## Unit III

## Curve-Fitting and Interpolation



## Curve-fitting and interpolation

- Curve fitting
  - linear least squares fitting problem
  - transformations to linearize nonlinear problems
  - three solution techniques:
    - normal equations
    - QR decomposition
    - SVD
- Interpolation
  - polynomial interpolation
  - piecewise polynomial interpolation
  - cubic splines



## Curve fitting

- finding an analytical function which approximates a set of data points
- statistics can quantify the relationship between the fit and errors in data points (e.g. from experiment)
- numerical methods is concerned with the calculation of the fitting curve itself



## ***Linear*** curve fitting

- consider a single variable problem initially
- start with m data points $(x_i,y_i)$, $i=1,\ldots,m$
- choose n basis functions $f_1(x),\ldots, f_n(x)$
- define a fit function $F(x) = c_1f_1(x) +\ldots+ c_nf_n(x)$ as a ***linear*** combination of the basis functions
- the problem:
  - find the unknown coefficients $c_1,\ldots, c_n$ so that $F(x_i) \approx y_i$
  - provide a means to quantify the fit



## Linear curve fitting

- in general $m > n$
  - large number of data points
  - (much) smaller number of coefficients/basis functions
- so $F(x_i) = y_i$ exactly is not possible to achieve
- fitting to a line uses two basis functions
  - $f_1(x) = x$
  - $f_2(x) = 1$
- the fitting function is $F(x) = c_1x + c_2$
- m residuals are defined by

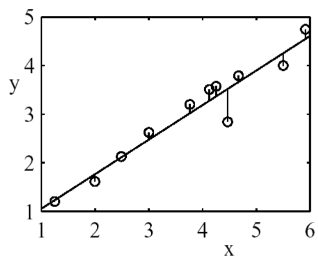
  $r_i = y_i - F(x_i) = y_i - (c_1x_i + c_2)$



## Least squares problem

- quality of fit can be measured by sum of squares of the residuals $\rho = \sum r_i^2 = \sum [y_i - (c_1x_i + c_2)]^2$
  - easy to calculate the fit coefficients
  - agrees with statistical expectations derived from data analysis
- minimizing $\rho$ with respect to $c_1$ and $c_2$ provides the least-squares fit
  - data points $x_i$ and $y_i$ are known in expression for $\rho$
  - only $c_1$ and $c_2$ are unknowns

## Least squares problem






## Geometry or algebra?

- to solve the least squares fitting problem we can ...
  1. minimize the sum of squares (geometry) OR
  2. solve the over-determined system (algebra)
- the equations you get are ...
  – mathematically the same, but
  – have significantly different numerical properties



## Geometric: Minimize the residual

- to minimize $\rho = \rho(c_1, c_2)$ we must have the partial derivatives

$$\frac{\partial \rho}{\partial c_1} = \frac{\partial \rho}{\partial c_2} = 0$$

- differentiating gives (sums are i=1,...,m)

$$\frac{\partial \rho}{\partial c_1} = \sum \frac{\partial}{\partial c_1}[y_i - (c_1 x_i + c_2)]^2 = \sum -2x_i[y_i - (c_1 x_i + c_2)]$$

$$\frac{\partial \rho}{\partial c_2} = \sum \frac{\partial}{\partial c_2}[y_i - (c_1 x_i + c_2)]^2 = \sum -2[y_i - (c_1 x_i + c_2)]$$



## Minimize the residual

- to minimize we get

$$0 = \sum x_i y_i - c_1 \sum x_i^2 - c_2 \sum x_i$$

$$0 = \sum y_i - c_1 \sum x_i - c_2 m$$

- organizing in matrix form gives the *normal equations* for the least squares fit:

$$\begin{bmatrix} \sum x_i^2 & \sum x_i \\ \sum x_i & m \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} \sum x_i y_i \\ \sum y_i \end{bmatrix}$$



## Algebraic: Over-determined system

- formally write the equation of the line through all the points

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

- Ac = y is an over-determined system
- it has an exact solution only if all the data points lie on a line, i.e. y is in the column space of A



## Over-determined system

- a compromise solution can be found by minimizing the residual r = y - Ac (as defined in unit I)
- find the minimum of $\rho = ||r||_2^2$

$= r^T r = (y - Ac)^T (y - Ac)$

$= y^T y - y^T Ac - c^T A^T y + c^T A^T Ac$

$= y^T y - 2y^T Ac + c^T A^T Ac$ [why?]

- to minimize $\rho$ requires that $\partial\rho/\partial c = 0$
  – differentiation with respect to a vector c
  – means a column vector of partial derivatives with respect to $c_1$, $c_2$

## Digression: vector derivatives

- how to differentiate with respect to a vector?
- we need some properties of vector derivatives ...
  - $\partial(Ax)/\partial x = A^T$
  - $\partial(x^TA)/\partial x = A$
  - $\partial(x^Tx)/\partial x = 2x$
  - $\partial(x^TAx)/\partial x = Ax + A^Tx$
- the notation convention for vector derivative $\partial/\partial x$ is NOT standardized with respect to transposes:
  - disagrees with Jacobian matrix definition



## Over-determined system

- now evaluate: $\partial\rho/\partial c = 0$
- $\partial\rho/\partial c = \partial/\partial c\,[y^Ty - 2y^TAc + c^TA^TAc]$
  $= -2A^Ty + [A^TAc + (A^TA)^Tc]$
  $= -2A^Ty + 2A^TAc$
  $= 0$



## Normal equations again

- these give the normal equations again in a slightly different disguise:

  $\mathbf{A^TAc = A^Ty}$
- these are the same equations as the normal equations derived previously from geometric reasoning
- to fit data to a line you can solve the normal equations for c
- Matlab example: *linefit(x,y)*
  - fits data points by solving the normal equations using backslash



## Linearizing a nonlinear relationship

- a line can be fit to an apparently nonlinear relationship by using a transformation
- example: to fit $y = c_1\exp(c_2x)$
  - take ln of both sides: $\ln y = \ln c_1 + c_2x$
  - put $v = \ln y$, $\alpha = c_2$, $\beta = \ln c_1$ to linearize the problem
  - you get $v = \alpha x + \beta$
  - fit the transformed data points: $(x_i, v_i)$
- this procedure minimizes the residuals of the transformed data fitted to a line, NOT the residuals of the original data



