Lecture 10
Computational tricks and Techniques

Forrest Brewer

Embedded Computation

- Computation involves Tradeoffs
  - Space vs. Time vs. Power vs. Accuracy
- Solution depends on resources as well as constraints
  - Availability of large space for tables may obviate calculations
  - Lack of sufficient scratch-pad RAM for temporary space may cause restructuring of whole design
  - Special Processor or peripheral features can eliminate many steps – but tend to create resource arbitration issues
- Some computations admit incremental improvement
  - Often possible to iterate very few times on average
    - Timing is data dependent
    - If careful, allows for soft-fail when resources are exceeded

Overview

- Basic Operations
  - Multiply (high precision)
  - Divide
- Polynomial representation of functions
  - Evaluation
  - Interpolation
  - Splines
- Putting it together

Multiply

- It is not so uncommon to work on a machine without hardware multiply or divide support
  - Can we do better than $O(n^2)$ operations to multiply n-bit numbers?
- Idea 1: Table Lookup
  - Not so good – for 8-bitx8-bit we need 65k 16-bit entries
    - By symmetry, we can reduce this to 32k entries – still very costly – we often have to produce 16-bit x 16-bit products…

Multiply – cont'd

- Idea 2: Table of Squares
  - Consider: $(\frac{u+v}{2})(\frac{u-v}{2}) = (1/4)(u^2 - v^2)$
  - Given a and b, we can write: $u = a + b, v = a - b$
  - Then to find ab, we just look up the squares and subtract…
  - This is much better since the table now has 2^{n+1}-2 entries
  - Eg consider n=7-bits, then the table is $\{u^2 : u=1..254\}$
    - If a = 17 and b = 18, u = 35, v = 1 so $u^2 - v^2 = 1224$, dividing by 4 (shift!) we get 306
    - Total cost is: 1 add, 2 sub, 2 lookups and 2 shifts…

    (this idea was used by Mattel, Atari, Nintendo…)

Multiply – still more

- Often, you need double precision multiply even if your machine supports multiply
  - Is there something better than 4 single-precision multiplies to get double precision?
  - ***Yes*** – can do it in 3 multiplies:

$$u = 2^n u_1 + u_2 \,, v = 2^n v_1 + v_2$$

$$uv = (2^{2n} + 2^n)u_1 v_1 + 2^n (u_1 - u_0)(v_0 - v_1) + (2^n + 1)u_0 v_0$$

  - Note that you only have to multiply u_1v_1, $(u_1\text{-}u_0)(v_0\text{-}v_1)$ and u_0v_0

  Every thing else is just add, sub and shift –

- More importantly, this trick can be done recursively
  - Leads to $O(n^{1.58})$ multiply for large numbers

Multiply – the (nearly) final word

- The fastest known algorithm for multiply is based on FFT! (Schönhage and Strassen '71)
  - The tough problem in multiply is dealing with the partial products which are of the form: $u_r v_0 + u_{r-1} v_1 + u_{r-2} v_2 + ..$
  - This is the form of a convolution – but we have a neat trick for convolution…

$$f * g = F^{-1}\{F\{g\} \cdot F\{h\}\}$$

  - This trick reduces the cost of the partials to O(n ln(n))
    - Overall runtime is O(n ln(n) ln(ln(n)))
  - Practically, the trick wins for n>200 bits…
    - 2007 Furer showed O(n ln(n) $2^{\lg^*(n)}$) – this wins for n>10^{10} bits..

Division

- There are hundreds of ‘fast’ division algorithms but none are particularly fast
  - Often can re-scale computations so that divisions are reduced to shift – need to pre-calculate constants.
- Newton’s Method can be used to find 1/v
  - Start with small table 1/v given v. (Here we assume v>1)
  - Let x_0 be the first guess
  - Iterate: $x_{n+1} = 2x_n - vx_n^2$
  - Note: if

$$x_n = (1/v) + \varepsilon, x_{n+1} = 2/v + 2\varepsilon - (1/v) - (2\varepsilon) - v\varepsilon^2 = 1/v - v\varepsilon^2$$

  - E.g. v=7, x_0=0.1 x_1=0.13 x_2=0.1417 x_3=0.142848 x_4=0.1428571422 – doubles number of bits each iteration
  - Win here is if table is large enough to get 2 or more digits

Polynomial evaluation

- When represented in monomial basis, polynomial

$$p_{n-1}(t) = x_1 + x_2 t + \cdots + x_n t^{n-1}$$

can be evaluated efficiently using *Horner's nested evaluation* scheme

$$p_{n-1}(t) = x_1 + t(x_2 + t(x_3 + t(\cdots(x_{n-1} + t x_n)\cdots)))$$

which requires only n additions and n multiplications

- For example,

$$1 - 4t + 5t^2 - 2t^3 + 3t^4 = 1 + t(-4 + t(5 + t(-2 + 3t)))$$

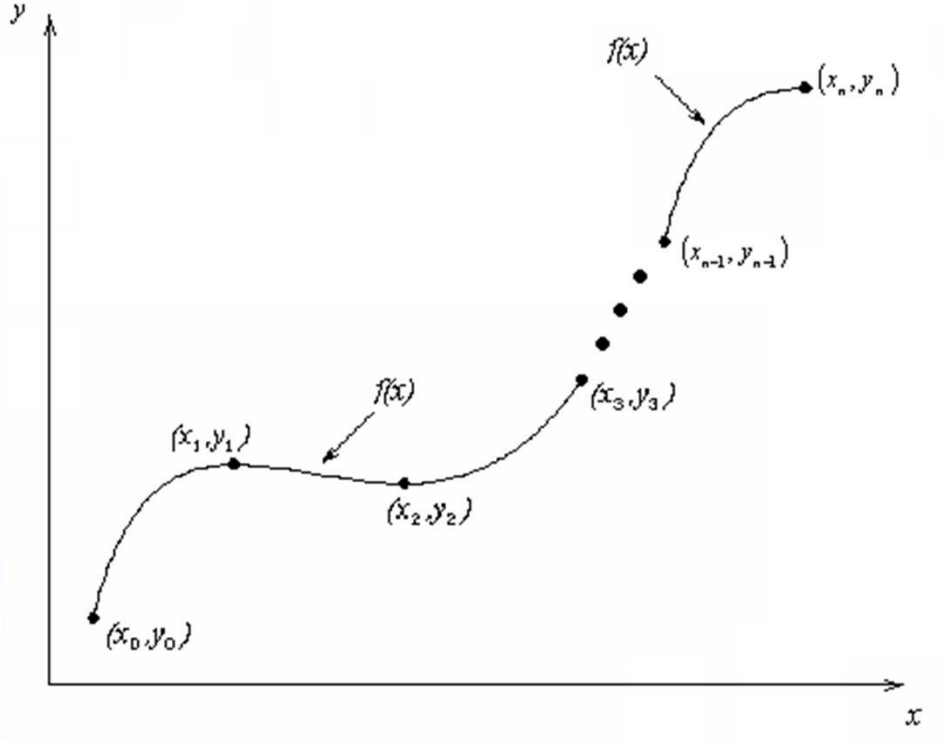
- Other manipulations of interpolating polynomial, such as differentiation or integration, are also relatively easy with monomial basis representation

Interpolation by Polynomials

- Function values listed in short table – how to approximate values not in the table?
- Two basic strategies
  - Fit all points into a (big?) polynomial
    - Lagrange and Newton Interpolation
    - Stability issues as degree increases
  - Fit subsets of points into several overlapping polynomials (splines)
    - Simpler to bound deviations since get new polynomial each segment
    - 1^{st} order is just linear interpolation
    - Higher order allows matching of slopes of splines at nodes
    - Hermite Interpolation matches values and derivatives at each node
- Sometimes polynomials are a poor choice—
  - Asymptotes and Meromorphic functions
  - Rational fits (quotients of polynomials) are good choice
  - Leads to non-linear fitting equations

What is Interpolation ?

Given (x_0,y_0), (x_1,y_1), ... (x_n,y_n), find the value of 'y' at a value of 'x' that is not given.



Interpolation

- Basic interpolation problem: for given data

$$(t_1, y_1),\ (t_2, y_2),\ \ldots\ (t_m, y_m) \quad \text{with} \quad t_1 < t_2 < \cdots < t_m$$

determine function $f : \mathbb{R} \to \mathbb{R}$ such that

$$f(t_i) = y_i, \quad i = 1, \ldots, m$$

- f is *interpolating function*, or *interpolant*, for given data
- Additional data might be prescribed, such as slope of interpolant at given points
- Additional constraints might be imposed, such as smoothness, monotonicity, or convexity of interpolant
- f could be function of more than one variable, but we will consider only one-dimensional case

Basis functions

- Family of functions for interpolating given data points is spanned by set of *basis functions* $\phi_1(t), \ldots, \phi_n(t)$
- Interpolating function f is chosen as linear combination of basis functions,

$$f(t) = \sum_{j=1}^{n} x_j \phi_j(t)$$

- Requiring f to interpolate data (t_i, y_i) means

$$f(t_i) = \sum_{j=1}^{n} x_j \phi_j(t_i) = y_i, \quad i = 1, \ldots, m$$

which is system of linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{y}$ for n-vector $\boldsymbol{x}$ of parameters x_j, where entries of $m \times n$ matrix $\boldsymbol{A}$ are given by $a_{ij} = \phi_j(t_i)$

Polynomial interpolation

- Simplest and most common type of interpolation uses polynomials
- Unique polynomial of degree at most n − 1 passes through n data points (t_i, y_i), i = 1, . . . , n, where t_i are distinct

- *Monomial basis functions*

$$\phi_j(t) = t^{j-1}, \quad j = 1, \ldots, n$$

give interpolating polynomial of form

$$p_{n-1}(t) = x_1 + x_2 t + \cdots + x_n t^{n-1}$$

with coefficients x given by $n \times n$ linear system

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} 1 & t_1 & \cdots & t_1^{n-1} \\ 1 & t_2 & \cdots & t_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & \cdots & t_n^{n-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \boldsymbol{y}$$

