Iterative Methods:

Basic idea:

- Convert $A\mathbf{x} = b$ into $\mathbf{x} = C\mathbf{x} + d$
- Make an initial guess at the solution. This guess is $\mathbf{x}_0$.
- Apply $\mathbf{x}_{i+1} = C\mathbf{x}_i + d$ until solution is acceptable

Conceptually similar to iterative root finding techniques (e.g. Newton's Method). The difference is that **x** is now a vector of values.

Advantages:

- Speed
- Roundoff errors do not accumulate

Disadvantages:

- Might not work (solution might not converge)

Conversion of equations:

$$\text{Start with } Ax=b: \quad \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} x = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$\text{Divide each equation by } a_{\text{ii}}: \quad \begin{bmatrix} 1 & a_{12}/a_{11} & a_{13}/a_{11} \\ a_{21}/a_{22} & 1 & a_{23}/a_{22} \\ a_{31}/a_{33} & a_{32}/a_{33} & 1 \end{bmatrix} x = \begin{bmatrix} b_1/a_{11} \\ b_2/a_{22} \\ b_3/a_{33} \end{bmatrix}$$

$$\text{Break up LHS}: \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} x + \begin{bmatrix} 0 & a_{12}/a_{11} & a_{13}/a_{11} \\ a_{21}/a_{22} & 0 & a_{23}/a_{22} \\ a_{31}/a_{33} & a_{32}/a_{33} & 0 \end{bmatrix} x = \begin{bmatrix} b_1/a_{11} \\ b_2/a_{22} \\ b_3/a_{33} \end{bmatrix}$$

$$\text{Rearrange}: \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} x = \begin{bmatrix} 0 & -a_{12}/a_{11} & -a_{13}/a_{11} \\ -a_{21}/a_{22} & 0 & -a_{23}/a_{22} \\ -a_{31}/a_{33} & -a_{32}/a_{33} & 0 \end{bmatrix} x + \begin{bmatrix} b_1/a_{11} \\ b_2/a_{22} \\ b_3/a_{33} \end{bmatrix}$$

$$\text{We now have } x = Cx+d: \quad x = \begin{bmatrix} 0 & -a_{12}/a_{11} & -a_{13}/a_{11} \\ -a_{21}/a_{22} & 0 & -a_{23}/a_{22} \\ -a_{31}/a_{33} & -a_{32}/a_{33} & 0 \end{bmatrix} x + \begin{bmatrix} b_1/a_{11} \\ b_2/a_{22} \\ b_3/a_{33} \end{bmatrix}$$

$$\text{Bottom Line}: \quad c_{ii}=0, \quad c_{ij}=-a_{ij}/a_{ii}, \quad d_i=b_i/a_{ii}$$

Jacobi:

```
x = x0;
while true
   xold = x;
   x = C * xold + d;
   if x and xold are close enough, break, end
end
```

Gauss –Seidel:

```
x= x0
while true
   xold = x;
   for i = 1 : n
      x(i) = C(i, :) * x + d(i);
   end
   if x and xold are close enough, break, end
end
```

The difference is that in the case of Gauss-Seidel the new x(0) is used in calculating the new x(1), the new x(1) and x(2) are used in calculating the new x(3), and so on.

Jacobi:

Better suited to parallel computing

Sufficient condition for convergence: original *A diagonally dominant **

Necessary and sufficient condition: magnitude of largest eigenvalue of *C* < 1

Gauss –Seidel:

Usually preferred

Converges faster

Sufficient condition for convergence: original *A diagonally dominant **

* A matrix if diagonally dominant if $$\left|a_{ii}\right| > \sum_{\substack{j=1:n \\ j\sim=i}} \left|a_{ij}\right|$$

```
[A b] = 0.0819   -0.0286         0         0    1.0667
       -0.0286    0.0668   -0.0200         0    0.7273
             0   -0.0200    0.0736   -0.0250    0.8571
             0         0   -0.0250    0.0639    2.1111
```

Note that *A* is diagonally dominant.

