laplacian
n = 25;
v = ones(n,1);
A = spdiags([v, -2*v, v], [-1, 0, 1], n, n);

spy(A)
```

Visualizing the 1D Laplacian operator

$$\begin{bmatrix} -2 & 1 & 0 & 0 & 0 & \cdots \\ 1 & -2 & 1 & 0 & 0 & \cdots \\ 0 & 1 & -2 & 1 & 0 & \cdots \\ 0 & 0 & 1 & -2 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$



Direct solvers can be problematic for sparse

- Generally, when you are dealing with a sparse matrix, it is very large. You are killed by the O(N^3) scaling of Gauss elimination.
- For some matrices, Gauss elimination can decrease sparsity (“fill-ins”).
- Fill-ins and removals take time in a list-like data structure.

Totally different solver: Jacobi method

- Derivation on blackboard

$$Ax=b$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{11} & a_{12} & a_{13} \\ a_{11} & a_{12} & a_{13} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

Consider 3x3 system

- Want to solve for x, y, z.

$$\begin{aligned} 5x+6y+7z&=2\\ -3x+8y-2z&=3\\ -2x+2y-10z&=4 \end{aligned}$$

- Rearrange to isolate x, y, z on LHS.

$$\begin{aligned} 5x&=2-6y-7z\\ 8y&=3+3x+2z\\ 10z&=-4-2x+2y \end{aligned}$$

- Solve for x, y, z

$$\begin{aligned} x&=(2-6y-7z)/5\\ y&=(3+3x+2z)/8\\ z&=(-4-2x+2y)/10 \end{aligned}$$

Done, right?

- Maybe not...

- What about x, y, z, on RHS?

$$x=(2-6y-7z)/5$$
$$y=(3+3x+2z)/8$$
$$z=(-4-2x+2y)/10$$

- What if we stick random values in for x, y, z and then iterate?

$$x_1=(2-6y_0-7z_0)/5$$
$$y_1=(3+3x_0+2z_0)/8$$
$$z_1=(-4-2x_0+2y_0)/10$$

Try it!

- Iteration code: jacobi.m
- Test harness: test_jacobi_heat
- Seems like magic!
  - But convergence is slow.

```
n = 7;
v = ones(n,1);
A = spdiags([v, -2*v, v], [-1, 0, 1], n, n);

% Make single heat source at middle of bar
b = zeros(1, n);
spike = floor(n/2);
b(spike) = 5;

% Now iterate
x = jacobi(A, b', 300);
```

Jacobi iterations solving Ax = b

$x_{true} = A \backslash b$

$x^{(0)}_{jacobi}$ $x^{(1)}_{jacobi}$ $x^{(2)}_{jacobi}$

```
x_true =     2.5000    5.0000    7.5000   10.0000    7.5000    5.0000    2.5000
x_jacobi =   0.0000    0.0000    0.0000   -5.0000    0.0000    0.0000    0.0000
-----------------------------------------------
iteration 1
x_true =     2.5000    5.0000    7.5000   10.0000    7.5000    5.0000    2.5000
x_jacobi =  -0.0000   -0.0000   -2.5000    2.5000   -2.5000   -0.0000   -0.0000
-----------------------------------------------
iteration 2
x_true =     2.5000    5.0000    7.5000   10.0000    7.5000    5.0000    2.5000
x_jacobi =  -0.0000   -1.2500    1.2500    0.0000    1.2500   -1.2500   -0.0000
-----------------------------------------------
iteration 3
x_true =     2.5000    5.0000    7.5000   10.0000    7.5000    5.0000    2.5000
x_jacobi =  -0.6250    0.6250   -0.6250    3.7500   -0.6250    0.6250   -0.6250
-----------------------------------------------
iteration 4
x_true =     2.5000    5.0000    7.5000   10.0000    7.5000    5.0000    2.5000
x_jacobi =   0.3125   -0.6250    2.1875    1.8750    2.1875   -0.6250    0.3125
```

Does Jacobi work on any random matrix?