## Generalizing to arbitrary basis functions

- $y = F(x) = c_1f_1(x) + ... + c_nf_n(x)$ is the general linear fit equation
- the basis functions $f_j$ ....
  - are independent of the coefficients $c_j$
  - may be individually nonlinear
- $F(x)$ itself is a linear combination of the $f_j$
- as before we can write the general over-determined system $Ac = y$:

$$\begin{bmatrix} f_1(x_1) \cdots f_n(x_1) \\ f_1(x_2) \cdots f_n(x_2) \\ \vdots \quad\quad \vdots \\ f_1(x_m) \cdots f_n(x_m) \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$



## Generalizing to arbitrary basis functions

- ... and derive the same normal equations as for line fit

  $\mathbf{A^TAc = A^Ty}$
- some examples of basis functions:
  - $\{1, x\}$ .... the line fit from previously
  - $\{1, x, x^2, x^3, ..., x^k\}$ ..... a polynomial fit
  - $\{x^{-2/3}, x^{2/3}\}$
  - $\{\sin x, \cos x, \sin 2x, \cos 2x, ...\}$ .... Fourier fit
- the solution c to the normal equations gives the coefficients of the fitting function
- inverse $(A^TA)^{-1}$ gives useful statistical information
  - covariances of the fitting parameters c
  - variances $\sigma^2(c_j)$ on the diagonal

## Arbitrary basis functions: example

- fit data with the basis functions $\{x^{-1},x\}$
- Matlab example:
  - load $(x_i,y_i)$ data from xinvpx.dat
  - design matrix a = [1./x  x]
  - solve the normal equations for $(c_1,c_2)$
  - can use the mfile *fitnorm(x,y,Afun)* with ...
  - inline function *Afun=inline('[a./x x]')*



## Solving the normal equations

- $A^TAc = A^Ty$ is a linear system
  - Gaussian elimination is sometimes ok and provides the inverse $(A^TA)^{-1}$ for statistical information
  - LU decomposition is numerically equivalent, and more efficient if the covariance matrix not required
  - Cholesky also an option since $A^TA$ is positive definite
- all solution methods based on normal equations are inherently susceptible to roundoff error and other numerical problems
- a better approach is to apply decomposition techniques directly to the '*design matrix*' A
  - QR decomposition is a stable algorithm
  - SVD is best and completely avoids numerical problems



## Normal equations: Numerical issues

- normal equations may be close-to-singular
- very small pivots giving very large $c_j$'s that nearly cancel when F(x) is evaluated
  - data doesn't match the chosen basis functions
  - two of them may be an equally good or equally bad fit
  - $A^TA$ cannot distinguish between the functions so they get similar very large weights
- 'under-determined' due to these ambiguities
- SVD can resolve these problems automatically



## Normal equations: example

- find the least squares solution to the system Ac=y where A = [1 -1; 3  2; -2  4], y = $[4\ \ 1\ \ 3]^T$
- $A^TA$ = [14  -3; -3  21]
- $A^Ty = [1\ 10]^T$
- so normal equations $A^TAc = A^Ty$ are

$$\begin{bmatrix} 14 & -3 \\ -3 & 21 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 10 \end{bmatrix}$$

- exact LS solution is $c_1$ = 17/95, $c_2$ = 143/285



## Normal equations: another example

- find the least squares solution to Ac=y with

$$A = \begin{bmatrix} 3 & -6 \\ 4 & -8 \\ 0 & 1 \end{bmatrix} \quad y = \begin{bmatrix} -1 \\ 7 \\ 2 \end{bmatrix}$$

- normal equations are $A^TAc=A^Ty$ with

$$A^TA = \begin{bmatrix} 25 & -50 \\ -50 & 101 \end{bmatrix} \qquad A^Ty = \begin{bmatrix} 25 \\ -48 \end{bmatrix}$$

- Cholesky factorization is $A^TA = \begin{bmatrix} 5 & 0 \\ -10 & 1 \end{bmatrix} \begin{bmatrix} 5 & -10 \\ 0 & 1 \end{bmatrix}$
- solution is $\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$



## Solving the over-determined system

- fit a line through three points (1,1), (2,3), (3,4)
- the over-determined system Ac = y is

$$\begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$$

- can solve the normal equations $A^TAc = A^Ty$ and get the fitting function y = 1.5x - 0.3333 y
- what about applying Gaussian elimination directly to the over-determined system?
- you get an inconsistent augmented system .... $\left[\begin{array}{cc|c} 1 & 1 & 1 \\ 0 & -1 & 1 \\ 0 & 0 & -1 \end{array}\right]$

## Why QR factorization?

- why doesn't this approach give the LS solution?
- the elementary row ops applied to A are

$$MA = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 0 \end{bmatrix}$$

- problem is multiplication by M doesn't preserve the $L_2$-norm of the residual:

$$||M(y - Ac)||_2 \neq ||y - Ac||_2$$

- so the 'solution' walks away from the LS solution



## Why QR factorization?

- however an orthogonal matrix Q preserves norms:
  $||Qx||_2 = [(Qx)^T(Qx)]^{1/2} = [x^TQ^TQx]^{1/2} = [x^Tx]^{1/2} = ||x||_2$
- so to minimize $||y - Ac||_2$ we can ...
  - look for an *orthogonal* Q so that ....
  - minimizing $||Q^Ty - Q^TAc||_2$ is an easy problem to solve
- factorize the m×n matrix A = QR
  - m×n orthogonal Q
  - n×n upper triangular R
- first how do we find the factorization? ....



## QR factorization

- can apply the Gramm-Schmidt process to the columns of A
  - converts a basis $\{u_1,...,u_n\}$ into an orthonormal basis $\{q_1,...,q_n\}$
  - the most direct approach to finding QR
  - not numerically stable though
  - other more sophisticated algorithms are used in practice
- consider a full rank n×n matrix A
- the columns of A are
  - linearly independent
  - form a basis of the column space of A
- what is the relationship between
  $A = [u_1|u_2|...|u_n]$ and $Q = [q_1|q_2|...|q_n]$?