x by left division: `24.2593    32.2101    36.4865    47.3208`

Gauss-Seidel results:

```
  1   13.023256   16.469098   16.127522   39.354248
  2   18.768290   23.760043   31.482329   45.362650
  3   21.311643   29.449105   35.070560   46.766741
  4   23.296199   31.373600   36.070842   47.158156
  5   23.967535   31.960637   36.363430   47.272646
  6   24.172315   32.135949   36.449992   47.306519
  7   24.233470   32.188059   36.475668   47.316566
  8   24.251648   32.203532   36.483288   47.319548
  9   24.257046   32.208126   36.485550   47.320433
 10   24.258648   32.209489   36.486222   47.320695
 11   24.259124   32.209894   36.486421   47.320773
 12   24.259265   32.210014   36.486480   47.320797
 13   24.259307   32.210050   36.486498   47.320803
```

Gauss-Seidel Code:

```
function [ x, flag, k ] = GaussSeidel( A, b, x0, tol, maxk, disp)
%GAUSSSEIDEL solves Ax = b using Gauss-Seidel iteration
% Inputs: A = n x n coefficient matrix
%        b = n x 1 right hand side
%        x0 = n x 1 initial guess
%        tol = tolerance, iteration ends when the norm of the change in
%              x becomes less than tol
%        maxk = maximum iterations
%        disp = display option 0 (default) = no display
% Outputs: x = n x 1 solution
%         flag = set to 1 on success, 0 on failure (max iterations)
%         k = iterations

if nargin < 6, disp = 0; end

[n, nn] = size(A); [m, mm] = size(b);
if n ~= nn || n ~= m || mm ~= 1, error ('bad dimensions'), end
```

The process of creating *C* and *d* fails if any of the diagonal elements of *A* are zero.

This problem could be addressed by employing row pivotting.

```
% generate the iteration matrices C and d
C = zeros(size(A)); d = zeros(size(b));
for i = 1 : n
   if A(i, i) == 0, error ('zero diagonal element'), end
   C(i, :) = -A(i, :) / A(i, i);  C(i, i) = 0;
   d(i) = b(i) / A(i, i);
end

x = x0;  % setup for iterating
```

Iterating ends when the norm of the difference between the old and new vectors is less than the specified tolerance.

$$\text{norm}(v) = \sqrt{{v_1}^2 + {v_2}^2 + {v_3}^2 + \ldots}$$

```
for k = 1: maxk

   xold = x;
   for i = 1 : n
      x(i) = C(i, :) * x + d(i);
   end

   % display code omitted to save space

   if norm(x - xold) < tol
      flag = 1; return;    % solution has converged
   end

end

flag = 0; % no convergance

end
```

Problem: (ch 5; Curve Fitting) A physics lab has produced the experimental results shown below. Theory predicts that $y = mx+b$ (i.e. that x and y are linearly related). What are m and b?

x	y
1.0000	5.4704
2.0000	7.0865
3.0000	8.3603
4.0000	10.1030
5.0000	11.2308
6.0000	12.7550
7.0000	14.1710
8.0000	15.6777
9.0000	17.7830

One possibility (not recommended) :

- Take a piece of graph paper
- Plot the points
- Draw a line that more or less matches the points
- Determine m by measuring rise and run (m – rise/run)
- Determine b from the y intercept of the line

This process is painful and the results are likely to be inaccurate.

A better way:

Load the data into Matlab:

```
>> load  data1.txt    % creates a matrix called  data1
```

Get the x and y values into separate vectors (for convenience):

```
>> x = data1(:, 1);
>> y = data1(:, 2);
```

Use *polyfit* to find the straight line that best matches the data:

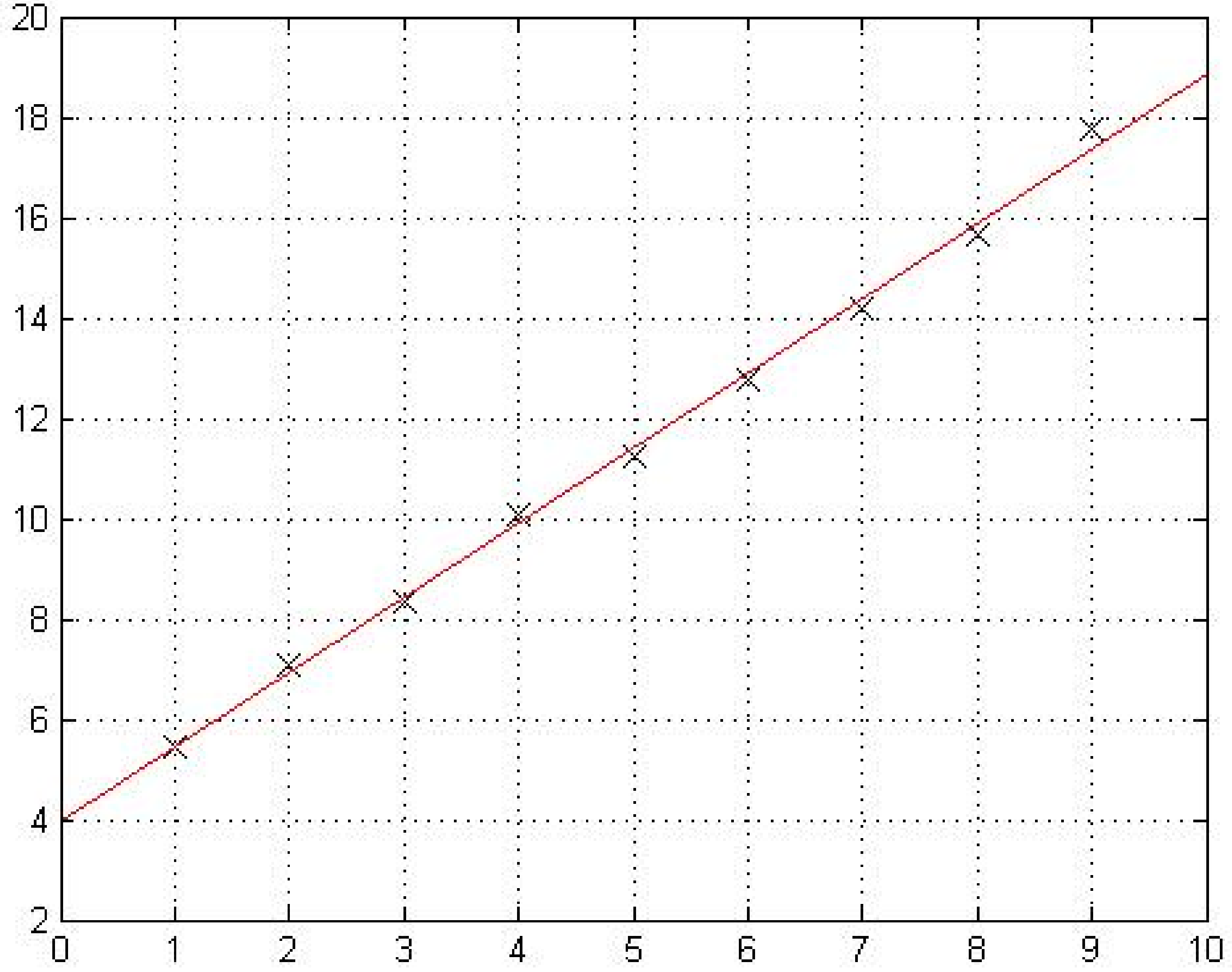
```
>> p = polyfit (x, y, 1);
>> fprintf ('The best fit line is %f * x + %f\n', p(1), p(2));
```

Create a nice plot showing both data points and the straight line:

```
xf = [0 10];    % only two points are required to define a straight line
yf = polyval (p, xf);     % get corresponding y values

plot (x, y, 'xk', xf, yf, 'r', 'MarkerSize', 10);  grid on;
```