```
>> jacobi(A, b, tol)
x_true =     0.2121  -0.4985  -0.9994   0.0927   1.2020  -0.7253   1.0983
x_jacobi =   0.5430   1.2574  -1.0717   0.0206   0.5124  -0.1673  -0.9431
------------------------------------------------
iteration 1
x_true =     0.2121  -0.4985  -0.9994   0.0927   1.2020  -0.7253   1.0983
x_jacobi =   0.4773  23.0403  -0.8990  -1.3236  -2.5008  -0.6296   9.6929
------------------------------------------------
iteration 2
x_true =     0.2121  -0.4985  -0.9994   0.0927   1.2020  -0.7253   1.0983
x_jacobi =  65.5693 -33.2449 -15.8157  38.4691  -1.2810 -12.4624  74.8363
------------------------------------------------
iteration 3
x_true =     0.2121  -0.4985  -0.9994   0.0927   1.2020  -0.7253   1.0983
x_jacobi = -46.4463 -1038.5696  75.3368   4.5241  90.7926 -77.7423 191.9649
------------------------------------------------
iteration 4
x_true =     0.2121  -0.4985  -0.9994   0.0927   1.2020  -0.7253   1.0983
x_jacobi = -1917.1752 -2956.7999 384.5904 -787.8674 978.1561 311.4677 -3317.6243
------------------------------------------------
iteration 5
x_true =     0.2121  -0.4985  -0.9994   0.0927   1.2020  -0.7253   1.0983
x_jacobi = -9037.1542 30479.2495 356.4004 -6033.4404 -2113.5948 4004.8109 -
18295.9981
```

- Solve for vector $[x_1^{(0)}, x_2^{(0)}, x_3^{(0)}]$

$$\begin{aligned}
x_1 &= (b_1 - a_{12} x_2 - a_{13} x_3)/a_{11} \\
x_2 &= (b_2 - a_{21} x_1 - a_{23} x_3)/a_{22} \\
x_3 &= (b_3 - a_{31} x_1 - a_{32} x_2)/a_{33}
\end{aligned}$$

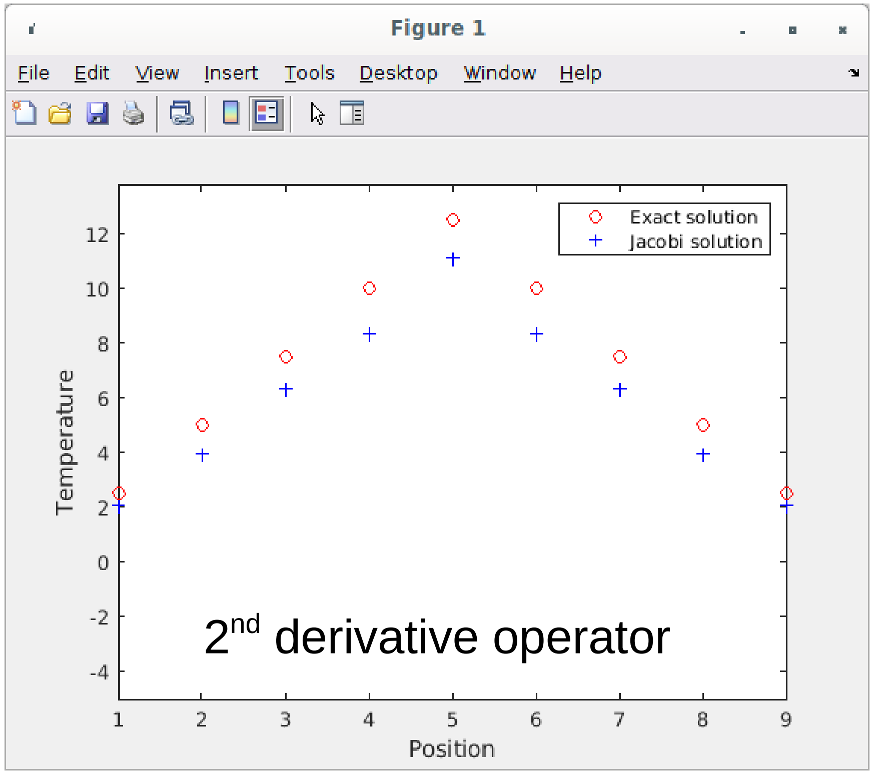
- Choose initial guess $[x_1^{(0)}, x_2^{(0)}, x_3^{(0)}] = [b_{1,} b_{2,} b_3]$
- Then iterate

$$\begin{aligned}
x_1^{(n+1)} &= (b_1 - a_{12} x_2^{(n)} - a_{13} x_3^{(n)})/a_{11} \\
x_2^{(n+1)} &= (b_2 - a_{21} x_1^{(n)} - a_{23} x_3^{(n)})/a_{22} \\
x_3^{(n+1)} &= (b_3 - a_{31} x_1^{(n)} - a_{32} x_2^{(n)})/a_{33}
\end{aligned}$$

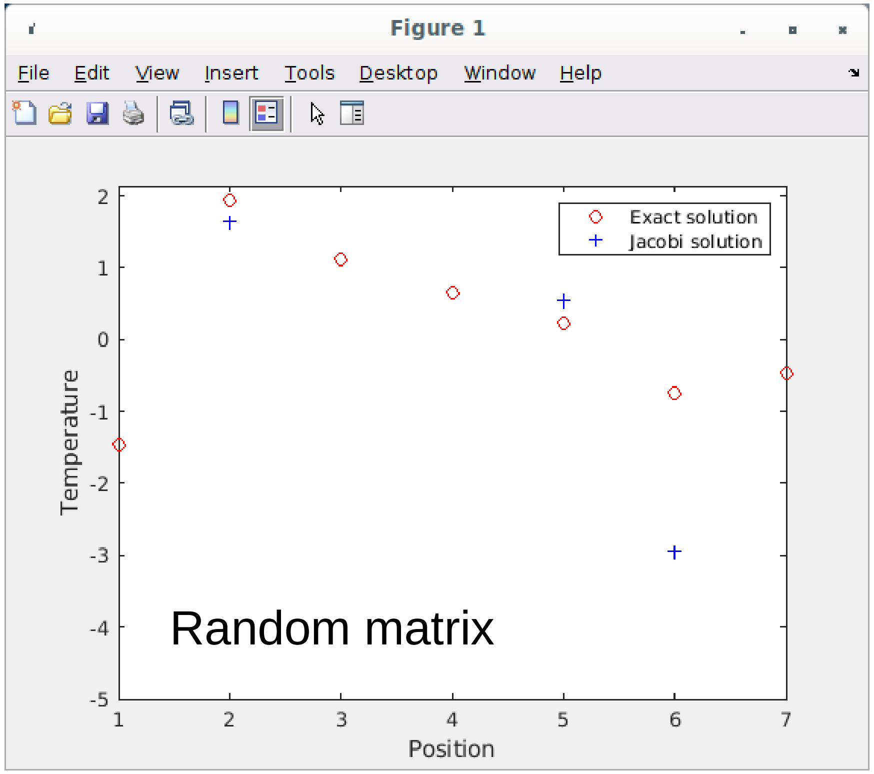
- This is called Jacobi's method

Jacobi iteration

- Works for some matrices



- Fails for other matrices



What's going on???

Jacobi Method – Matrix Derivation

Start with

$$Ax=b$$

Separate diagonal from non-diagonal elements