## QR Factorization

- each u can be expressed using the orthonormal basis as:
  $u_i = (u_i \cdot q_1)q_1 + (u_i \cdot q_2)q_2 + ... + (u_i \cdot q_n)q_n \qquad i = 1,2,...,n$
- in matrix form this is

$$\left[\, u_1 \,\middle|\, u_2 \,\middle|\, \cdots \,\middle|\, u_n \,\right] = \left[\, q_1 \,\middle|\, q_2 \,\middle|\, \cdots \,\middle|\, q_n \,\right] \begin{bmatrix} u_1 \cdot q_1 & u_2 \cdot q_1 & \cdots & u_n \cdot q_1 \\ u_1 \cdot q_2 & u_2 \cdot q_2 & \cdots & u_n \cdot q_2 \\ \vdots & \vdots & & \vdots \\ u_1 \cdot q_n & u_2 \cdot q_n & \cdots & u_n \cdot q_n \end{bmatrix}$$

- A = Q R
- this is the QR factorization required
- why is R upper triangular?
  - for j≥2 the G-S ensures that $q_j$ is orthogonal to all $u_1$, $u_2$, ... $u_{j-1}$



## QR factorization: example

- find the QR decomposition of $A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$
- apply Gramm-Schmidt to the columns of A to get columns of Q:

$$q_1 = \begin{bmatrix} 1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix}, q_2 = \begin{bmatrix} -2/\sqrt{6} \\ 1/\sqrt{6} \\ 1/\sqrt{6} \end{bmatrix} \; q_3 = \begin{bmatrix} 0 \\ -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

- then $R = \begin{bmatrix} u_1 \cdot q_1 & u_2 \cdot q_1 & u_3 \cdot q_1 \\ 0 & u_2 \cdot q_2 & u_3 \cdot q_2 \\ 0 & 0 & u_3 \cdot q_3 \end{bmatrix} = \begin{bmatrix} 3/\sqrt{3} & 2/\sqrt{3} & 1/\sqrt{3} \\ 0 & 2/\sqrt{6} & 1/\sqrt{6} \\ 0 & 0 & 1/\sqrt{2} \end{bmatrix}$
- so $A = \begin{bmatrix} 1/\sqrt{3} & -2/\sqrt{6} & 0 \\ 1/\sqrt{3} & 1/\sqrt{6} & -1/\sqrt{2} \\ 1/\sqrt{3} & 1/\sqrt{6} & 1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 3/\sqrt{3} & 2/\sqrt{3} & 1/\sqrt{3} \\ 0 & 2/\sqrt{6} & 1/\sqrt{6} \\ 0 & 0 & 1/\sqrt{2} \end{bmatrix}$



## Skinny QR factorization

- suppose A is m×n and has rank n (linearly independent columns)
- previous method still works and gives A=QR with
  - Q m×n
  - R n×n
- this is the *economy (skinny) QR factorization*
  - contains all the information necessary to reconstruct A

## Full vs skinny QR factorization

- can also have a full QR factorization with extra stuff
  - Q m×m
  - R m×n
- last m-n rows of R are zero
- first n columns of Q span the column space of A
- last m-n columns of Q span the orthogonal complement of the column space of A
  - this is the nullspace of A
  - not needed to solve the LS problem
  - so the skinny QR factorization is good enough for LS problems
- in general if A has rank k<n only the first k columns of Q are needed



## QR Factorization in Matlab

- the Matlab function *[Q,R] = qr(A)* provides the QR factorization of A
- the function *[Q,R] = qr(A,0)* provides the skinny QR factorization
  - we'll always use this one
- example:
  - calculate the skinny QR factorization of the A in the simple 3-point fitting example on slide 23
  - check that the residuals are small



## QR solution to the LS problem

- a solution to the LS problem Ac = y is obtained by minimizing the 2-norm of the residual
- some simple calculation and apply the QR factorization:

$$\begin{aligned}||y - Ac||_2 &= ||Q^T(y - Ac)||_2\\ &= ||Q^Ty - Q^TAc||_2\\ &= ||Q^Ty - Rc||_2\end{aligned}$$

- write the full QR factorization in block matrix form:

$$Q = \left[\, Q_1 \,\middle|\, Q_2 \,\right] \qquad R = \left[\frac{R_1}{0}\right]$$

- then the column vector residual above can be written

$$Q^T y - Rc = \left[\begin{array}{c} Q_1^T y - R_1 c \\ Q_2^T y - 0 \end{array}\right]$$



## QR solution to LS problem

- so the 2-norm of the LS residual is

$$||Q^T y - Rc||_2 = ||Q_1^T y - R_1 c||_2 + ||Q_2^T y||_2$$

- magnitude of the residual consists of two parts:
  - the first part can be zero by selecting a suitable c vector
  - the second part cannot be influenced by the choice of c so is irrelevant to the LS problem
  - that's why only the skinny QR factorization is needed
  - we'll drop the $Q_2$ part of Q
- to solve the LS problem Ac = y choose c so that

**$Q^Ty = Rc$**

  - where A = QR is the skinny QR factorization



## QR, LS, and Matlab

- solve the simple 3-point line fit example on slide 23 using Matlab and the skinny QR factorization
- the backslash operator in Matlab A\y
  - if A rectangular (over-determined) Matlab automatically applies the QR factorization and finds the LS solution
  - Matlab assumes the LS solution is desired when backslash in encountered under this circumstance
  - Q is only used to evaluate $Q^Ty$
  - can avoid storing the Q matrix by careful algorithm
- for a standard system notated as Ax=b the LS solution becomes: $Q^Tb = Rx$



## QR vs Cholesky: example

- minimize ||Ax - b|| with $A = \begin{bmatrix} 1 & -1 \\ 0 & 10^{-5} \\ 0 & 0 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 10^{-5} \\ 1 \end{bmatrix}$
- the normal equations $A^TAx = A^Tb$

$$\begin{bmatrix} 1 & -1 \\ -1 & 1+10^{-10} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 10^{-10} \end{bmatrix}$$

have solution $x_1 = x_2 = 1$

- compare solution to this LS problem with Cholesky and QR keeping 8 significant digits ....

## QR vs Cholesky: example solution

**method 1** (Cholesky factorization)

$A^TA$ and $A^Tb$ rounded to 8 digits:

$$A^TA = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \quad A^Tb = \begin{bmatrix} 0 \\ 10^{-10} \end{bmatrix}$$

no solution (singular matrix)

**method 2** (QR-factorization): factor $A = QR$ and solve $Rx = Q^Tb$

$$Q = \begin{bmatrix} 1 & -1 \\ 0 & 10^{-5} \\ 0 & 0 \end{bmatrix}, \quad R = \begin{bmatrix} 1 & -1 \\ 0 & 10^{-5} \end{bmatrix}, \quad Q^Tb = \begin{bmatrix} 0 \\ 10^{-5} \end{bmatrix}$$

rouding does not change any values

solution of $Rx = Q^Tb$ is $x_1 = 1$, $x_2 = 1$



## QR vs normal equations

- in the example Cholesky solution of the normal equations fails due to rounding error
- normal equations can be very inaccurate for ill-conditioned systems where cond($A^TA$) is large
- but when m » n the normal equations involve
  - half the arithmetic as compared to QR
  - less storage than QR
- if A is ill-conditioned and ...
  - residual is small then the normal equations are less accurate than QR
  - residual is large then both methods give an inaccurate LS solution
- choosing the right algorithm is not an easy decision