- Matrix of this form is called *Vandermonde matrix*

Example

- Determine polynomial of degree two interpolating three data points $(-2, -27), (0, -1), (1, 0)$
- Using monomial basis, linear system is

$$\boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \boldsymbol{y}$$

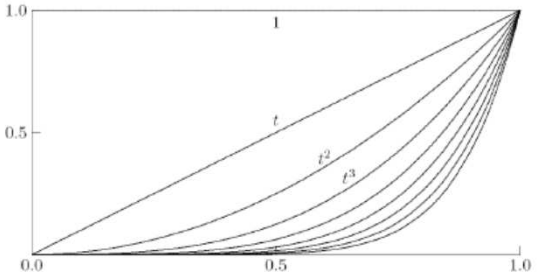
- For these particular data, system is

$$\begin{bmatrix} 1 & -2 & 4 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -27 \\ -1 \\ 0 \end{bmatrix}$$

whose solution is $x = \begin{bmatrix} -1 & 5 & -4 \end{bmatrix}^T$, so interpolating polynomial is

$$p_2(t) = -1 + 5t - 4t^2$$

$O(n^3)$ operations to solve linear system



Conditioning

- For monomial basis, matrix A is increasingly ill-conditioned as degree increases
- Ill-conditioning does not prevent fitting data points well, since residual for linear system solution will be small
- It means that values of coefficients are poorly determined
- Change of basis still gives same interpolating polynomial for given data, but representation of polynomial will be different

- Conditioning with monomial basis can be improved by shifting and scaling independent variable t

$$\phi_j(t) = \left(\frac{t-c}{d}\right)^{j-1}$$

where, $c = (t_1 + t_n)/2$ is midpoint and $d = (t_n - t_1)/2$ is half of range of data

Still not well-conditioned,
Looking for better alternative

Lagrange interpolation

- For given set of data points (t_i, y_i), $i = 1, \ldots, n$, *Lagrange basis functions* are defined by

$$\ell_j(t) = \prod_{k=1, k\neq j}^{n} (t - t_k) \Big/ \prod_{k=1, k\neq j}^{n} (t_j - t_k), \quad j = 1, \ldots, n$$

Easy to determine, but expensive to evaluate, integrate and differentiate comparing to monomials

- For Lagrange basis,

$$\ell_j(t_i) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}, \quad i, j = 1, \ldots, n$$

so matrix of linear system $\boldsymbol{A}x = y$ is identity matrix

- Thus, Lagrange polynomial interpolating data points (t_i, y_i) is given by

$$p_{n-1}(t) = y_1\ell_1(t) + y_2\ell_2(t) + \cdots + y_n\ell_n(t)$$

Example

- Use Lagrange interpolation to determine interpolating polynomial for three data points $(-2, -27)$, $(0, -1)$, $(1, 0)$
- Lagrange polynomial of degree two interpolating three points (t_1, y_1), (t_2, y_2), (t_3, y_3) is given by $p_2(t) =$

$$y_1 \frac{(t-t_2)(t-t_3)}{(t_1-t_2)(t_1-t_3)} + y_2 \frac{(t-t_1)(t-t_3)}{(t_2-t_1)(t_2-t_3)} + y_3 \frac{(t-t_1)(t-t_2)}{(t_3-t_1)(t_3-t_2)}$$

- For these particular data, this becomes

$$p_2(t) = -27 \frac{t(t-1)}{(-2)(-2-1)} + (-1) \frac{(t+2)(t-1)}{(2)(-1)}$$

Piecewise interpolation (Splines)

- Fitting single polynomial to large number of data points is likely to yield unsatisfactory oscillating behavior in interpolant
- Piecewise polynomials provide alternative to practical and theoretical difficulties with high-degree polynomial interpolation. Main advantage of piecewise polynomial interpolation is that large number of data points can be fit with low-degree polynomials
- In piecewise interpolation of given data points (t_i, y_i), different functions are used in each subinterval $[t_i, t_{i+1}]$
- Abscissas t_i are called knots or breakpoints, at which interpolant changes from one function to another
- Simplest example is piecewise linear interpolation, in which successive pairs of data points are connected by straight lines
- Although piecewise interpolation eliminates excessive oscillation and nonconvergence, it may sacrifice smoothness of interpolating function
- We have many degrees of freedom in choosing piecewise polynomial interpolant, however, which can be exploited to obtain smooth interpolating function despite its piecewise nature

Why Splines ?

$$f(x) = \frac{1}{1+25x^2}$$

Table : Six equidistantly spaced points in [-1, 1]

x	$y = \frac{1}{1+25x^2}$
-1.0	0.038461
-0.6	0.1
-0.2	0.5
0.2	0.5
0.6	0.1
1.0	0.038461

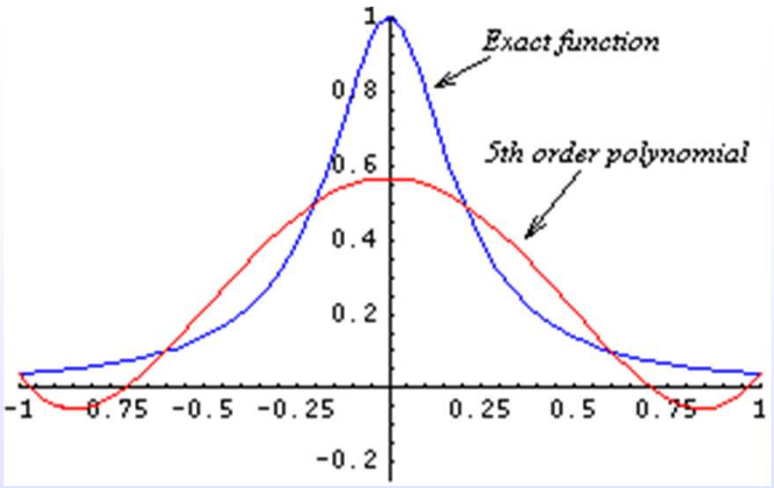


Figure : 5th order polynomial vs. exact function

Why Splines ?

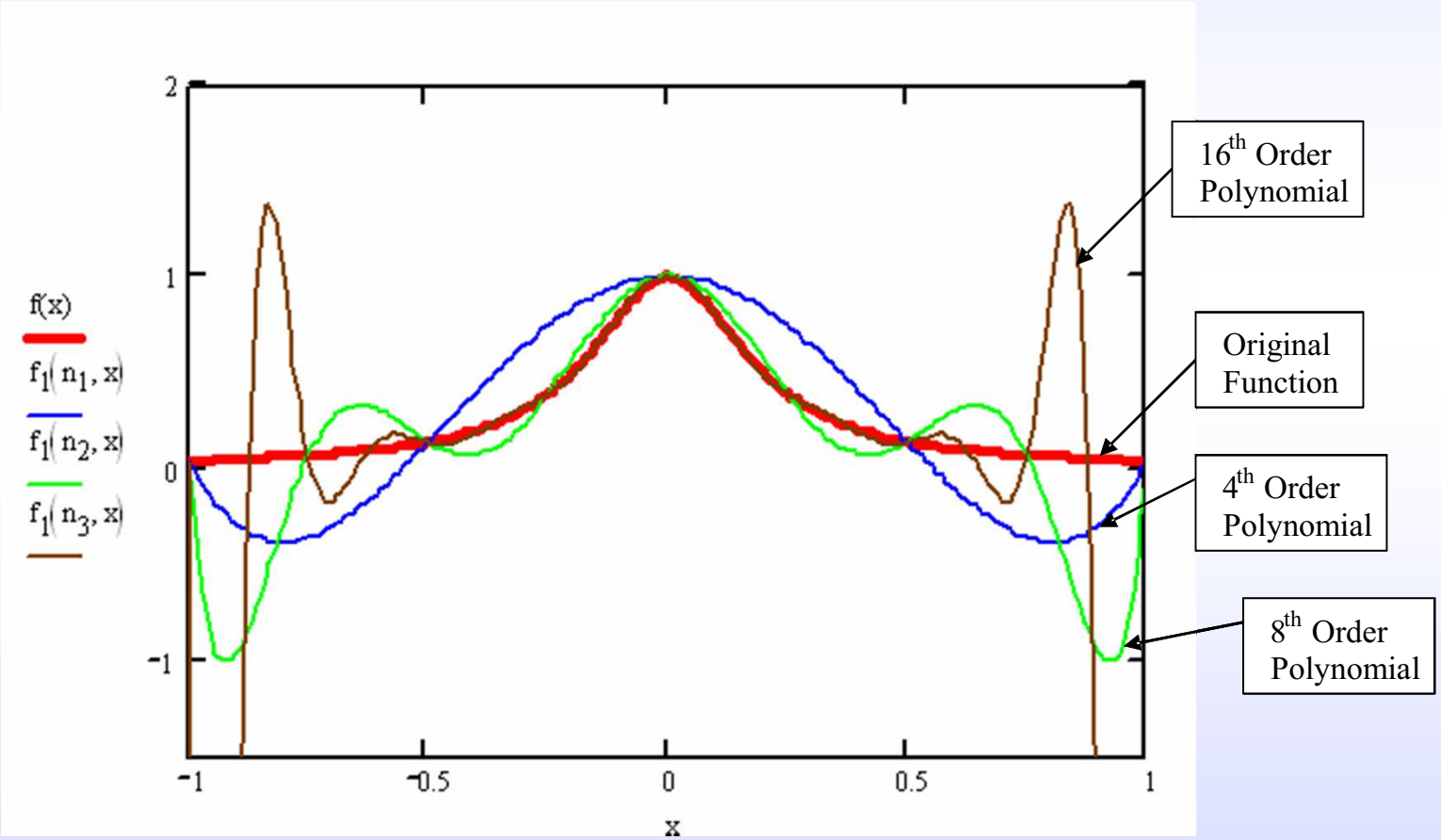
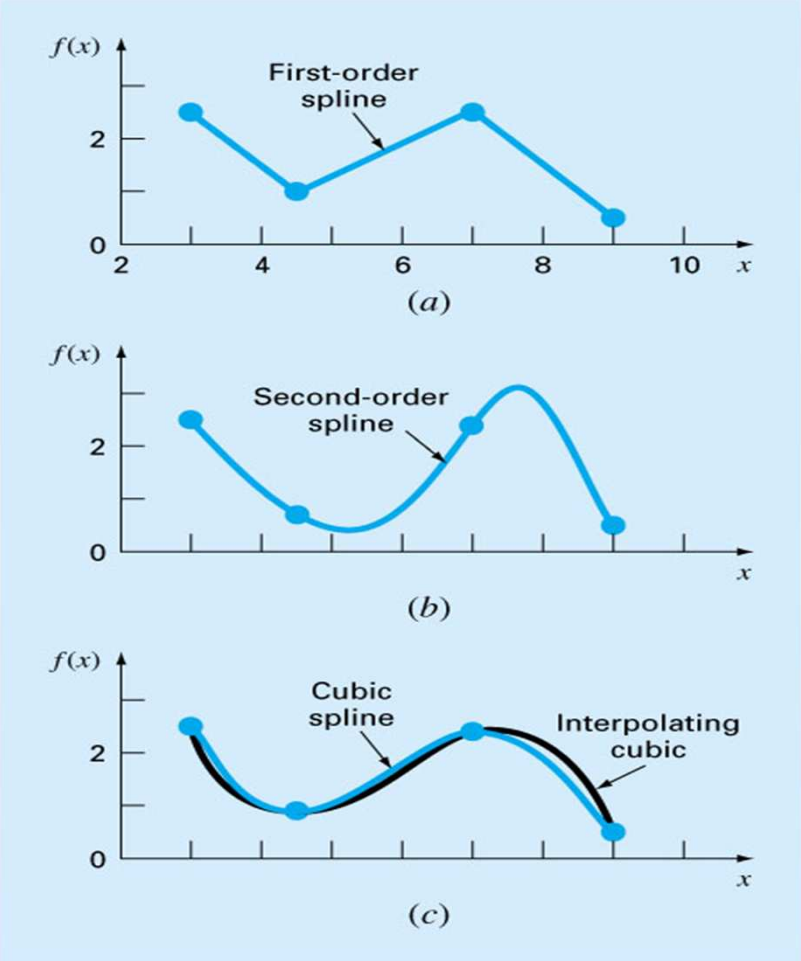