$$(D+N)x=b$$

Move *Nx* to rhs, and multiply through by *D*$^{-1}$. Note taking inverse of diagonal matrix is trivial.

$$x=D^{-1}(b-Nx)$$

Iterate

$$x_{n+1}=D^{-1}(b-Nx_n)$$

Jacobi method -- algorithm

1. Decompose *A* into *N* and *D*.
2. Compute D^{-1}.
3. Select initial guess: $x_0 = b$ (or any value)
4. For loop:
5. Compute next *x* value: $x_{n+1} = D^{-1}(b - Nx_n)$
6. If $norm(x_{n+1} - x_n) < tol$, return x_{n+1}
7. Else loop again.

Why does Jacobi work?

- We must show two things:
    - The iteration converges, i.e. $x_{n+1} = x_n$
    - The iteration converges to the correct solution.
- The second is easy. The iteration expresses an identity:

$$Ax = b \quad \rightarrow \quad x = D^{-1}(b - Nx) \quad \rightarrow \quad x_{n+1} = D^{-1}(b - Nx_n)$$

so if $x_{n+1} = x_n$, then x_n must be the solution.

- Proving convergence is more involved....

Consider Jacobi iteration

$$x_1 = D^{-1}(b - Nx_0)$$

$$x_2 = D^{-1}(b - ND^{-1}(b - Nx_0))$$

$$x_3 = D^{-1}(b - ND^{-1}(b - ND^{-1}(b - Nx_0)))$$

$$= D^{-1}b - D^{-1}ND^{-1}b - D^{-1}ND^{-1}ND^{-1}b - D^{-1}ND^{-1}ND^{-1}Nx_0$$

Notice powers of $D^{-1}N$

Recognize there are two types of terms:

$$term0 = (D^{-1}N)(D^{-1}N)\cdots(D^{-1}N)x_0$$ Multiplies x_0

$$term1 = \ldots - D^{-1}b - D^{-1}ND^{-1}b - D^{-1}ND^{-1}ND^{-1}b$$ Multiplies b

Consider matrix powers (square matrix)

What is $A^n, n \to \infty$?

A can be decomposed

$$A^n = (U S U^{-1})(U S U^{-1})(U S U^{-1})\ldots$$

where U is unitary and S is diagonal (eigenvalues on diagonal). Next note $U^{-1}*U$ is identity, so

$$A^n = U(S S S \ldots S)U^{-1} = U S^n U^{-1}$$

Behavior of A^n depends upon its eigenvalues.

$|\lambda_{max}| < 1, A^n \to 0, n \to \infty$ $|\lambda_{min}| > 1, A^n \to \infty, n \to \infty$

Term0 should decay to zero

$$term0 = (D^{-1}N)(D^{-1}N)\cdots(D^{-1}N)x_0$$

$$= (D^{-1}N)^n x_0 \rightarrow 0 \quad \text{for} \quad n \rightarrow \infty$$

For this to go to zero, max eigenvalue must satisfy

$$|\lambda_{max}| < 1$$

Also, note that convergence speed depends upon eigenvalue. The closer the eigenvalue is to 1, the slower the convergence.