## A third choice: SVD and LS problem

- we'll use the standard notation for the LS problem Ax = b with A m×n
- apply the SVD to A ...
- the norm of the residual is ...

  ||r|| = ||Ax - b||

  = $||USV^Tx - b||$

  = $||SV^Tx - U^Tb||$
- minimizing the residual is equivalent to minimizing ||Sz - d|| where $z=V^Tx$ and $d=U^Tb$



## SVD and LS problem

- written in blocks this is

$$\left\| \begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ 0 & 0 & \cdots & \sigma_n \\ \hline 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} - \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} \right\|$$

- to choose z so that ||r|| is minimal requires (k=1,...,n)

  $z_k = d_k / \sigma_k$ $\quad\quad \sigma_k \neq 0$

  $z_k = 0$ $\quad\quad \sigma_k = 0$
- we have $d = U^Tb$ and $z = V^Tx$



## SVD and LS problem: pseudo-inverse

- write $S^+$ to denote the transpose of the matrix obtained by replacing each non-zero $\sigma_k$ by its reciprocal
  - this is NOT a regular matrix inverse since S is not square
- the minimization condition can be written as $V^Tx = S^+U^Tb$ so ...
- the LS solution to Ax = b (with $A = USV^T$) is

  **$x = V S^+U^Tb$**
- the matrix $A^+ = VS^+U^T$ is called the *pseudo-inverse* of the coefficient matrix A
  - behaves for rectangular matrices like the normal inverse for square matrices



## SVD and LS problem: sales pitch

- the SVD method is ....
  - powerful
  - convenient
  - intuitive
  - numerically advantageous
- problems with ill-conditioning and large residuals can be circumvented automatically
- the SVD can solve problems for which both the normal equations and QR factorization fail

## SVD and LS problem: zeroing

- use a zero entry in $S^+$ if $\sigma_j = 0$ (to machine precision)
- this forces a zero coefficient in the linear combination of basis functions that gives the fitting equation ....
- ... instead of a random large coefficient that has to delicately cancel with another one
- if the ratio $\sigma_j / \sigma_1 \sim n\varepsilon_m$ then zero the entry in the pseudo-inverse matrix since the value is probably corrupted by roundoff anyway



## Polynomial curve fitting

- consider curve-fitting with mononomial basis functions $\{1, x, x^2, ..., x^k\}$
- can solve the least-squares problem with any of the previous techniques (normal equations, QR, SVD)
- the simplicity of the basis functions allows some trickery when setting up the equations
  – to fit known data in column vectors x and y to the quadratic $y = c_1x^2 + c_2x + c_3$ requires solving the over-determined system

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ \vdots & \vdots & \vdots \\ x_n^2 & x_n & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$



## Polynomial curve fitting in Matlab

- these types of coefficient matrices are called *Vandermonde matrices*
  – but see interpolation section of this unit .....
- in Matlab: *A = [x.^2 x ones(size(x)) ]*
  – the least squares solution can then be found with *x = A\y*
- Matlab can automate this with the *polyfit* function
  – fits data to an arbitrary nth order polynomial by ...
  – constructing the design matrix A and ...
  – obtaining the least squares solution by QR factorization
  – vector *p = polyfit(x,y,n)* contains the coefficients of the fitting polynomial in descending order
  – the polynomial can be evaluated at xf by *yf = polyval(p,xf)*
  – it's also possible to analyse residuals and uncertainties with optional parameters to polyfit and polyval functions



## Interpolation

- general remarks
- polynomial interpolation
  – mononomial basis functions
  – Lagrange basis
  – Newton basis ... divided differences
  – polynomial wiggle
- piecewise polynomial interpolation
  – cubic splines
  – Bezier curves and B-splines
- Matlab comments



## Interpolation

- interpolation is the 'exact' form of curve-fitting
- for given data points $(x_i, y_i)$
  – curve-fitting finds the solution to the over-determined system that is *closest* to the data (least-squares solution)
  – interpolation finds a function that passes through *all* the points and 'fills in the spaces' smoothly
- piece-wise linear interpolation is the simplest, but lacks smoothness at the *support points* $(x_i, y_i)$
- to approximate a function outside the range of values of $x_i$ use *extrapolation*



## Basic ideas

- given: support points $(x_i, y_i)$ i = 1, ... ,n that are supposed to result from a function y = f(x)
- to find: an interpolating function y = *F*(x) valid for a range of x values that includes the $x_i$'s
- requirements
  – $F(x_i) = f(x_i)$, i=1, ...,n
  – *F*(x) should be a good approximation of the f(x)
- the function f(x) may
  – not be known at all
  – not be easy (or possible) to express symbolically
  – not easy to evaluate
  – known only in tabular format

## Basis functions

- n basis functions $\Phi_1, \Phi_2, \ldots, \Phi_n$ can be used to define the interpolating function
  $F(x) = a_1\Phi_1(x) + a_2\Phi_2(x) + \ldots + a_n\Phi_n(x)$
- polynomial basis functions are common
  $F(x) = a_1 + a_2x^2 + \ldots + a_nx^n$
- .....so are Fourier interpolations
  $F(x) = a_1 + a_2e^{ix} + \ldots + a_ne^{(n-1)ix}$
- polynomials are easy to evaluate but there are numerical issues ...



## Mononomial basis functions

- n basis functions are $x^0, x^1, x^2, \ldots, x^{n-1}$
- there is a unique polynomial of degree n-1 passing through n support points
- so no matter how we get the interpolating polynomial it will be unique
- however it can be expressed in multiple ways with different coefficients multiplying the basis functions
  - for instance the *offset monomials* $(x-a)^0, (x-a)^1, (x-a)^2, \ldots, (x-a)^{n-1}$ could be used
  - we develop techniques to use alternative basis functions for the interpolating polynomial (Lagrange, Newton)
- but first look at simple mononomials....



## Vandermonde systems

- this is the simple, direct approach to the problem
- example
  - quadratic interpolating polynomial $y = c_1x^2 + c_2x + c_3$
  - support points $(x_1,y_1)$, $(x_2,y_2)$ and $(x_3,y_3)$
  - sub the points into the interpolating function and you get the *Vandermonde system*:

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

- compare this system to that for curve fitting....