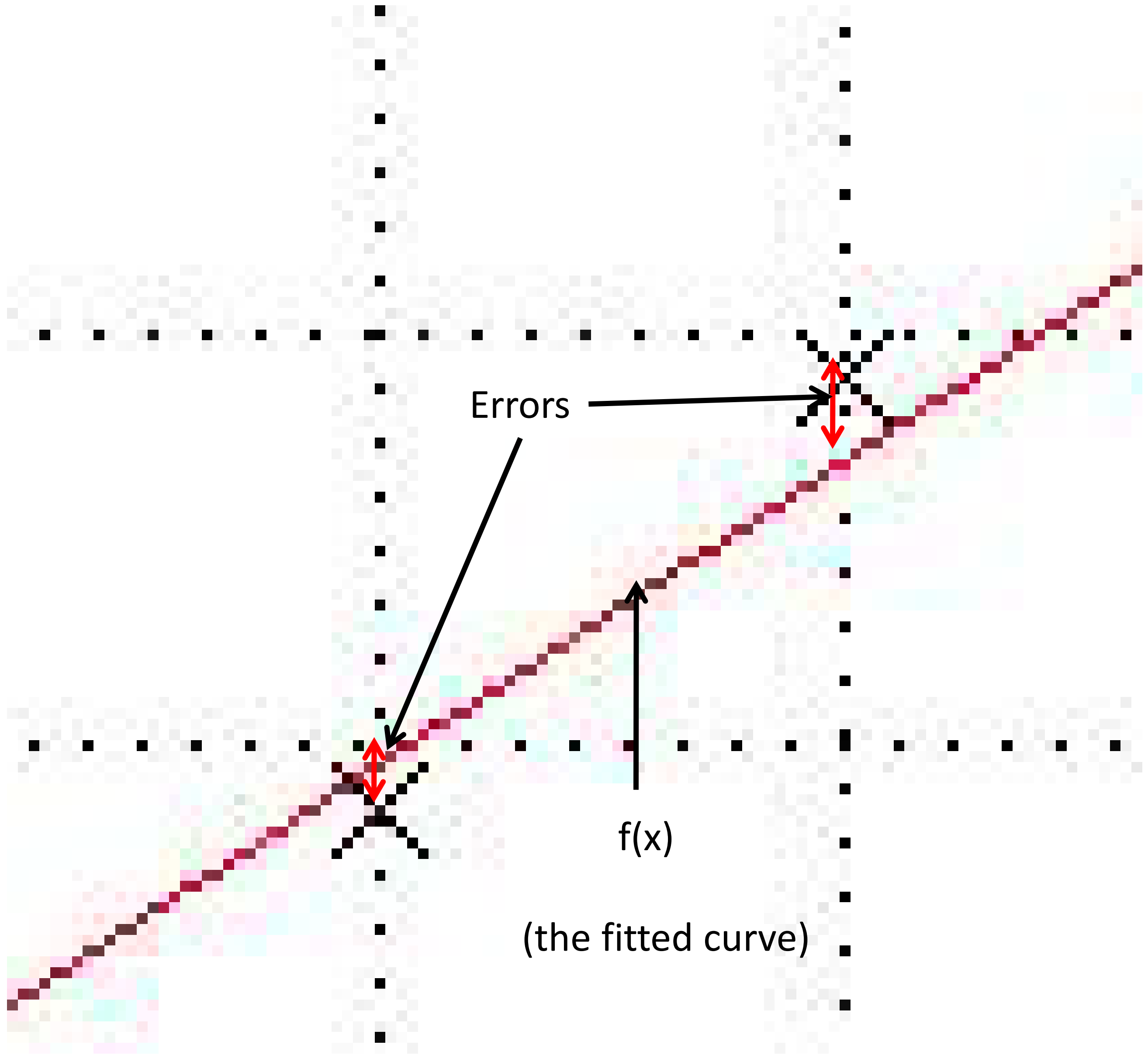
The best fit line is 1.488290 * x + 3.962739



Errors:

The difference between each data point and the fitted curve.

The basic concept applies whether the fitted curve is a straight line (as in this case) or something more exotic.



Possible “Best Fit” Criteria:

Sum of errors:

positive and negative errors cancel

Sum of absolute errors:

best fit is not unique

Minimax (minimize maximum error):

gives outliers undue influence
best fit not unique

Sum of squares of errors:

best fit is unique
generally used

The “best fit” line is the line that minimizes $\sum \left(y_i - f(x_i)\right)^2$

Quality of fit:

Correlation of determination (r^2): $r^2 = \dfrac{S_t - S_r}{S_t}$

S_t is the sum of differences between data points and the average *y*: $S_t = \sum \left(y_i - \bar{y}\right)^2$

S_r is the sum of differences between data points and the fitted line: $S_r = \sum \left(y_i - f(x_i)\right)^2$

If fitted line fits data points perfectly, $S_{r=0}$ and $r^2 = 1$.

If fitted line no better than just using the average *y*, $S_r = S_t$ and $r^2 = 0$.