Term1 must tend to a finite value

$$term1 = D^{-1}b - D^{-1}ND^{-1}b - D^{-1}ND^{-1}ND^{-1}b\ldots$$

$$= D^{-1}b - (D^{-1}N)D^{-1}b - (D^{-1}N)(D^{-1}N)D^{-1}b\ldots$$

$$= D^{-1}b - (D^{-1}N)D^{-1}b - (D^{-1}N)^2 D^{-1}b\ldots$$

Again, this requires

$(D^{-1}N)^n x_0 \rightarrow 0$ for $n \rightarrow \infty$

For this to go to zero, max eigenvalue must satisfy

$$|\lambda_{max}| < 1$$

What are the eigenvalues of $D^{-1}N$?

$$D^{-1}=\begin{bmatrix} -1/2 & 0 & 0 & 0 & 0 & \cdots \\ 0 & -1/2 & 0 & 0 & 0 & \cdots \\ 0 & 0 & -1/2 & 0 & 0 & \cdots \\ 0 & 0 & 0 & -1/2 & 0 & \cdots \\ 0 & 0 & 0 & 0 & -1/2 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \qquad N=\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & \cdots \\ 1 & 0 & 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & 0 & 1 & \cdots \\ 0 & 0 & 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

So:

$$(D^{-1}N)=\begin{bmatrix} 0 & -1/2 & 0 & 0 & 0 & \cdots \\ -1/2 & 0 & -1/2 & 0 & 0 & \cdots \\ 0 & -1/2 & 0 & -1/2 & 0 & \cdots \\ 0 & 0 & -1/2 & 0 & -1/2 & \cdots \\ 0 & 0 & 0 & -1/2 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

This is a Toeplitz matrix

Eigenvalues of D^{-1}N

$$(D^{-1}N) = \begin{bmatrix} 0 & -1/2 & 0 & 0 & 0 & \cdots \\ -1/2 & 0 & -1/2 & 0 & 0 & \cdots \\ 0 & -1/2 & 0 & -1/2 & 0 & \cdots \\ 0 & 0 & -1/2 & 0 & -1/2 & \cdots \\ 0 & 0 & 0 & -1/2 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$\lambda = -\cos\left(\frac{s\pi}{N+1}\right) \qquad s = 1 \cdots N$$

Clearly, $|\lambda_{max}| < 1$ so Jacobi iteration will work for the original matrix. $\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix}$

However, max eigenvalue gets close to 1. And as N grows, the max eigenvalue asymptotes to 1.

Beginnings of proof: Eigenvalues of D^{-1}N are cosines

Eigenvalue equation

$$-\frac{1}{2}\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \lambda \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix}$$

Implies recurrence relations:

$$u_2 = \lambda u_1$$

$$u_1 + u_3 = \lambda u_2$$

$$u_{n-1} + u_{n+1} = \lambda u_n$$

$$u_{N-1} = \lambda u_N$$

Outline of proof

- Make guess for eigenvector

$$u_s = A\sin(s\theta) + B\cos(s\theta)$$

- Substitute into recurrence equations to get

$$\lambda u_s = (2\cos\theta) u_s$$

- Use boundary conditions (recurrence equations at s = 1 and s = N) to find θ

$$\theta_s = \frac{s\pi}{N+1}$$

- Finishing the proof is your homework....

That's great, but.....

- What if you don't have the eigenvalues of your matrix?
- Require diagonally dominant matrix
  - For each row j, require $\sum_{i\neq j} |a_{ij}| < |a_{jj}|$

$$\begin{bmatrix} 4.5 & 1.1 & -0.6 \\ 3.2 & -7.3 & 1.1 \\ -3.2 & 2.2 & 6.7 \end{bmatrix}$$

Diagonally dominant

$$\begin{bmatrix} -3.5 & 4.1 & -0.6 \\ 4.7 & -2.3 & 1.1 \\ -3.2 & 4.3 & 8.7 \end{bmatrix}$$

Not diagonally dominant

- This works because of the Gershgorin circle theorem

Remarks on Jacobi's Method

- Works as long as $|\lambda_{max}| < 1$ for iteration matrix $(D^{-1}N)$
- Not all matrices satisfy this criterion, but many important ones do.
- Convergence is slow.
  - "High frequency" components converge faster.
  - "Low frequency" components converge more slowly.

What's so great about iteration?

- Consider Gaussian elimination:
  - Choose pivot, iterate over rows below.
  - For each row, iterate over elements in columns and subtract.