## Vandermonde systems

- $[c_1\ c_2\ c_3]$ is supposed to be
  - the *exact*, *unique* solution to the 3×3 non-singular system in interpolation
  - the least squares solution to an m×3 over-determined system in curve-fitting
- there is a Matlab function to evaluate the Vandermonde matrix: *vander( [$y_1$ $y_2$ $y_3$] )*
- but...Vandermonde systems often
  - are ill-conditioned
  - have solutions with very large order of magnitude differences in the coefficients



## Vandermonde systems: example

- yearly prices

```
year = [1986 1988 1990 1992 1994 1996]';
price = [133.5 132.2 138.7 141.5 137.6 144.2]';
A = vander(year);
c = A\price;
y = linspace(min(year),max(year));
p = polyval(c,y);
plot(year,price,'o',y,p,'-');
```

- cond(A) ~ $10^{31}$... not good ☹☹
  - serious problems with roundoff error in calculating the coefficients [see the oscillations in the plot]



## Vandermonde systems: example

- coefficients vary over 16 orders of magnitude
- adding and subtracting very large quantities is supposed to give a delicate balance, but the result is mostly roundoff corruption
- a simple re-scaling using offset mononomials can sometimes fix the problem

```
ys = year - mean(year)
A = vander(ys)
```

- system not ill-conditioned now ☺☺
- coefficients vary over only 5 orders of magnitude
  - no spurious oscillations now from roundoff

## Lagrange polynomials

- an alternative basis for polynomials of degree n
- first degree polynomial first....
- the (linear) poly passing through $(x_1,y_1)$ & $(x_2,y_2)$ is:

$$P_1(x) = y_1 \underbrace{\frac{x - x_2}{x_1 - x_2}}_{L_1} + y_2 \underbrace{\frac{x - x_1}{x_2 - x_1}}_{L_2}$$

- find this representation by ...
  – writing the Vandermonde system
  – solving for $c_1$ and $c_2$ and ...
  – re-arranging the polynomial in the form above



## Lagrange polynomials

- now the interpolating polynomial is expressed as
  $p_1(x) = y_1 L_1(x) + y_2 L_2(x)$
- the basis functions $L_1(x)$ & $L_2(x)$ are called first-degree *Lagrange interpolating polynomials*
- continuing to second degree we get...

$$P_2(x) = y_1 \underbrace{\frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}}_{L_1(x)} + y_2 \underbrace{\frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)}}_{L_2(x)} + y_3 \underbrace{\frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}}_{L_3(x)}$$



## Lagrange polynomials

- in general for nth order interpolation
  **$p_{n-1}(x) = y_1 L_1(x) + y_2 L_2(x) + \ldots + y_n L_n(x)$**
  where....
- the nth degree *Lagrange interpolating polynomials* are

$$L_j(x) = \prod_{k=1, k \neq j}^{n} \frac{x - x_k}{x_j - x_k}$$

- *notation warning*: the meaning of the symbol '$L_j(x)$' depends on the degree of interpolating polynomial being used

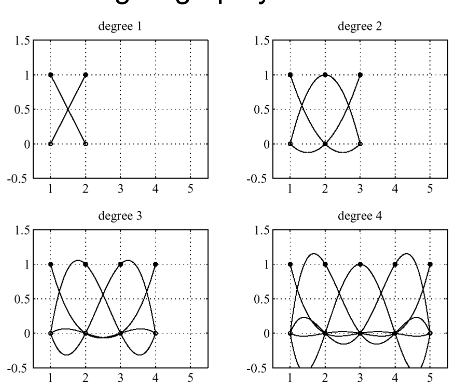


## Lagrange polynomials

- no system of equations has to be solved to get the interpolating polynomial with Lagrange polynomial basis functions
- not susceptible to roundoff error problems
- illustrative Matlab function *lagrint* shows an efficient implementation of Lagrange polys



## Lagrange polynomials






## Lagrange polynomials

- both Vandermonde and Lagrange can be improved on
  – too much arithmetic involved
  – data points cannot be added or deleted without starting the calculations again from scratch
  – we don't know what degree polynomial to use up front and we can't adjust that later without starting again
- an alternative is ....

## Newton basis functions

- Newton basis functions are 1, $(x-x_1)$, $(x-x_1)(x-x_2)$, $(x-x_1)(x-x_2)(x-x_3)$, ..., $(x-x_1)(x-x_2)...(x-x_n)$
- express the interpolating polynomial $p_n(x)$ in terms of this basis:

$$\begin{aligned} P_n(x) &= c_1 + c_2(x-x_1) + c_3(x-x_1)(x-x_2) + \\ &\quad \cdots + c_{n+1}(x-x_1)(x-x_2)\cdots(x-x_n) \end{aligned}$$

- Newton basis functions
  - are computationally efficient
  - have important relevance in numerical integration
  - have good numerical properties



## Newton basis functions

- first consider the quadratic version...
  $p_2(x) = c_1 + c_2(x-x_1) + c_3(x-x_1)(x-x_2)$
- apply the support point constraints $p_n(x_i) = y_i$

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & (x_2-x_1) & 0 \\ 1 & (x_3-x_1) & (x_3-x_1)(x_3-x_2) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

- and apply forward elimination, first to get

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & (x_3-x_2) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ \frac{y_2-y_1}{x_2-x_1} \\ \frac{y_3-y_1}{x_3-x_1} \end{bmatrix}$$



## ....First order divided differences

- this can be written compactly in the form

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & (x_3-x_2) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ f[x_1,x_2] \\ f[x_1,x_3] \end{bmatrix}$$

- where $f[x_1,x_2]$ is a *first order divided difference*

$$f[x_i,x_j] = \frac{y_j - y_i}{x_j - x_i}$$

- continuing with the elimination we get...



## ....Second order divided differences

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ f[x_1,x_2] \\ f[x_1,x_2,x_3] \end{bmatrix}$$

- where $f[x_1,x_2,x_3]$ is a *second order divided difference*

$$f[x_1,x_2,x_3] = \frac{f[x_2,x_3] - f[x_1,x_2]}{x_3 - x_1}$$

- in general, for nth degree poly interpolation we get...