Figure : Higher order polynomial interpolation is dangerous

Spline orders

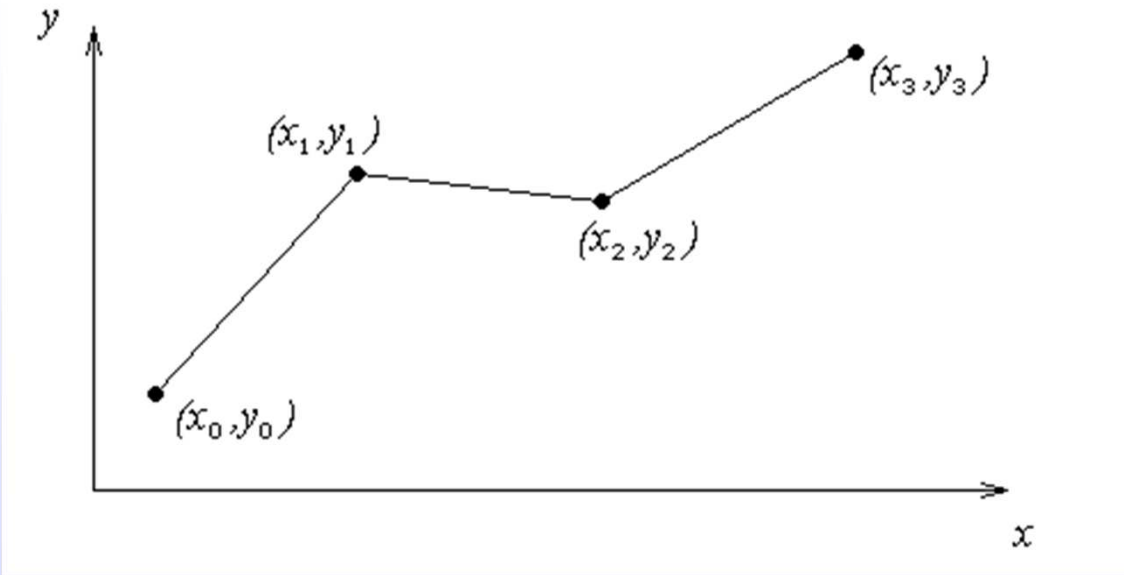
- Linear spline
  - Derivatives are not continuous
  - Not smooth
- Quadratic spline
  - Continuous 1st derivatives
- Cubic spline
  - Continuous 1st & 2nd derivatives
  - Smoother



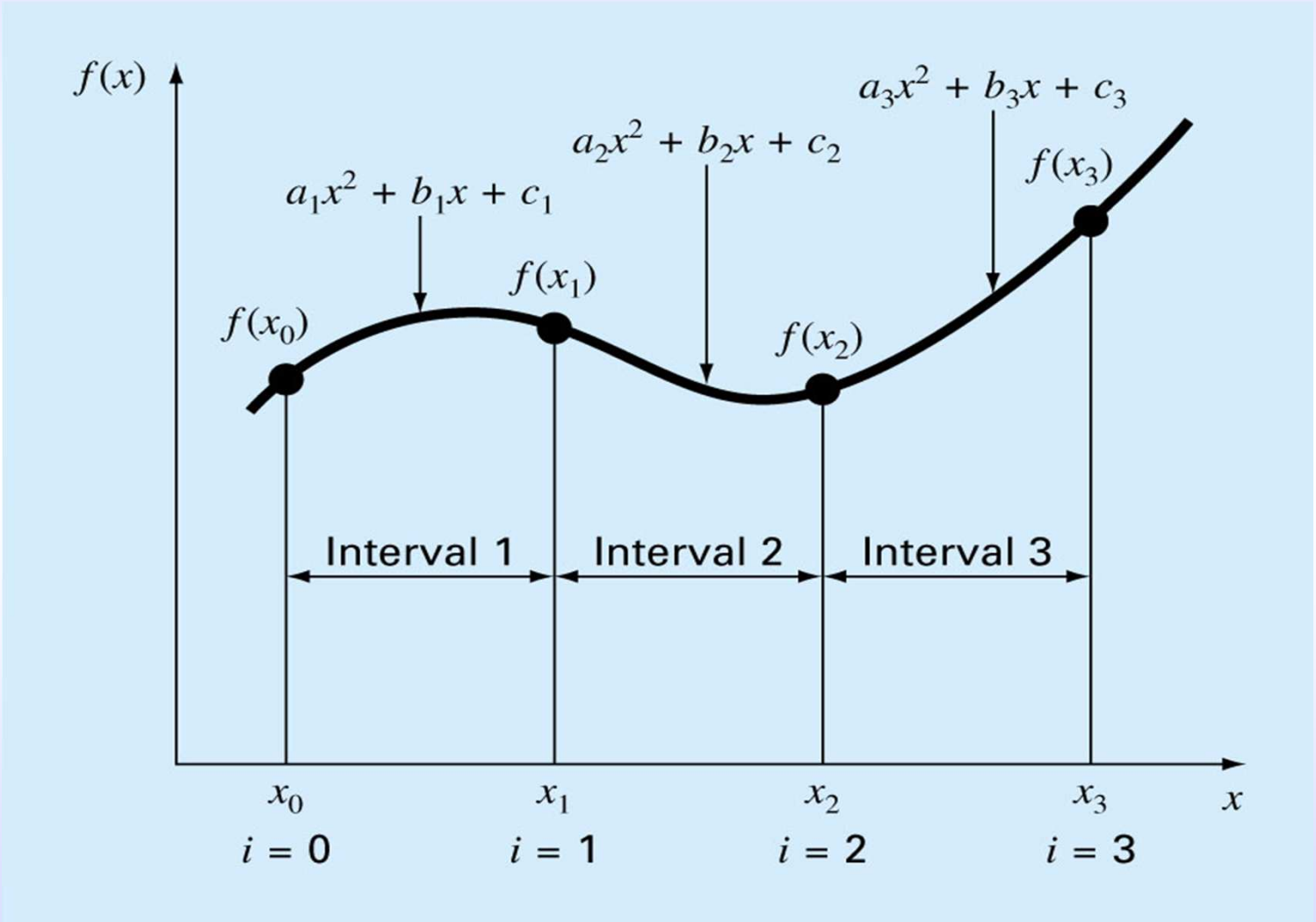
Linear Interpolation

Given $(x_0, y_0), (x_1, y_1), \ldots\ldots, (x_{n-1}, y_{n-1}) (x_n, y_n)$, fit linear splines to the data. This simply involves forming the consecutive data through straight lines. So if the above data is given in an ascending order, the linear splines are given by $(y_i = f(x_i))$

Figure : Linear splines



Quadratic Spline



Cubic Spline

- Spline of Degree 3
- A function S is called a spline of degree 3 if
  - The domain of S is an interval $[a, b]$.
  - S, S' and S'' are continuous functions on $[a, b]$.
  - There are points t_i (called knots) such that

    $a = t_0 < t_1 < \ldots < t_n = b$ and Q is a polynomial of degree at most 3 on each subinterval $[t_i, t_{i+1}]$.

Cubic Spline ($4n$ conditions)

1. Interpolating conditions ($2n$ conditoins).
2. Continuous 1st derivatives (n-1 conditions)
   - The 1st derivatives at the interior knots must be equal.
3. Continuous 2nd derivatives (n-1 conditions)
   - The 2nd derivatives at the interior knots must be equal.
4. Assume the 2nd derivatives at the end points are zero (2 conditions).
   - This condition makes the spline a "natural spline".

Hermite cubic Interpolant

Hermite cubic interpolant: piecewise cubic polynomial interpolant with continuous first derivative

- Piecewise cubic polynomial with n knots has $4(n-1)$ parameters to be determined
- Requiring that it interpolate given data gives $2(n-1)$ equations
- Requiring that it have one continuous derivative gives $n-2$ additional equations, or total of $3n-4$, which still leaves n free parameters
- Thus, Hermite cubic interpolant is not unique, and remaining free parameters can be chosen so that result satisfies additional constraints

Spline example

- Determine natural cubic spline interpolating three data points (t_i, y_i), $i = 1, 2, 3$
- Required interpolant is piecewise cubic function defined by separate cubic polynomials in each of two intervals $[t_1, t_2]$ and $[t_2, t_3]$
- Denote these two polynomials by

$$p_1(t) = \alpha_1 + \alpha_2 t + \alpha_3 t^2 + \alpha_4 t^3$$

$$p_2(t) = \beta_1 + \beta_2 t + \beta_3 t^2 + \beta_4 t^3$$

- Eight parameters are to be determined, so we need eight equations

Example

- Requiring first cubic to interpolate data at end points of first interval $[t_1, t_2]$ gives two equations

$$\alpha_1 + \alpha_2 t_1 + \alpha_3 t_1^2 + \alpha_4 t_1^3 = y_1$$

$$\alpha_1 + \alpha_2 t_2 + \alpha_3 t_2^2 + \alpha_4 t_2^3 = y_2$$

- Requiring second cubic to interpolate data at end points of second interval $[t_2, t_3]$ gives two equations

$$\beta_1 + \beta_2 t_2 + \beta_3 t_2^2 + \beta_4 t_2^3 = y_2$$

$$\beta_1 + \beta_2 t_3 + \beta_3 t_3^2 + \beta_4 t_3^3 = y_3$$

- Requiring first derivative of interpolant to be continuous at t_2 gives equation

$$\alpha_2 + 2\alpha_3 t_2 + 3\alpha_4 t_2^2 = \beta_2 + 2\beta_3 t_2 + 3\beta_4 t_2^2$$

Example

- Requiring second derivative of interpolant function to be continuous at t_2 gives equation

$$2\alpha_3 + 6\alpha_4 t_2 = 2\beta_3 + 6\beta_4 t_2$$

- Finally, by definition natural spline has second derivative equal to zero at endpoints, which gives two equations