  - Do this for each pivot
- This algorithm is $O(N^3)$

$$\begin{bmatrix} 10 & 4 & -3 \\ 5 & -2 & 1 \\ -8 & 2 & -3 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 4 & -3 \\ 0 & 8 & -5 \\ 0 & \frac{52}{8} & -\frac{54}{8} \end{bmatrix} \rightarrow \begin{bmatrix} 10 & 4 & -3 \\ 0 & 8 & -5 \\ 0 & 0 & \frac{43}{13} \end{bmatrix}$$

What’s so great about iteration?

- Sparse matrices show up frequently in engineering problems.
- Sparse matrices typically have large (NxN), but far fewer elements.
- An iterative method typically has time complexity O(N*m*k)
  - N = number of rows
  - m = number of non-zeros in each col.
  - k = number of iterations
- Number of iterations is related to accuracy. You can trade accuracy off against computation time.

Next topic: Gauss-Seidel iteration

Jacobi iteration:

$$x_1^{(n+1)} = (b_1 - a_{12} x_2^{(n)} - a_{13} x_3^{(n)}) / a_{11}$$
$$x_2^{(n+1)} = (b_2 - a_{21} x_1^{(n)} - a_{23} x_3^{(n)}) / a_{22}$$
$$x_3^{(n+1)} = (b_3 - a_{31} x_1^{(n)} - a_{32} x_2^{(n)}) / a_{33}$$

Jacobi: Collect all new x values before using them.

Gauss-Seidel iteration:

$$x_1^{(n+1)} = (b_1 - a_{12} x_2^{(n)} - a_{13} x_3^{(n)}) / a_{11}$$
$$x_2^{(n+1)} = (b_2 - a_{21} x_1^{(n+1)} - a_{23} x_3^{(n)}) / a_{22}$$
$$x_3^{(n+1)} = (b_3 - a_{31} x_1^{(n+1)} - a_{32} x_2^{(n+1)}) / a_{33}$$

Gauss-Seidel: Use each new x value as soon as you have it.

What's so great about iteration?

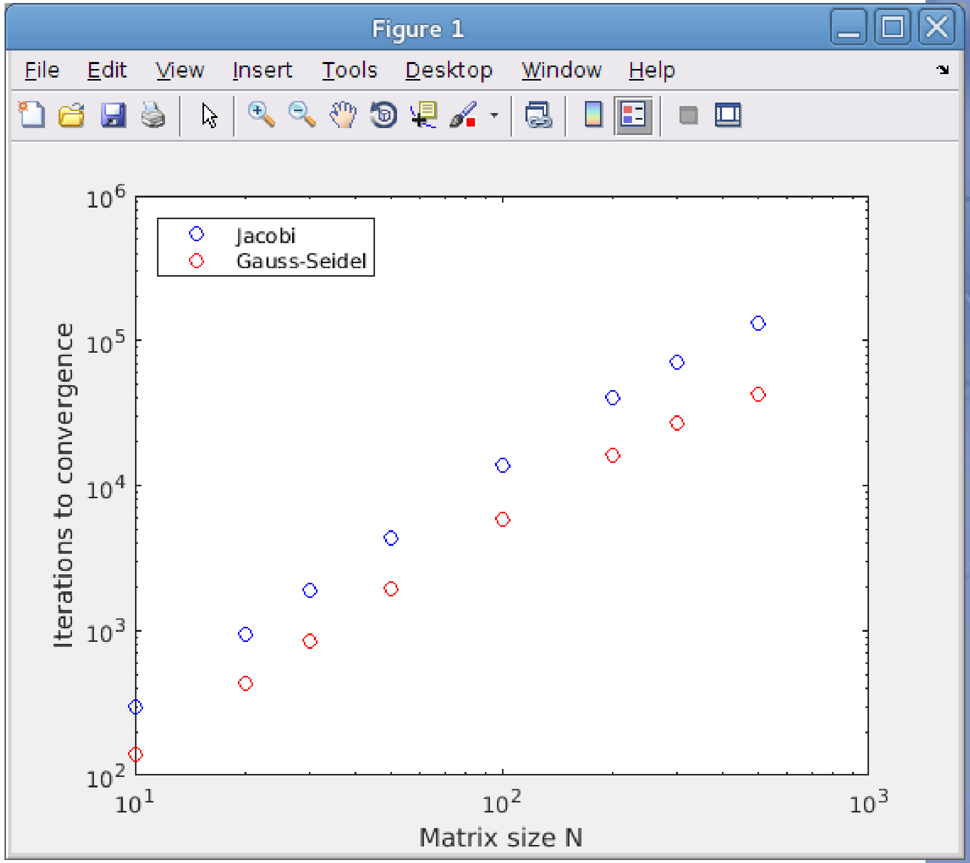
- For Jacobi and Gauss-Seidel iteration:
  - There is an overall loop.