## ....nth order divided differences

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} f[x_1] \\ f[x_1,x_2] \\ f[x_1,x_2,x_3] \\ \vdots \\ f[x_1,x_2,\cdots,x_{n+1}] \end{bmatrix}$$

- where $f[x_1,x_2,\ldots,x_n]$ is an *nth order divided difference*

$$f[x_1,x_2,\cdots,x_n] = \frac{f[x_2,\cdots,x_n] - f[x_1,\cdots,x_{n-1}]}{x_n - x_1}$$

- *notation warning*: the indices usually start at zero, but here they are adjusted to be consistent with Matlab convention .... array indices cannot be zero



## Divided differences

- find the 3rd degree Newton interpolating poly for first four data points in the table below

| $x_i$ | $y_i$ | $f[x_i,x_{i+1}]$ | $f[x_i,\cdots,x_{i+2}]$ | $f[x_i,\cdots,x_{i+3}]$ | $f[x_i,\cdots,x_{i+4}]$ |
|---|---|---|---|---|---|
| 3.2 | 22.0 | 8.400 | 2.856 | −0.528 | 0.256 |
| 2.7 | 17.8 | 2.118 | 2.012 | 0.865 | |
| 1.0 | 14.2 | 6.342 | 2.263 | | |
| 4.8 | 38.2 | 16.750 | | | |
| 5.6 | 51.7 | | | | |

- divided differences can be calculated as shown

$$\begin{aligned} p_3(x) &= 22.0 + 8.400(x-3.2) + 2.856(x-3.2)(x-2.7) \\ &\quad - 0.528(x-3.2)(x-2.7)(x-1.0) \end{aligned}$$

## Adding a data point is easy

- to find the 4th degree poly that fits all the five points in the table
  - calculate the extra divided difference(s) and get the last coefficient (see the table on previous slide)
  - you start with $p_3(x)$ and add one term to it

$$p_4(x) = p_3(x) + 0.256(x - 3.2)(x - 2.7)(x - 1.0)(x - 4.8)$$

- data points can easily be added or deleted
- a better form for computation is with nested multiplication (Horner's rule):

$$p_3(x) = ((-0.528(x - 1.0) + 2.856)(x - 2.7) + 8.400)(x - 3.2) + 22.0$$

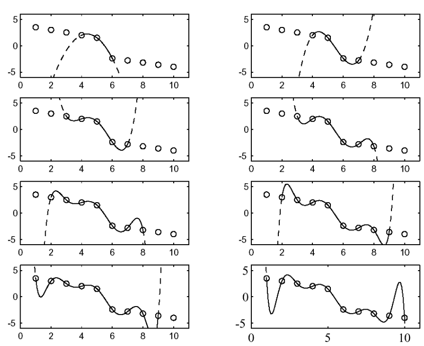


## Wiggle

- more support points $\Rightarrow$ higher degree interpolating polynomial $F(x)$ is required
- $F(x_i) = y_i$ is OK since the interpolant passes through all the support points
- but higher order polynomials will exhibit rapid oscillations between the support points
- this polynomial wiggle is an extraneous artifact of the interpolation method
- limits the applicability of higher order polynomial interpolation to improve accuracy
- suggests piecewise methods....



## Wiggle






## Piecewise polynomial interpolation

- instead of finding a single higher order interpolating polynomial passing through all the support points...
- ...find multiple lower order polynomials going through subsets of support points
- the joints where these fit together are called *breakpoints* or *knots*
- desire for global performance raises complexity by demanding constraints on how the local interpolants relate to their neighbours



## Continuity constraints

- can demand f(x), f′(x) and/or f″(x) to be continuous at the breakpoints
- fundamental issue: are neighbouring interpolating polynomials constrained with respect to each other? ...
  - leads to large, sparse systems to solve (cubic splines)
- ... or with respect to the original function?
  - leads to de-coupled equations to solve, but you need more information from the data points (Hermite)
  - de-coupled polynomials require identification of the appropriate sub-interval for evaluation



## Searching for sub-intervals

- $p_j(x)$ is the interpolating polynomial for the jth sub-interval $(x_j, x_{j+1})$
- how to locate the sub-interval which brackets a given test point x?
- assume the data is ordered or manage appropriate book-keeping to order it
- two techniques that work:
  - incremental search ... look through the data in sequence until the bracketing interval is found
  - binary search ... use bisection to locate the sub-interval that x lies in

## Piecewise interpolants

- piecewise linear on sub-intervals
  - solve for coefficients using Lagrange interpolating polynomials
- piecewise quadratic: conic splines
  - exact representation of lines, circle, ellipses, parabolas and hyperbolas
- piecewise cubic: cubic splines
- what's a spline?
  - carries some element of slope or curve shape constraint
  - represent the curve of minimum strain energy (abstraction of beam theory, e.g. f''''= zero at break points)
  - drafting device from old times



## Why cubic splines?

- piecewise linear interpolation provides....
  - a smooth *y′*(x) everywhere and...
  - zero *y″*(x) inside the sub-intervals but...
  - undefined (or infinite) *y″*(x) at the breakpoints $x_j$
- cubic splines improve the behaviour of *y″*(x) ....



## Constructing cubic splines

- consider n tabulated support points

  $y_j = y(x_j)$, j = 1,2,...,n
- first use *linear* interpolation on the jth sub-interval $(x_j, x_{j+1})$
- the local interpolant is

  $y = F(x) = Ay_j + By_{j+1}$ ..... eqn 1

  with coefficients

$$A \equiv \frac{x_{j+1} - x}{x_{j+1} - x_j} \qquad B \equiv 1 - A = \frac{x - x_j}{x_{j+1} - x_j}$$

  derived from Lagrange interpolating polynomials



## Constructing cubic splines

- suppose (blindly for now) that we have tabulated values for the second derivative at each breakpoint $y_j''(x_j)$
- add to the RHS of eqn [1] a cubic polynomial so
  - $y = Ay_j + By_{j+1} + p(x)$
- to ensure that this doesn't alter continuity at the breakpoints we have to have
  - $p(x_j) = p(x_{j+1}) = 0$
- to ensure that the second derivative is continuous at the breakpoints we have to have
  - $y''$ varying linearly (i.e. IS linear) from $y_j''$ to $y_{j+1}''$
- can we do all this?