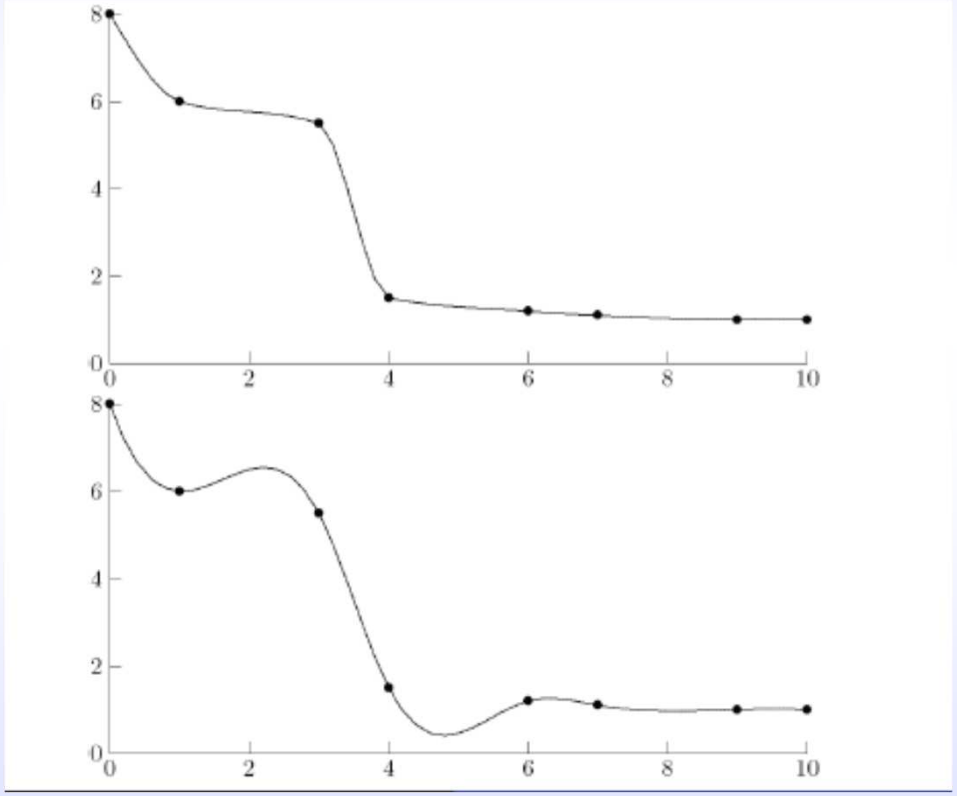
$$2\alpha_3 + 6\alpha_4 t_1 = 0$$

$$2\beta_3 + 6\beta_4 t_3 = 0$$

- When particular data values are substituted for t_i and y_i, system of eight linear equations can be solved for eight unknown parameters α_i and β_i

Hermite vs. spline

- Choice between Hermite cubic and spline interpolation depends on data to be fit
- If smoothness is of paramount importance, then spline interpolation wins
- Hermite cubic interpolant allows flexibility to preserve monotonicity if original data are monotonic



Function Representation

- Put together minimal table, polynomial splines/fit
- Discrete error model
- Tradeoff table size/run time vs. accuracy

Embedded Processor Function Evaluation

--Dong-U Lee/UCLA 2006

- Approximate functions via polynomials
- Minimize resources for given target precision
- Processor fixed-point arithmetic
- Minimal number of bits to each signal in the data path
- Emulate operations larger than processor word-length

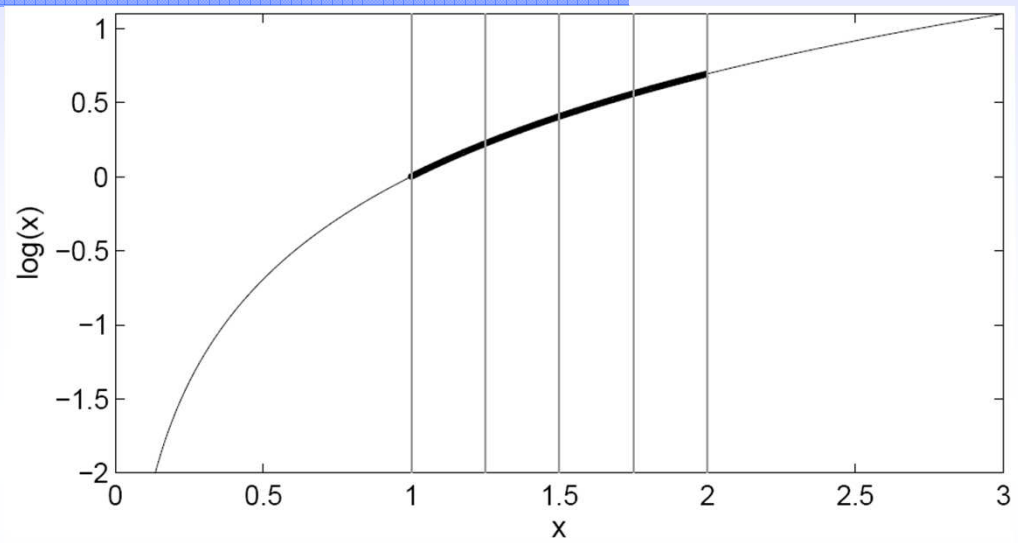
Function Evaluation

- Typically in three steps
  - (1) reduce the input interval [a,b] to a smaller interval [a',b']
  - (2) function approximation on the range-reduced interval
  - (3) expand the result back to the original range
- Evaluation of log(x):

$$\log(M_x \times 2^{E_x}) \quad = \quad \log(M_x) + E_x \times \log(2)$$

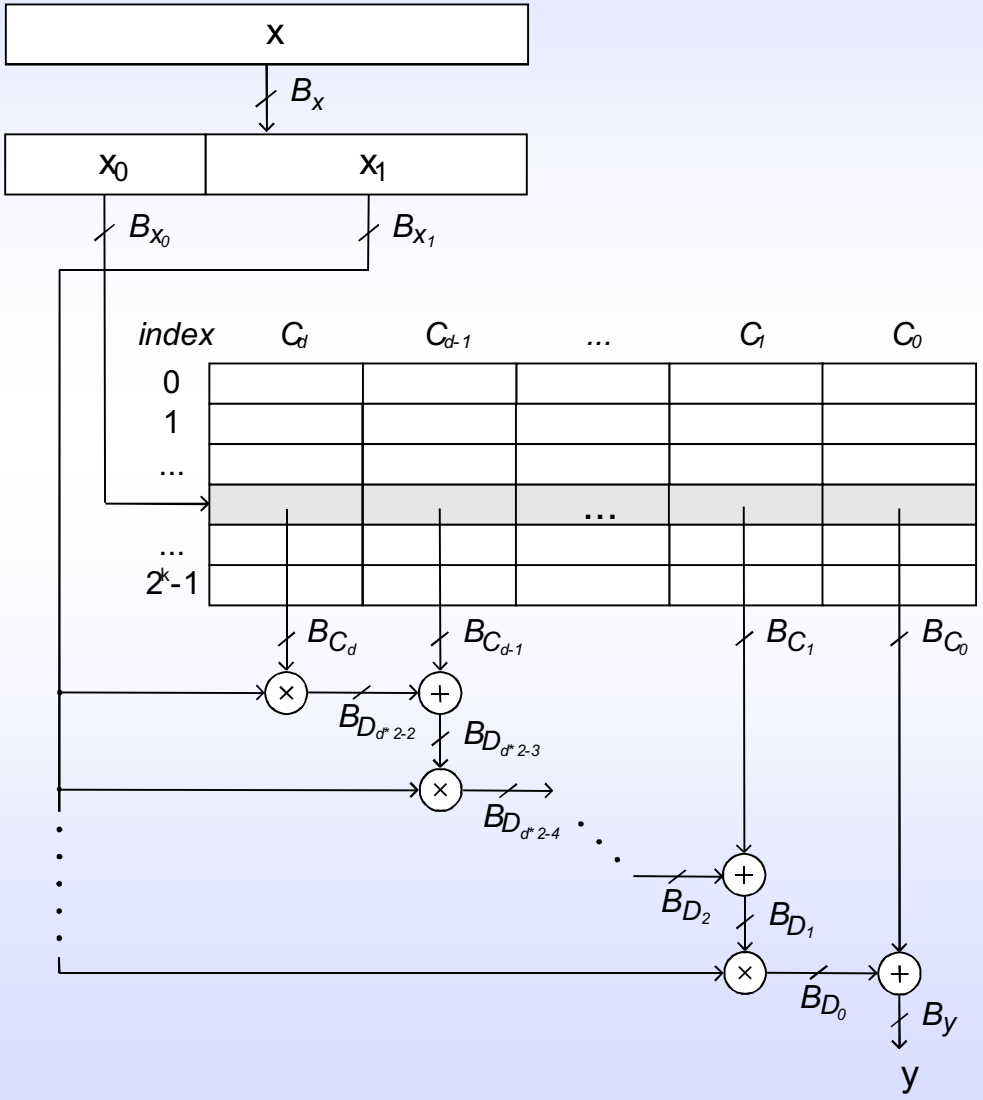
where M_x is the mantissa over the [1,2) and E_x is the exponent of x