Correlation coefficient (*r*): $r = \sqrt{r^2}$

Calculation of correlation coefficient:

```
function [r] = correlate( x, y, f )
%CORRELATE calculates the correlation coefficient for given data and function

yBar = mean(y);

St = 0; Sr = 0;
for i = 1:length(x)
   St = St + (y(i) - yBar) ^ 2;
   Sr = Sr + (y(i) - f(x(i))) ^2;
end;

r2 = (St - Sr) / St;
r2 = max([r2 0]); % avoid silly numbers when fit worse than average
r = sqrt(r2);

end
```

Anscombe’s Data:

In all four cases the best fit line is

$y = 0.5x + 3$

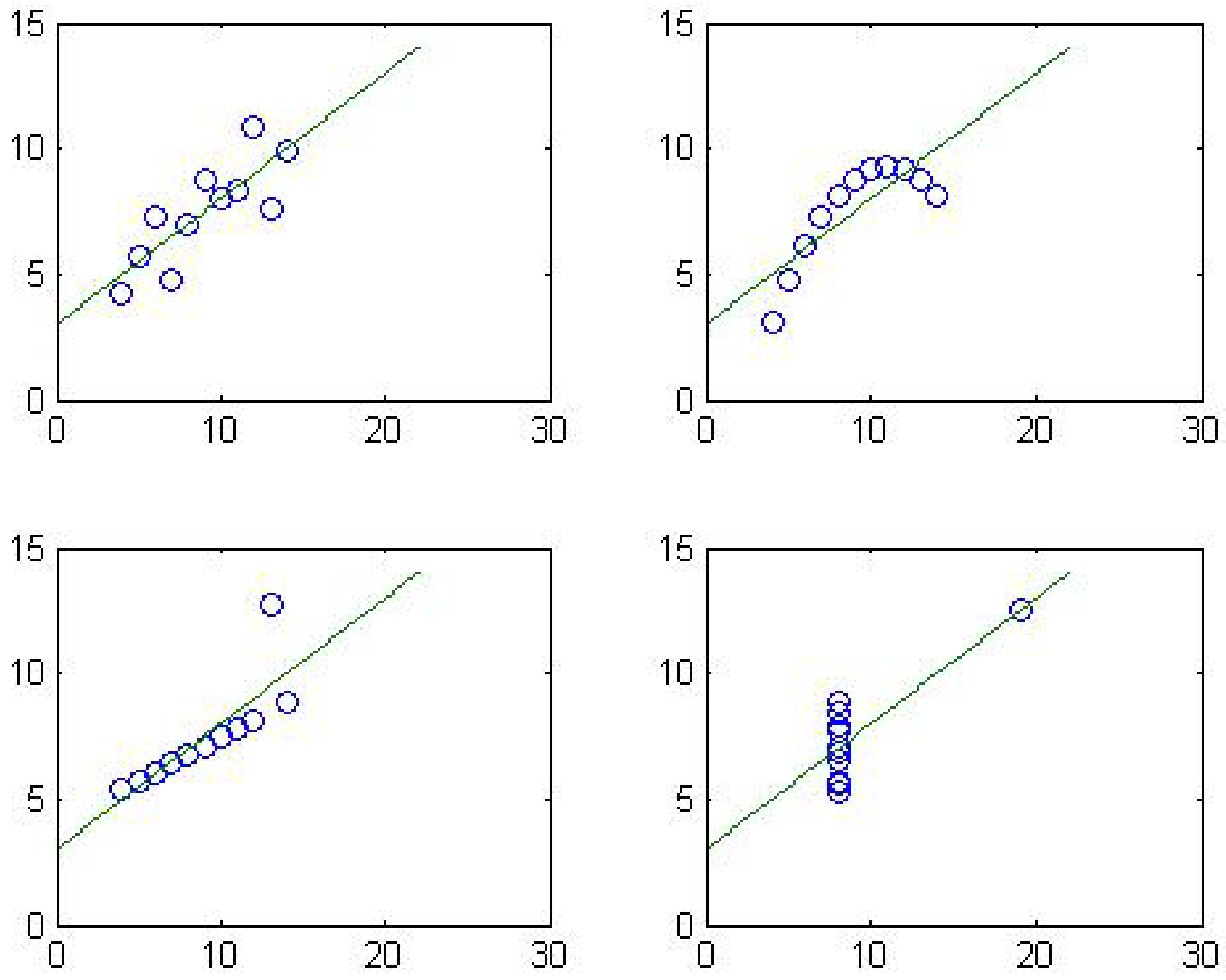
In all four cases

$r^2 = 0.67$

Moral:

- Beware of relying on numbers

- Plots are a valuable tool



15
10
5
0
0
10
20
30