  - For each outer loop, iterate over rows
  - For each row, do operation on k non-zero entries. For sparse, K is usually a small constant, and doesn't grow with N.
- This is O(N*k*something), where “something” depends on the outer loop. What is “something”?
- “Something” depends upon the accuracy you demand. If you use a good algorithm, it can be a small constant number.

Gauss-Seidel

- Convergence is twice as fast as Jacobi iteration
- Same convergence criteria apply:

$$|\lambda_{max}| < 1$$

- Demo: HeatCountIterations



Next topic: Advanced solvers -- solve Ax = b as an optimization

- Consider f(x) and SPD matrix A:

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

- f(x) is minimized by the x which makes the gradient = 0 (true for symmetric positive-definite A).

$$\nabla f(x)=A\cdot x-b=0$$

- Therefore, solving the linear equation Ax = b for x is equivalent to extremizing f(x).

Minimizing for SPD A

Solve for x $\Leftrightarrow$ Find x minimizing

$$A\cdot x-b=0 \quad \Leftrightarrow \quad f(x)=\frac{1}{2}x^T\cdot A\cdot x-x^T\cdot b$$

Quadratic form displaced from origin

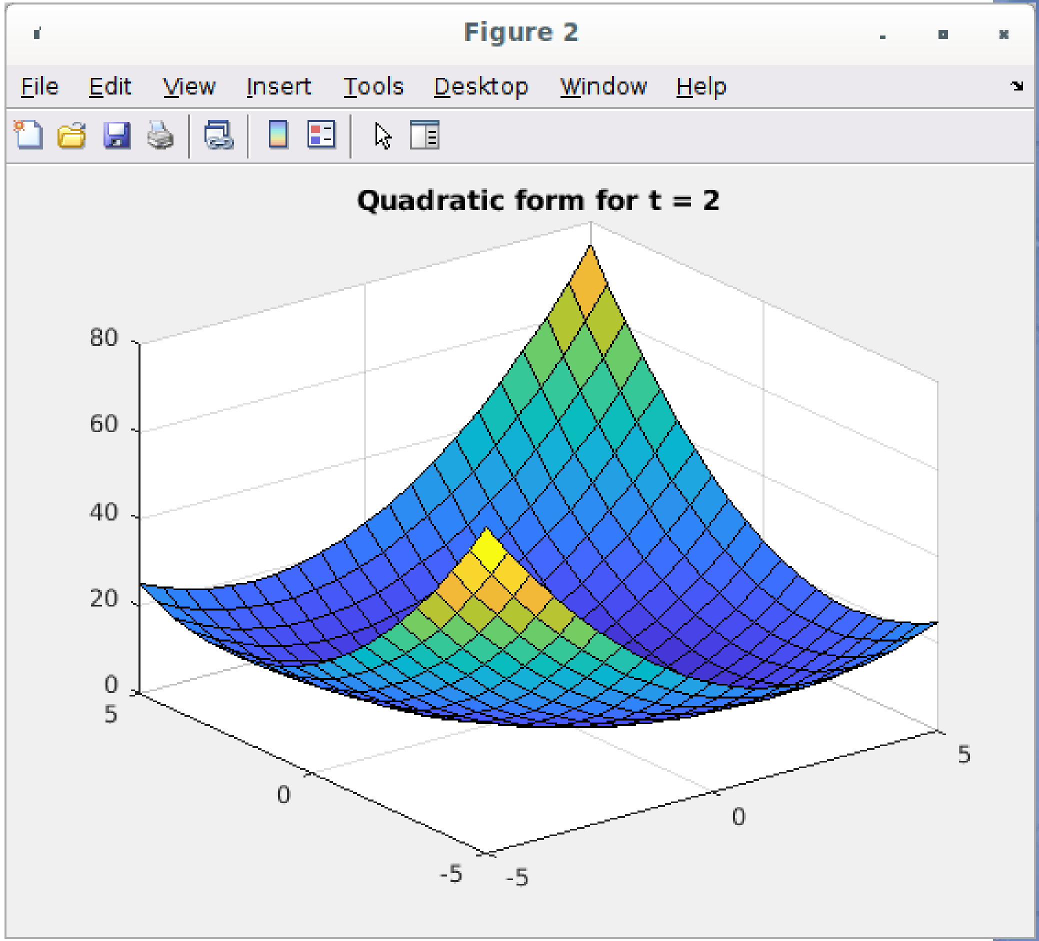
- Quadratic form

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

- Gradient

$$\nabla f(x) = A \cdot x - b = 0$$

- Gradient is linear system we want to solve.



When can f(x) be minimized?

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

- Matrix A must be positive definite for f(x) be upward-facing parabola
  - $x^T A x$ is parabola (quadratic form)
  - $x^T b$ term simply shifts the parabola's bottom point.