## Constructing cubic splines

- YES ... and to make it work we have to have

$$y = Ay_j + By_{j+1} + Cy_j'' + Dy_{j+1}''$$

$$C \equiv \frac{1}{6}(A^3 - A)(x_{j+1} - x_j)^2$$

$$D \equiv \frac{1}{6}(B^3 - B)(x_{j+1} - x_j)^2$$

- the A, B, C, D factors depend on x:
  - A, B are linearly dependent on x (from the previous expressions) and ...
  - C, D are cubicly dependent on x (through A, B)



## Cconstructing cubic splines

- calculate the derivatives and check that it works

$$\frac{dy}{dx} = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{3A^2 - 1}{6}(x_{j+1} - x_j)y_j'' + \frac{3B^2 - 1}{6}(x_{j+1} - x_j)y_{j+1}''$$

- this eqn [2] comes from the derivatives

$$\frac{dA}{dx} = \frac{-1}{x_{j+1} - x_j}$$

$$\frac{dB}{dx} = \frac{1}{x_{j+1} - x_j}$$

$$\frac{dC}{dx} = \frac{1}{6}(3A^2 - 1)\frac{dA}{dx}(x_{j+1} - x_j)^2 = -\frac{1}{6}(3A^2 - 1)(x_{j+1} - x_j)$$

$$\frac{dD}{dx} = \frac{1}{6}(3B^2 + 1)\frac{dB}{dx}(x_{j+1} - x_j)^2 = \frac{1}{6}(3B^2 + 1)(x_{j+1} - x_j)$$

## Cconstructing cubic splines

- the second derivative is

$$\begin{aligned}\frac{d^2y}{dx^2} &= -\frac{d}{dx}\left(\frac{3A^2-1}{6}\right)(x_{j+1}-x_j)y''_j + \frac{d}{dx}\left(\frac{3B^2-1}{6}\right)(x_{j+1}-x_j)y''_{j+1}\\ &= Ay''_j + By''_{j+1}\end{aligned}$$

- A = 1 and B = 0 when $x=x_j$ and opposite for $x_{j+1}$ so $y''(x_j) = y_j''$ and $y''(x_{j+1}) = y_{j+1}''$
- this ensures that
  - the second derivative actually agrees with the tabulated assumed values
  - the continuity condition for the second derivatives at the breakpoints is satisfied
- so now what do we do about our assumed $y_j''$ ??



## Constructing cubic splines

- we now get an equation to solve for these unknown (assumed) $y_j''$ values
- the trick is to demand continuity of the first derivative at the breakpoints
- use eqn [2] to evaluate it at
  - $x = x_j$ the right endpoint of $(x_{j-1},x_j)$, and
  - $x = x_j$ the left endpoint of $(x_j,x_{j+1})$ and
  - equate these two for continuity
- after some [messy] simplification you get the n-2 **cubic spline equations** (j= 2, ...,n-1) [eqn 3]

$$\boxed{\frac{x_j-x_{j-1}}{6}y''_{j-1} + \frac{x_{j+1}-x_{j-1}}{3}y''_j + \frac{x_{j+1}-x_j}{6}y''_{j+1} = \frac{y_{j+1}-y_j}{x_{j+1}-x_j} - \frac{y_j-y_{j-1}}{x_j-x_{j-1}}}$$



## Cubic spline equations!

$$\begin{bmatrix} h_1 & 2(h_1+h_2) & h_2 & & & \\ & h_2 & 2(h_2+h_3) & h_3 & & \\ & & & \ddots & & \\ & & & h_{n-2} & 2(h_{n-2}+h_{n-1}) & h_{n-1} \end{bmatrix} \begin{bmatrix} y''_1 \\ y''_2 \\ y''_3 \\ \vdots \\ y''_{n-1} \\ y''_n \end{bmatrix}$$

$$= 6\begin{bmatrix} f[x_2,x_3]-f[x_1,x_2] \\ f[x_3,x_4]-f[x_2,x_3] \\ \vdots \\ f[x_{n-2},x_{n-1}]-f[x_{n-3},x_{n-2}] \end{bmatrix}$$

- $h_j = x_{j+1}-x_j, j = 1,2,\cdots,n-1$

  is the size of the jth interval



## Cubic spline equations

- these are n-2 equations in the n unknown breakpoint second derivatives $y_j''$, j = 1, ...., n
- there is a two parameter family of solutions unless we specify additional constraints
- for a unique solution we can specify boundary conditions at the endpoints $x_1$ and $x_n$
  - defines the behaviour of the interpolating function at the global endpoints
- three choices are common for cubic spline boundary conditions ....

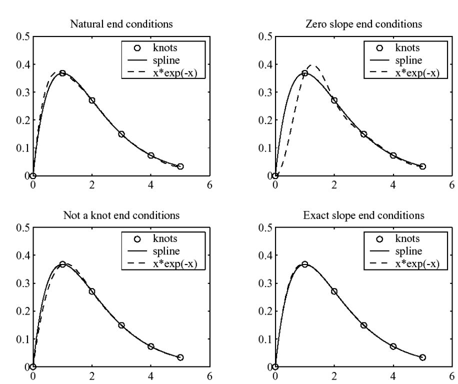


## Cubic splines: boundary conditions

- $y_1'' = y_n'' = 0$, i.e. a smooth flow away from the two global endpoints (these called *natural* cubic splines) OR
- set $y_1''$ and/or $y_n''$ to values calculated from eqn [2] so that $y_1'$ and/or $y_n'$ have desired values (*specified endpoint slopes*)
- force continuity of the third derivative at the first and last breakpoints, e.g. $y_2'''$ from first or second interval is to be the identical value, similarly $y_{n-1}'''$
  - this is called the *not-a-knot condition*
  - it effectively makes the first two interpolating cubics identical



## Cubic splines: boundary conditions



## Cubic splines: computational issues

- have to solve a linear system for the n unknowns $y_j''$ consisting of
  - n-2 spine equations
  - two boundary conditions
- this is a (sparse) tri-diagonal linear system
- only the main diagonal and its neighbours have non-zero entries
- this reflects the fact that each sub-interval in the interpolation problem is coupled only to its two nearest neighbours
- special efficient algorithms are available for solving tri-diagonal systems



## Cubic splines: example

- find the equations for the four natural cubic splines which interpolate the data points:
  (-2,4), (-1,-1), (0,2), (1,1), (2,8)
- all $h_j = x_{j+1} - x_j = 1$ for this data [nice]
- the spline equations

$$\frac{h_j}{6}y''_{j-1} + \frac{h_{j+1}+h_j}{3}y''_j + \frac{h_{j+1}}{6}y''_{j+1} = \frac{y_{j+1}-y_j}{h_{j+1}} - \frac{y_j - y_{j-1}}{h_j}$$

are (j = 2,3,4):

$$\frac{1}{6}y''_{j-1} + \frac{2}{3}y''_j + \frac{1}{6}y''_{j+1} = y_{j+1} - 2y_j + y_{j-1}$$