Polynomial Approximations



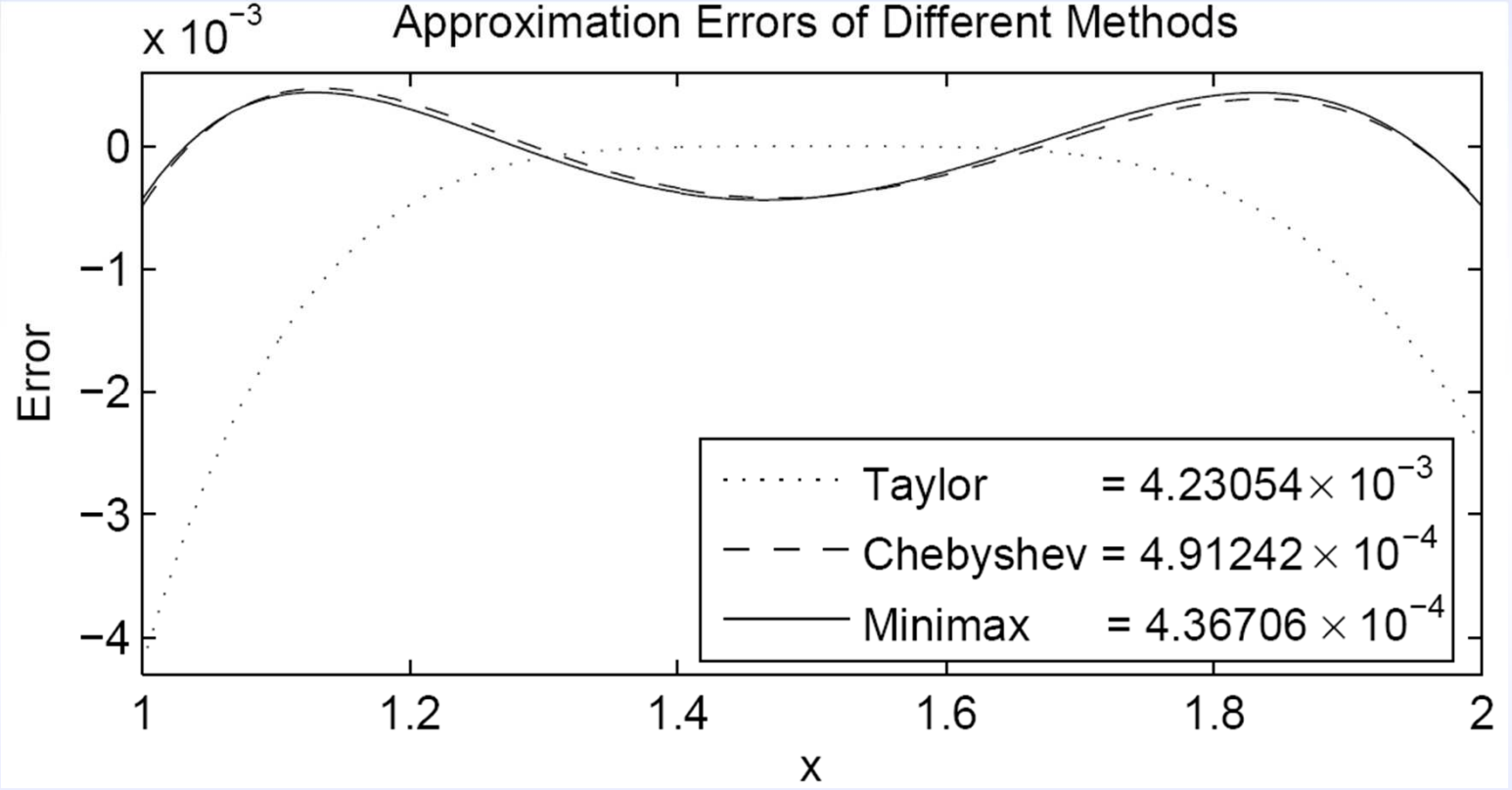
- Single polynomial
  - Whole interval approximated with a single polynomial
  - Increase polynomial degree until the error requirement is met
- Splines (piecewise polynomials)
  - Partition interval into multiple segments: fit polynomial for each segment
  - Given polynomial degree, increase the number of segments until the error requirement is met

Computation Flow



- Input interval is split into 2^{Bx_0} equally sized segments
- Leading B_{x0} bits serve as coefficient table index
- Coefficients computed to
- Determine minimal bit-widths → minimize execution time
- x_1 used for polynomial arithmetic normalized over [0,1)

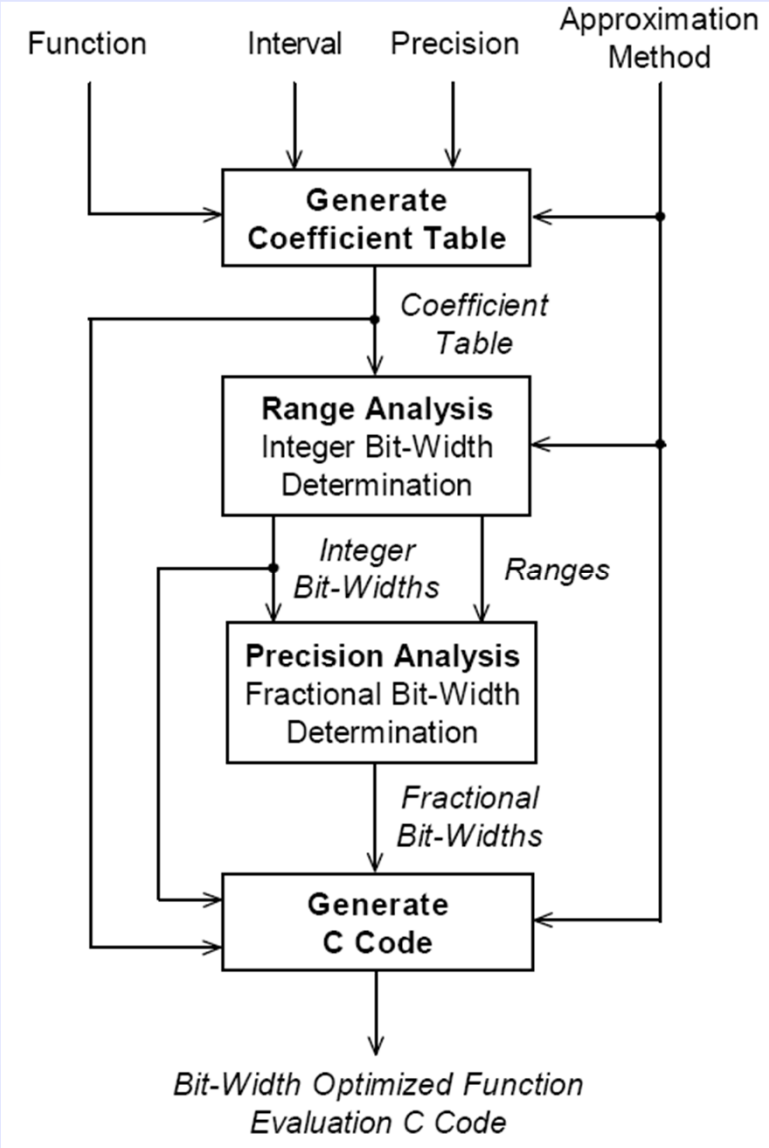
Approximation Methods



- Degree-3 Taylor, Chebyshev, and minimax approximations of log(x)

Design Flow Overview

- Written in MATLAB
- Approximation methods
  - Single Polynomial
  - Degree-*d* splines
- Range analysis
  - Analytic method based on roots of the derivative
- Precision analysis
  - Simulated annealing on analytic error expressions

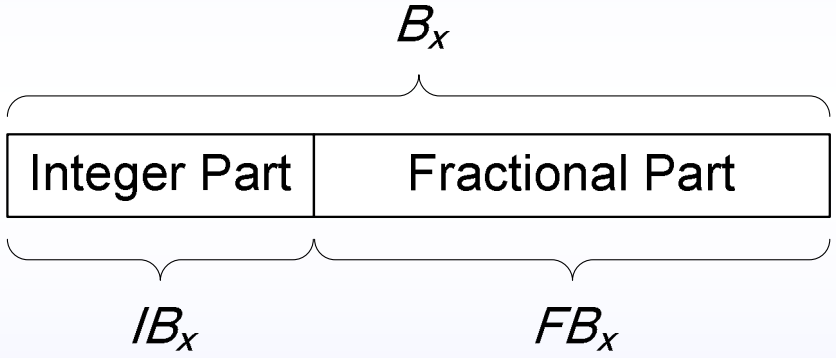


Error Sources

- Three main error sources:
  1. Inherent error E_∞ due to approximating functions
  2. Quantization error E_Q due to finite precision effects
  3. Final output rounding step, which can cause a maximum of 0.5 ulp
- To achieve 1 ulp accuracy at the output, $E_\infty + E_Q \leq 0.5$ ulp
  - Large E_∞
    - Polynomial degree reduction (single polynomial) or required number of segments reduction (splines)
    - However, leads to small E_Q, leading to large bit-widths
  - Good balance: allocate a maximum of 0.3 ulp for E_∞ and the rest for E_Q

Range Analysis

- Inspect dynamic range of each signal and compute required number of integer bits
- Two's complement assumed, for a signal *x*:



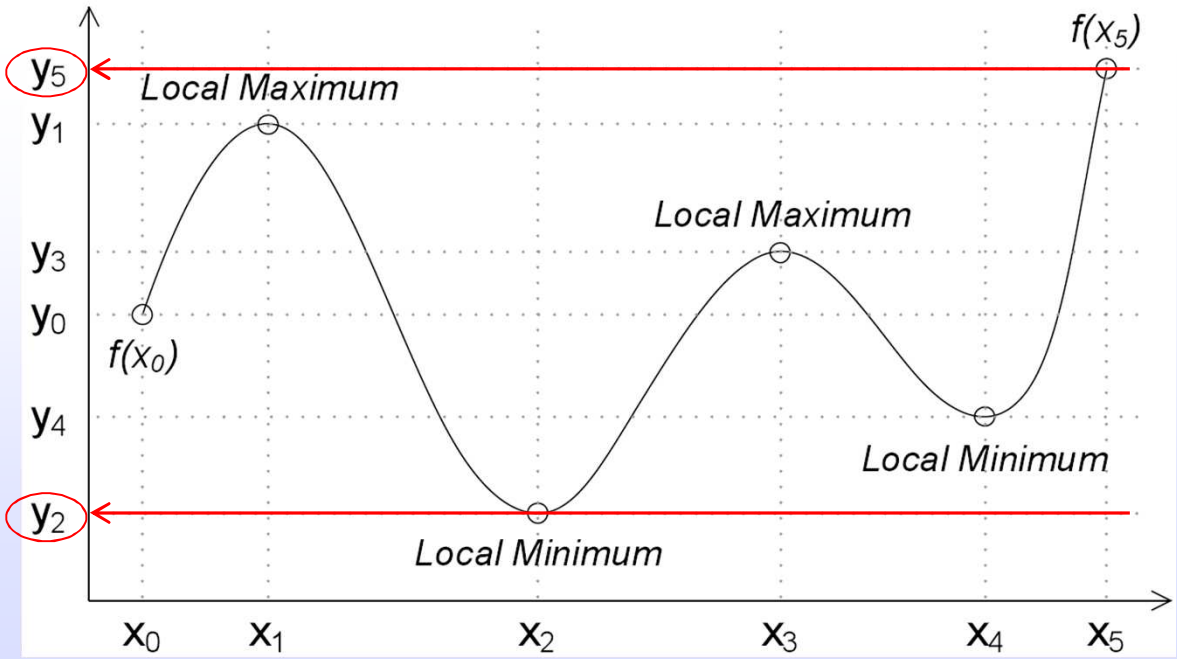
- For a range $x=[x_{min},x_{max}]$, IB_x is given by

$$IB_x = \lceil \log_2(\max(|x_{min}|, |x_{max}|)) \rceil + \alpha$$

$$\text{where } \alpha = \begin{cases} 1, & \text{mod}(\log_2(x_{max}), 1) \neq 0 \\ 2, & \text{mod}(\log_2(x_{max}), 1) = 0 \end{cases}$$