- For many of the coming algorithms, matrix A should be symmetric (or Hermitian) so that

$$\nabla f(x)=A\cdot x-b=0$$

These conditions mean A should be symmetric positive definite (SPD).

Minimization via iteration

- Draw picture on blackboard
- Use iteration to find minimum of f(x).
  - Alpha is (scalar) step length
  - r is direction vector

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

- What direction to use? A simple answer:

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

- Take step in same direction as gradient. ***Method of steepest descent (gradient descent).***

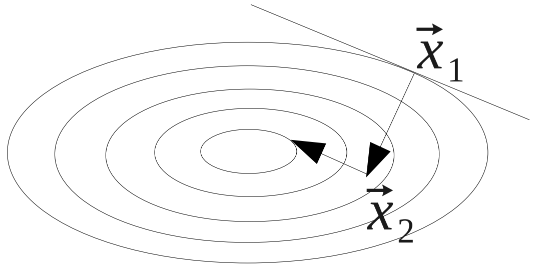
Method of steepest descent

Solve for x		Find x minimizing
$A \cdot x - b = 0$	$\Leftrightarrow$	$f(x) = \frac{1}{2} x^T \cdot A \cdot x - x^T \cdot b$

- Also called “method of gradient descent”

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

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



- Needs gradient at x

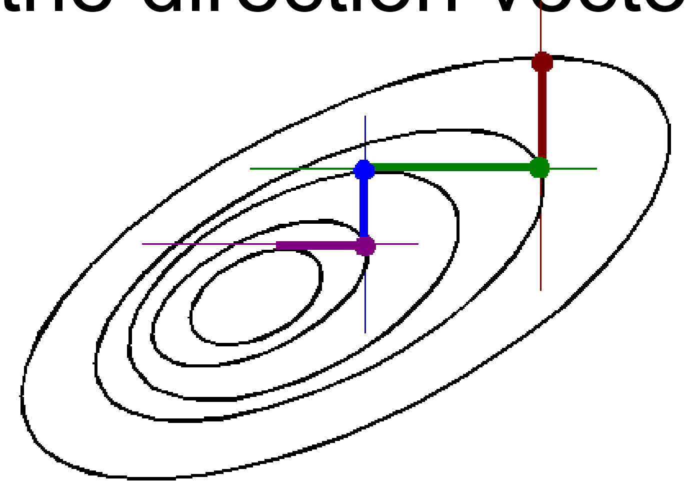
$$f(x) = \frac{1}{2} x^T \cdot A \cdot x - x^T \cdot b$$

$$\nabla f(\vec{x}) = A \cdot \vec{x} - \vec{b} = \vec{r}$$ Gradient = residual!

- What to use for alpha?

How to get alpha?

- Choose alpha which minimizes $f(x_{n+1})$ along the line defined by direction vector. That is, we step to the deepest point along the line defined by the direction vector r.



- Derivation on blackboard (and next slide).
- Result: $$\alpha_n = \frac{\vec{r}_n^T \cdot \vec{r}_n}{\vec{r}_n^T \cdot A \cdot \vec{r}_n}$$

To get alpha

- Residual at point n+1

$$\vec{r}_{n+1}=A\vec{x}_{n+1}-\vec{b}$$

$$\vec{r}_{n+1}=A(\vec{x}_n-\alpha_n\vec{r}_n)-\vec{b}$$

$$=\vec{r}_n-\alpha_n A\vec{r}_n$$

- Invoke orthogonality $\vec{r}_n^T\cdot\vec{r}_{n+1}=0$

This is because the next step direction *n+1* is orthogonal to the last one *n*. The last step took us to the minumum of *f(x)* along the direction $\vec{r}_n$.