## Cubic splines: example (cont)

- written out this system is

$$\begin{array}{llllllll} j=2: & \frac{1}{6}y''_1 & + & \frac{2}{3}y''_2 & + & \frac{1}{6}y''_3 & & & & = y_3 - 2y_2 + y_1 \\ j=3: & & & \frac{1}{6}y''_2 & + & \frac{2}{3}y''_3 & + & \frac{1}{6}y''_4 & & = y_4 - 2y_3 + y_2 \\ j=4: & & & & & \frac{1}{6}y''_3 & + & \frac{2}{3}y''_4 & + \frac{1}{6}y''_5 & = y_5 - 2y_4 + y_3 \end{array}$$

- natural splines have $y''_1 = y''_5 = 0$ so the system simplifies to:

$$\begin{array}{rcrcrcr} 4y''_2 & + & y''_3 & & & = & 48 \\ y''_2 & + & 4y''_3 & + & y''_4 & = & -24 \\ & & y''_3 & + & 4y''_4 & = & 48 \end{array}$$

solution

$$\boxed{\begin{array}{rcl} y''_1 & = & 0 \\ y''_2 & = & 15.4286 \\ y''_3 & = & -13.7143 \\ y''_4 & = & 15.4286 \\ y''_5 & = & 0 \end{array}}$$



## Cubic splines: example (cont)

- step #1 is complete now we have the second derivatives at the breakpoints
- now use these values to obtain the cubic spline equation coefficients A,B,C,D [formulas on slides 75&77], with j=1,...,4
- first j = 1:

$$A = x_2 - x = -(1+x)$$
$$B = 1 - A = x + 2$$
$$C = \frac{1}{6}(A^3 - A) = \frac{1}{6}[-(1+x)^3 + (1+x)]$$
$$D = \frac{1}{6}(B^3 - B) = \frac{1}{6}[(x+2)^3 - (x+2)]$$



## Cubic splines: example (cont)

- the spline equation for the first interval 2 < x < -1 is then given by

$$\begin{array}{rcl} y & = & Ay_1 + By_2 + Cy''_1 + Dy''_2 \\ & = & -(1+x)(4) + (x+2)(-1) + 0 + \frac{1}{6}[(x+2)^3 - (x+2)](15.4286) \\ & = & 2.57(x+2)^3 - 4(x+1) - 3.57(x+2) \end{array}$$

....S1

- j = 2 gives the spline equation for the second interval -1 < x < 0

$$A = x_3 - x = -x$$
$$B = 1 - A = x + 1$$
$$C = \frac{1}{6}(A^3 - A) = \frac{1}{6}[-x^3 + x]$$
$$D = \frac{1}{6}(B^3 - B) = \frac{1}{6}[(x+1)^3 - (x+1)]$$



## Cubic splines: example (cont)

$$\begin{array}{rcl} y & = & Ay_2 + By_3 + Cy''_2 + Dy''_3 \\ & = & -x(-1) + (x+1)(2) + \frac{1}{6}[-x^3 + x](15.4286) + \frac{1}{6}[(x+1)^3 - (x+1)](-13.7143) \\ & = & -2.57x^3 - 2.29(x+1)^3 + 3.57x + 4.29(x+1) \end{array}$$

....S2

- for the third interval 0 < x < 1 we have j = 3:

$$A = x_4 - x = 1 - x$$
$$B = 1 - A = x$$
$$C = \frac{1}{6}(A^3 - A) = \frac{1}{6}[(1-x)^3 - (1-x)]$$
$$D = \frac{1}{6}(B^3 - B) = \frac{1}{6}[x^3 - x]$$

and the third spline equation ....

## Cubic splines: example (cont)

$$\begin{aligned} y &= Ay_3 + By_4 + Cy_3'' + Dy_4'' \\ &= (1-x)(2) + x(1) + \frac{1}{6}[(1-x)^3 - (1-x)](-13.7143) + \frac{1}{6}[x^3 - x](15.4286) \\ &= -2.29(1-x)^3 + 2.57x^3 + 4.29(1-x) - 1.57x \end{aligned}$$ ..... S3

- finally, for the 4th interval 1 < x < 2 we have j = 4:

$$A = x_5 - x = 2 - x$$
$$B = 1 - A = x - 1$$
$$C = \frac{1}{6}(A^3 - A) = \frac{1}{6}[(2-x)^3 - (2-x)]$$
$$D = \frac{1}{6}(B^3 - B) = \frac{1}{6}[(x-1)^3 - (x-1)]$$

and the fourth spline equation ....



## Cubic splines: example (cont)

$$\begin{aligned} y &= Ay_4 + By_5 + Cy_4'' + Dy_5'' \\ &= (2-x)(1) + (x-1)(8) + \frac{1}{6}[(2-x)^3 - (2-x)](15.4286) + 0 \\ &= 2.57(2-x)^3 - 1.57(2-x) + 8(x-1) \end{aligned}$$ .... S4

- these equations (S1-S4) are the four natural cubic spline equations for y over the interval -2 < x < 2
- to evaluate y(x) you decide which interval x lies in and use the applicable equation



## Font wars

- proportional fonts are rendered at any size using curves determined by interpolation through specified points
- the curves are rasterized for display following 'hints' to keep the fonts realistic in small sizes
- Type I fonts, Adobe 1985
  - used in postscript and acrobat
  - based on cubic splines as developed above
- TrueType fonts, Apple 1988 + now MS
  - conic splines
  - a subset of cubic splines with simpler interpolation
  - faster rendering but require more 'hints'



## Interpolation: Matlab implementation

- the basic Matlab functions you need to know are
  - *interp1*: one-dimensional interpolation with piecewise polynomials
  - *spline*: one-dimensional interpolation with cubic splines using not-a-knot or fixed-slope end conditions
- there are others too
  - multi-dimensional interpolations
  - a sophisticated spline toolbox



## Interpolation: Matlab implementation

- ytest = interp1(x,y,xtest,method)
  - y = tabulated function values
  - xtest = test x value to evaluate the interpolant
  - x = tabulated x values for interpolation (1:n default)
  - method = 'nearest' [nearest neighbour constant], 'linear', 'cubic' [cubic polys s.t. interpolant & first derivative are continuous at the breakpoints], or 'spline' [cubic splines, same as spline function]
  - ytest = calculated interpolated value corresponding to xtest
- default for splines is not-a-knot end conditions
- if size(y) = size(x)+2, first and last elements are used for $y'(x_1)$ and $y'(x_n)$ end slopes