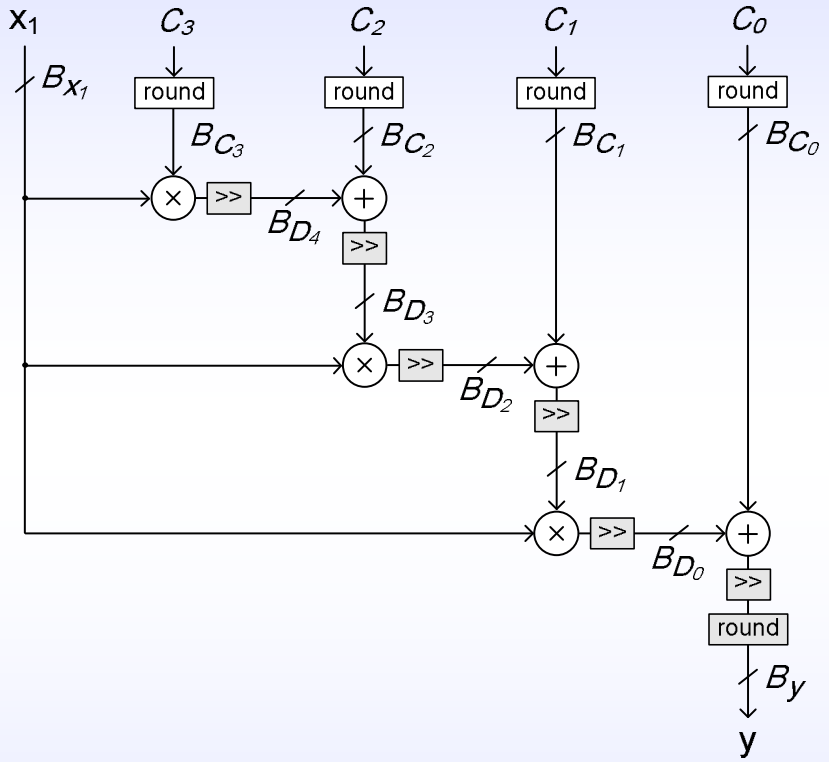
Range Determination

- Examine local minima, local maxima, and minimum and maximum input values at each signal
- Works for designs with differentiable signals, which is the case for polynomials



range = $[y_2, y_5]$

Range Analysis Example



- Degree-3 polynomial approximation to log(x)
- Able to compute exact ranges
- *IB* can be negative as shown for C_3, C_0, and D_4: leading zeros in the fractional part

Coefficient	C_3	C_2	C_1	C_0
Value	0.11006	-0.40048	0.98362	0.00044
IB	-2	0	1	-10

Signal	Range	IB
D_4	[0.00000,0.11006]	-2
D_3	[-0.40048,-0.29042]	0
D_2	[-0.29042,0.00000]	0
D_1	[0.69320,0.98362]	1
D_0	[0.00000,0.69320]	1
y	[0.00044,0.69363]	1

Precision Analysis

- Determine minimal *FBs* of all signals while meeting error constraint at output
- Quantization methods
  - Truncation: 2^{-FB} (1 ulp) maximum error
  - Round-to-nearest: 2^{-FB-1} (0.5 ulp) maximum error
- To achieve 1 ulp accuracy at output, round-to-nearest must be performed at output
- Free to choose either method for internal signals: Although round-to-nearest offers smaller errors, it requires an extra adder, hence truncation is chosen

Error Models of Arithmetic Operators

- Let $\tilde{x}$ be the quantized version and $\epsilon_{\tilde{x}}$ be the error due to quantizing a signal x
- Addition/subtraction

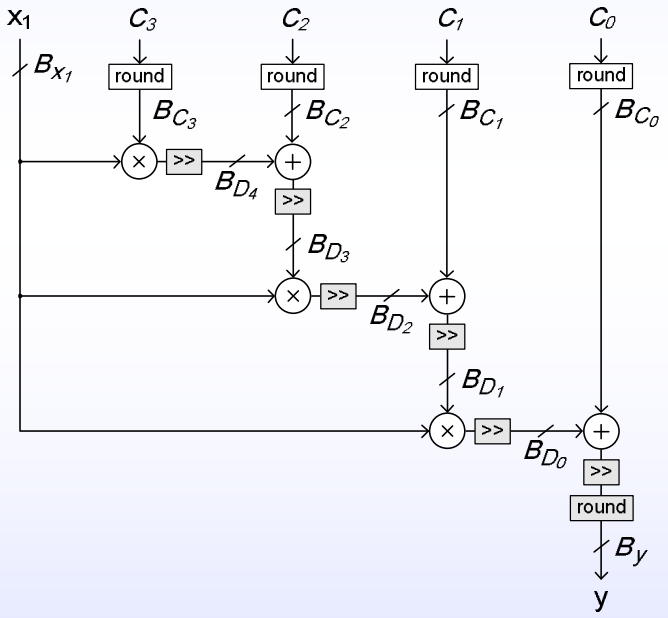
$$\begin{aligned} \tilde{z} &= \tilde{x} \pm \tilde{y} = x \pm y + \epsilon_{\tilde{x}} + \epsilon_{\tilde{y}} + 2^{-FB_{\tilde{z}}} \\ \Rightarrow \epsilon_{\tilde{z}} &= \epsilon_{\tilde{x}} + \epsilon_{\tilde{y}} + 2^{-FB_{\tilde{z}}} \end{aligned}$$

- Multiplication

$$\begin{aligned} \tilde{z} &= \tilde{x}\tilde{y} \\ &= xy + a\epsilon_{\tilde{y}} + b\epsilon_{\tilde{x}} + \epsilon_{\tilde{x}}\epsilon_{\tilde{y}} + 2^{-FB_{\tilde{z}}} \\ \Rightarrow \epsilon_{\tilde{z}} &= x\epsilon_{\tilde{y}} + y\epsilon_{\tilde{x}} + \epsilon_{\tilde{x}}\epsilon_{\tilde{y}} + 2^{-FB_{\tilde{y}}} \end{aligned}$$

Precision Analysis for Polynomials

- Degree-3 polynomial example, assuming that coefficients are rounded to the nearest:



$$
\begin{aligned}
\epsilon_{D_4} &= 2^{-FB_{C_3}-1} + 2^{-FB_{D_4}} \\
\epsilon_{D_3} &= \epsilon_{D_4} + 2^{-FB_{C_2}-1} + 2^{-FB_{D_3}} \\
\epsilon_{D_2} &= \epsilon_{D_3} + 2^{-FB_{D_2}} \\
\epsilon_{D_1} &= \epsilon_{D_2} + 2^{-FB_{C_1}-1} + 2^{-FB_{D_1}} \\
\epsilon_{D_0} &= \epsilon_{D_1} + 2^{-FB_{D_0}} \\
\epsilon_y &= \epsilon_{D_0} + 2^{-FB_{C_0}-1} + 2^{-FB_y-1} + \epsilon_\infty
\end{aligned}
$$

Inherent approximation error

$$
\Rightarrow \epsilon_y = \sum_{i=0}^{3} 2^{-FB_{C_i}-1} + \sum_{i=0}^{4} 2^{-FB_{D_i}} + 2^{-FB_y-1} + \epsilon_\infty
$$

Uniform Fractional Bit-Width

- Obtain 8 fractional bits with 1 ulp (2^{-8}) accuracy

$$\epsilon_y \leq 2^{-FB_y}$$

$$\Rightarrow 2^{-8} \geq \sum_{i=0}^{3} 2^{-FB_{C_i}} + \sum_{i=0}^{4} 2^{1-FB_{D_i}} + 2 \times \epsilon_\infty$$

- Suboptimal but simple solution is the uniform fractional bit-width (UFB)

$$\begin{aligned} 2^{-8} \quad &\geq \quad 2^{2-UFB} + 2^{1-UFB} + 2^{3-UFB} \\ &\quad\quad + 2 \times 4.36706 \times 10^{-4} \end{aligned}$$

$$\Rightarrow UFB \quad = \quad 13 \text{ bits}$$

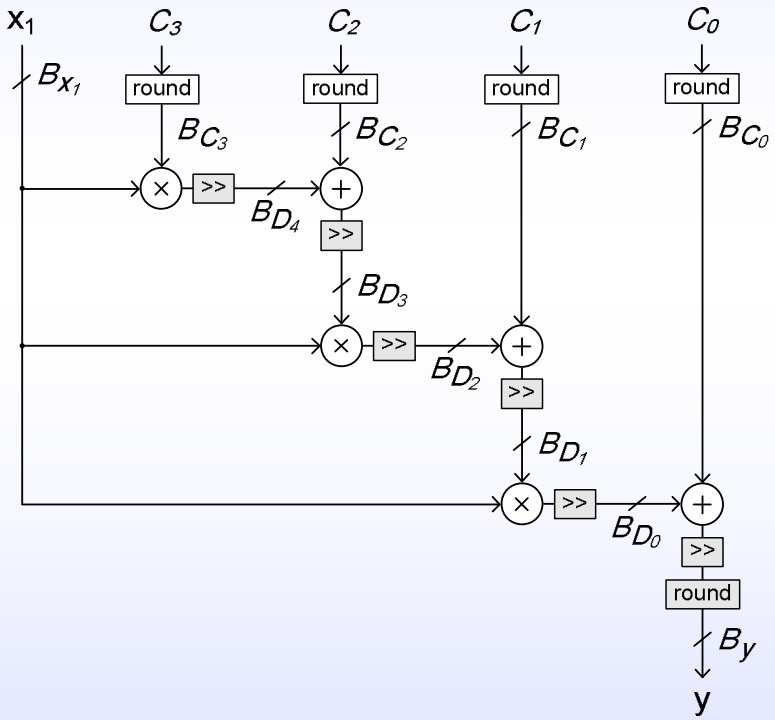
Non-Uniform Fractional Bit-Width

- Let the fractional bit-widths to be different
- Use adaptive simulated annealing (ASA), which allows for faster convergence times than traditional simulated annealing
  - Constraint function: error inequalities
  - Cost function: latency of multiword arithmetic
- Bit-widths must be a multiple of the natural processor word-length *n*

$$FB_x = n - IB_x + i \times n \qquad \text{where } i \in \mathbb{Z}^*$$

- On an 8-bit processor, if signal $IB_x = 1$, then $FB_x = \{ 7, 15, 23, \ldots \}$