$$\vec{r}_n^T\cdot\vec{r}_{n+1}=0$$

$$\vec{r}_n^T\cdot(\vec{r}_n-\alpha A\cdot\vec{r}_n)=0$$

$$\vec{r}_n^T\cdot\vec{r}_n=(\alpha_n\vec{r}_n^T\cdot A)\cdot\vec{r}_n$$

- So we get the desired result: $\alpha_n=\dfrac{\vec{r}_n^T\cdot\vec{r}_n}{\vec{r}_n^T\cdot A\cdot\vec{r}_n}$

Gradient descent algorithm

1. Start with $\vec{x}_0=\vec{b}$
2. $\vec{r}_n=A\cdot\vec{x}_n-\vec{b}$
3. $\alpha_n=\dfrac{\vec{r}_n^{\,T}\cdot\vec{r}_n}{\vec{r}_n^{\,T}\cdot A\cdot\vec{r}_n}$
4. $\vec{x}_{n+1}=\vec{x}_n-\alpha_n\vec{r}_n$
5. Check for convergence:

   Return if converged

   Else loop back to 2

Solve

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

Consider

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

Iterate for x

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

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

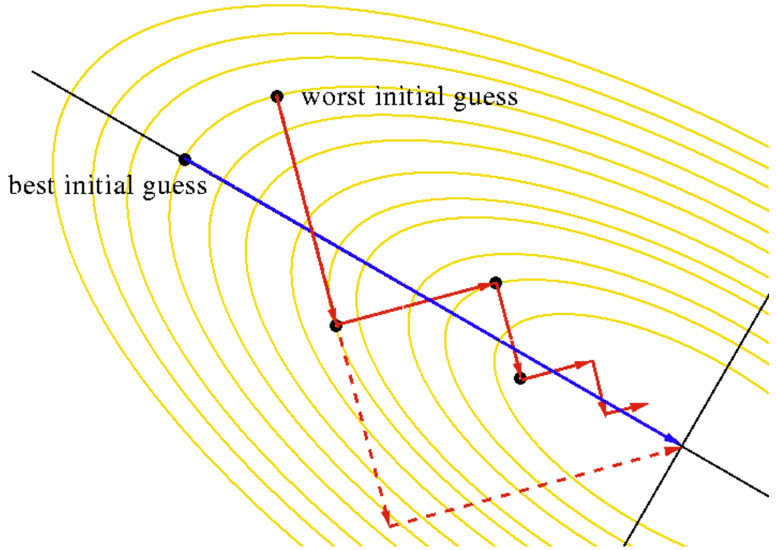
How to know when to stop?

$$\vec{r}_n = -\nabla f(\vec{x}_n)$$
$$\vec{x}_{n+1} = \vec{x}_n + \alpha_n \vec{r}_n$$

- Bad: $\|\vec{x}_{n+1} - \vec{x}_n\| < tol$ Norm of step
  - Bad because gradient descent often takes tiny steps even far from the solution point.
- Less bad: $\|\nabla f(\vec{x}_n)\| < \epsilon$ Norm of slope (gradient)
  - Requires knowing about the scale of f(x)
- Another possibility: $\|\nabla f(\vec{x}_n)\| < \epsilon |f(\vec{x}_n)|$ Relative norm of slope
  - Problematic if f(optimum) = 0
- Best: $\|\nabla f(\vec{x}_n)\| < \epsilon (1 + |f(\vec{x}_n)|)$

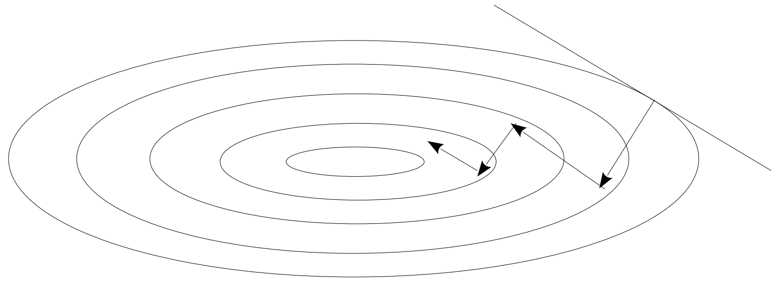
Observations

- If you start on an axis of the ellipse, convergence is fast.
- If the ellipse is close to spherical, convergence is fast.

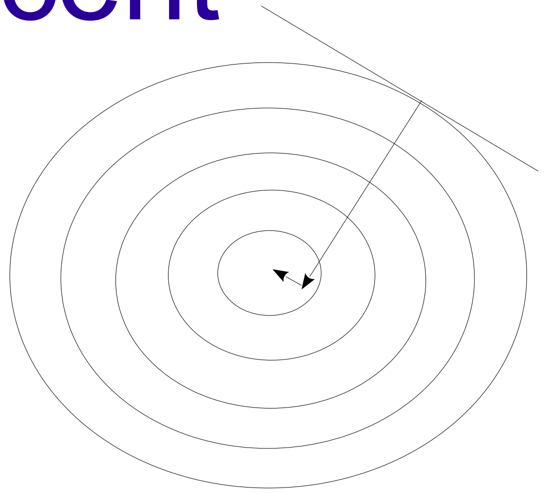


- Conversely, if the ellipse is very long and narrow, you will likely zig-zag with slow convergence. This is more likely.
  - This effect is characterized by the condition number of the matrix.

Matrix condition number and gradient descent



High K



Low K

- Zig-zag walk.
- Slow convergence.

- GD shoots almost to center of quadratic form in 1 step.
- Fast convergence.

Review of session topics

It's all about solving $Ax=b$ for sparse

- Jacobi iteration.
  - Iterative methods are good for sparse.
  - Slow convergence.
- Gauss-Seidel.
- Solving Ax = b as a minimization problem.
- Steepest descent (a.k.a gradient descent).