Bit-Widths for Degree-3 Example



Shifts for binary point alignment

Signal	8-bit Constrained				C Constrained			
	B	IB	FB	$>>$	B	IB	FB	$>>$
C_3	8	-2	10	-	8	-2	10	-
C_2	16	0	16	-	16	0	16	-
C_1	16	1	15	-	16	1	15	-
C_0	8	-10	18	-	16	-10	26	-
D_4	16	0	16	2	16	0	16	2
D_3	16	0	16	0	16	0	16	0
D_2	16	1	15	9	16	1	15	9
D_1	16	1	15	0	16	1	15	0
D_0	24	6	18	5	32	6	26	-3
y	16	8	8	10	16	8	8	18

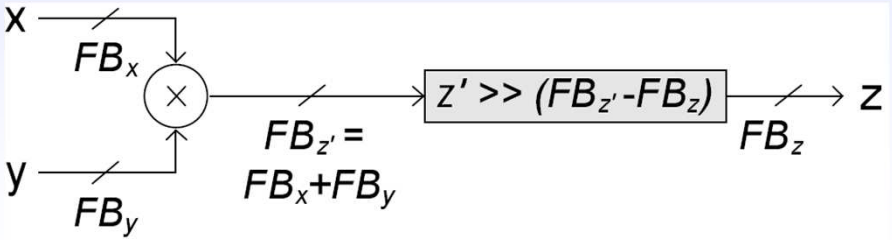
Total Bit-Width

Integer Bit-Width

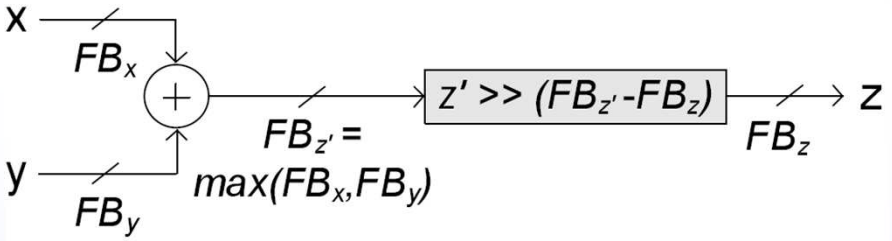
Fractional Bit-Width

Fixed-Point to Integer Mapping

Multiplication



Addition
(binary point of operands must be aligned)



- Fixed-point libraries for the C language do not provide support for negative integer bit-widths
  → emulate fixed-point via integers with shifts

Multiplications in C language

- In standard C, a 16-bit by 16-bit multiplication returns the least significant 16 bits of the full 32-bit result → undesirable since access to the full 32 bits is required
- Solution 1: pad the two operands with 16 leading zeros and perform 32-bit by 32-bit multiplication
- Solution 2: use special C syntax to extract full 32-bit result from 16-bit by 16-bit multiplication

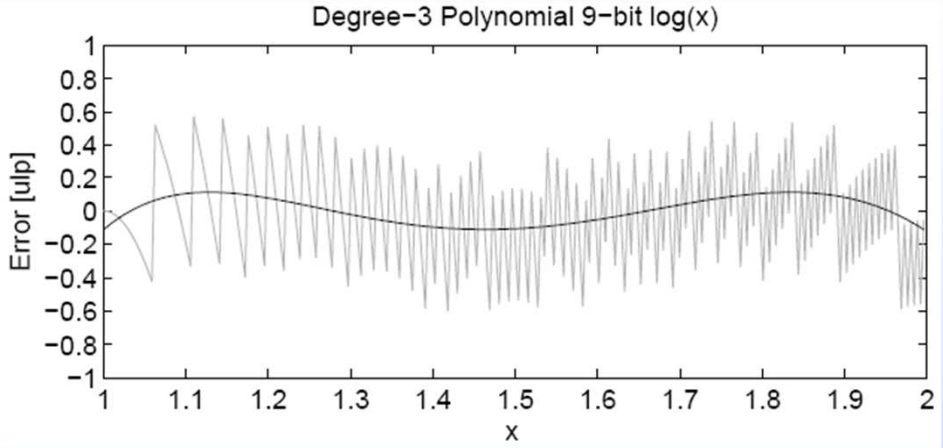
```
int32_t z = (int32_t)(int16_t)x * (int32_t)(int16_t)y
```

More efficient than solution 1 and works on both Atmel and TI compilers

Automatically Generated C Code for Degree-3 Polynomial

```
inline int16_t log_poly3_9bit(int16_t x) {

  static const int8_t  c3 = 113;
  static const int16_t c2 = -26246;
  static const int16_t c1 = 32231;
  static const int16_t c0 = 29307;
  static const int32_t r  = 131072;

  static int16_t d4, d3, d2, d1, y;
  static int32_t d0;

  d4 = (int16_t)(((int32_t)(int16_t)x
       * (int32_t)(int8_t)c3) >> 2);
  d3 = (int16_t)(((int16_t)(int16_t)d4
       + (int16_t)(int16_t)c2) >> 0);
  d2 = (int16_t)(((int32_t)(int16_t)x
       * (int32_t)(int16_t)d3) >> 9);
  d1 = (int16_t)(((int16_t)(int16_t)d2
       + (int16_t)(int16_t)c1) >> 0);
  d0 = (int32_t)(((int32_t)(int16_t)x
       * (int32_t)(int16_t)d1) << 3);
  y = (int16_t)(((int32_t)(int32_t)d0
      + (int32_t)(int32_t)c0
      + (int32_t)(int32_t)r) >> 18);

  return y;

}
```

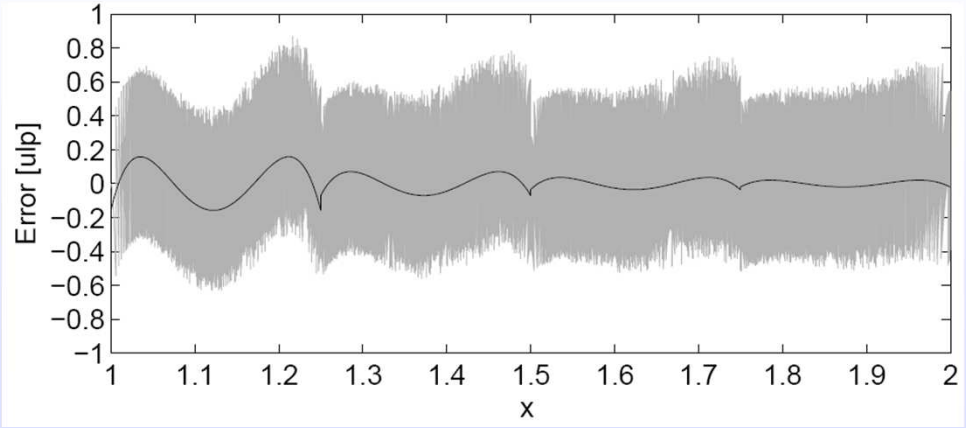
- Casting for controlling multi-word arithmetic (inttypes.h)
- Shifts after each operation for quantization
- *r* is a constant used for final rounding



Automatically Generated C Code for Degree-3 Splines

```
inline int16_t log_splines3_16bit(int16_t x) {

  static const int16_t c3_table[4]
   = { 31031, 16933, 10237, 6655 };
  static const int16_t c2_table[4]
   = { -31870, -20579, -14365, -10589 };
  static const int32_t c1_table[4]
   = { 2146092910, 1717372057, 1431343304, 1226958295 };
  static const int32_t c0_table[4]
   = { 10372, 479201753, 870732054, 1201767086 };
  static const int16_t r = 32768;

  static int8_t  x0;
  static int16_t d4, d3, d1, c3, c2, x1, y;
  static int32_t d2, d0, c1, c0;

  x0 = x >> 13;
  x1 = x & 8191;

  c3 = c3_table[x0]; c2 = c2_table[x0];
  c1 = c1_table[x0]; c0 = c0_table[x0];

  d4 = (int16_t)(((int32_t)(int16_t)x1
       * (int32_t)(int16_t)c3) >> 16);
  d3 = (int16_t)(((int16_t)(int16_t)d4
       + (int16_t)(int16_t)c2) >> 0);
  d2 = (int32_t)(((int32_t)(int16_t)x1
       * (int32_t)(int16_t)d3) >> 0);
  d1 = (int16_t)(((int32_t)(int32_t)d2
       + (int32_t)(int32_t)c1) >> 16);
  d0 = (int32_t)(((int32_t)(int16_t)x1
       * (int32_t)(int16_t)d1) << 1);
  y = (int16_t)(((int32_t)(int32_t)d0
      + (int32_t)(int32_t)c0
      + (int32_t)(int16_t)r) >> 16);

  return y;

}
```

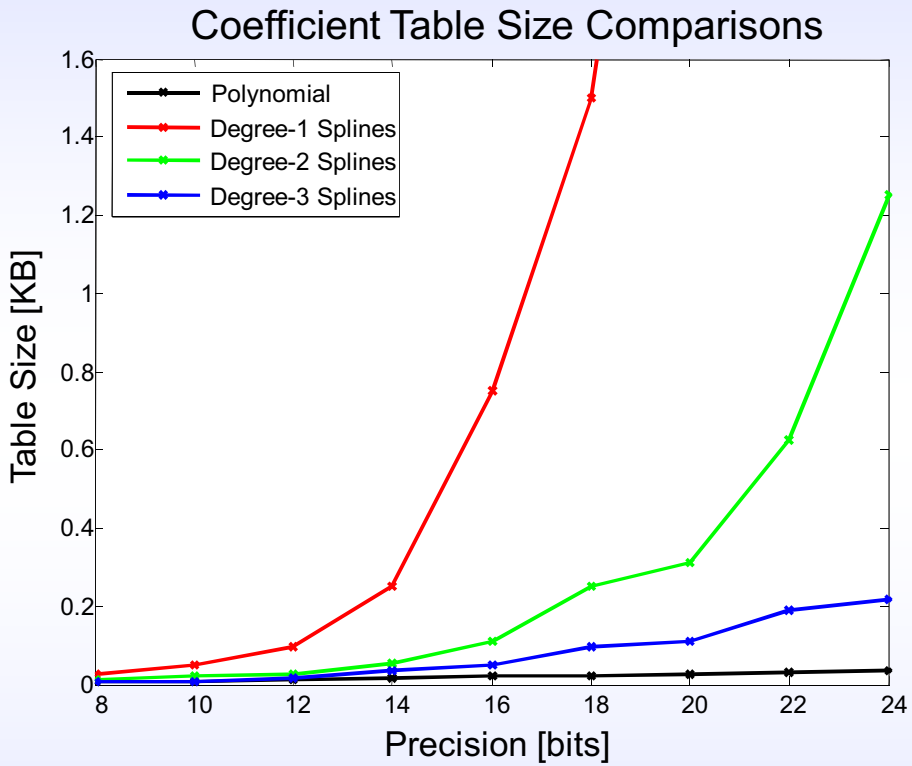
- Accurate to 15 fractional bits (2^{-15})
- 4 segments used
- 2 leading bits of *x* of the table index
- Over 90% are exactly rounded less than ½ ulp error



Experimental Validation

- Two commonly-used embedded processors
  - Atmel ATmega 128 8-bit MCU
    - Single ALU & a hardware multiplier
    - Instructions execute in one cycle except multiplier (2 cycles)
    - 4 KB RAM
    - Atmel AVR Studio 4.12 for cycle-accurate simulation
  - TI TMS320C64x 16-bit fixed-point DSP
    - VLIW architecture with six ALUs & two hardware multipliers
    - ALU: multiple additions/subtractions/shifts per cycle
    - Multiplier: 2x 16b-by-16b / 4x 8b-by-8b per cycle
    - 32 KB L1 & 2048 KB L2 cache
    - TI Code Composer Studio 3.1 for cycle-accurate simulation
- Same C code used for both platforms

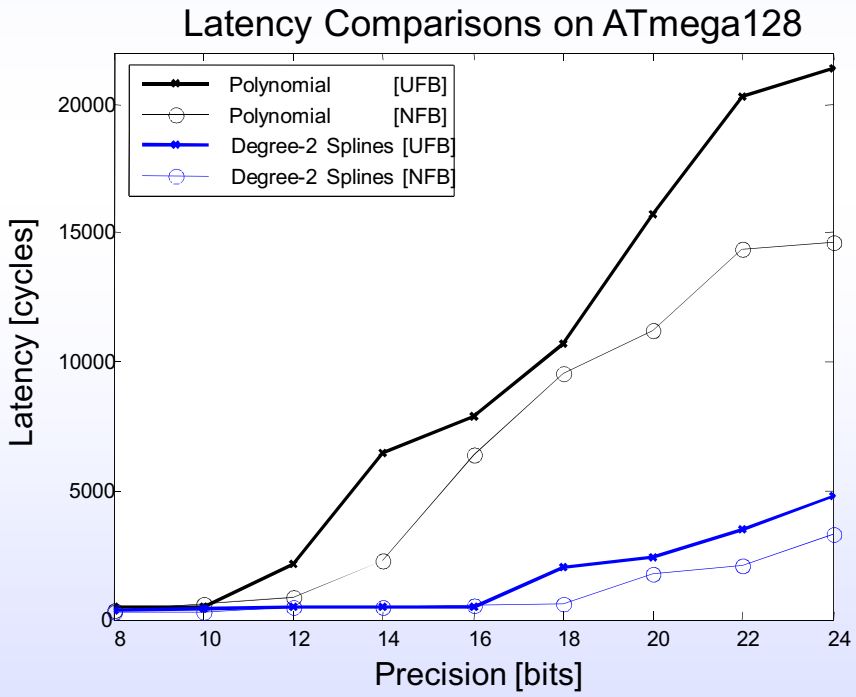
Table Size Variation

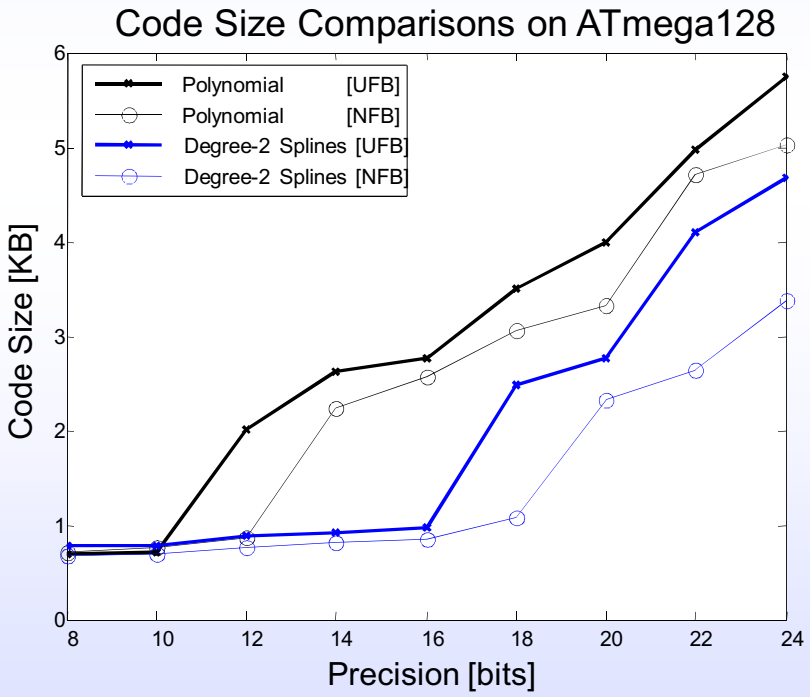


- Single polynomial approximation shows little area variation
- Rapid growth with low degree splines due to increasing number of segments

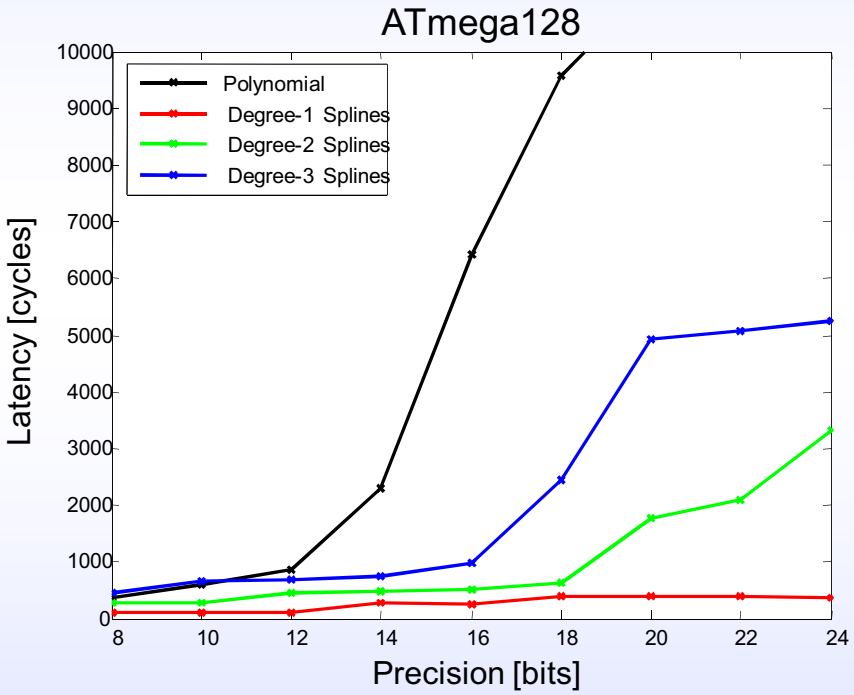
NFB vs UFB Comparisons

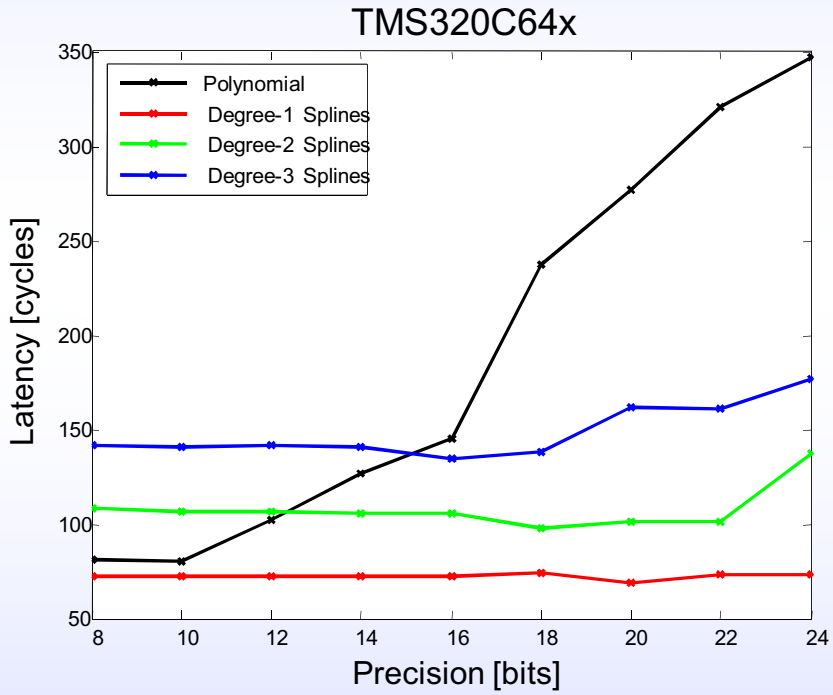
- Non-uniform fractional bit widths (NFB) allow reduced latency and code size relative to uniform fractional bit-width (UFB)



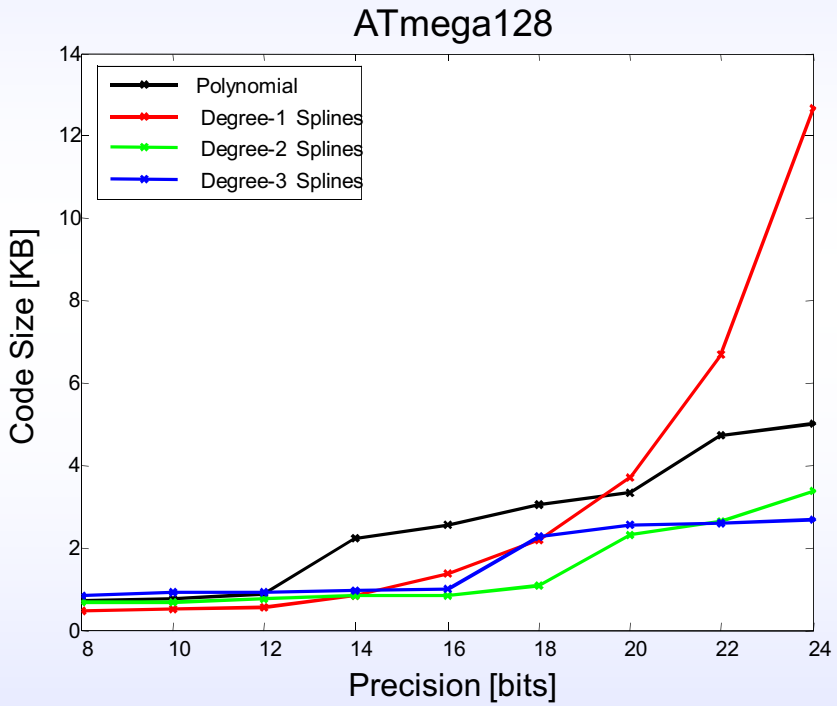


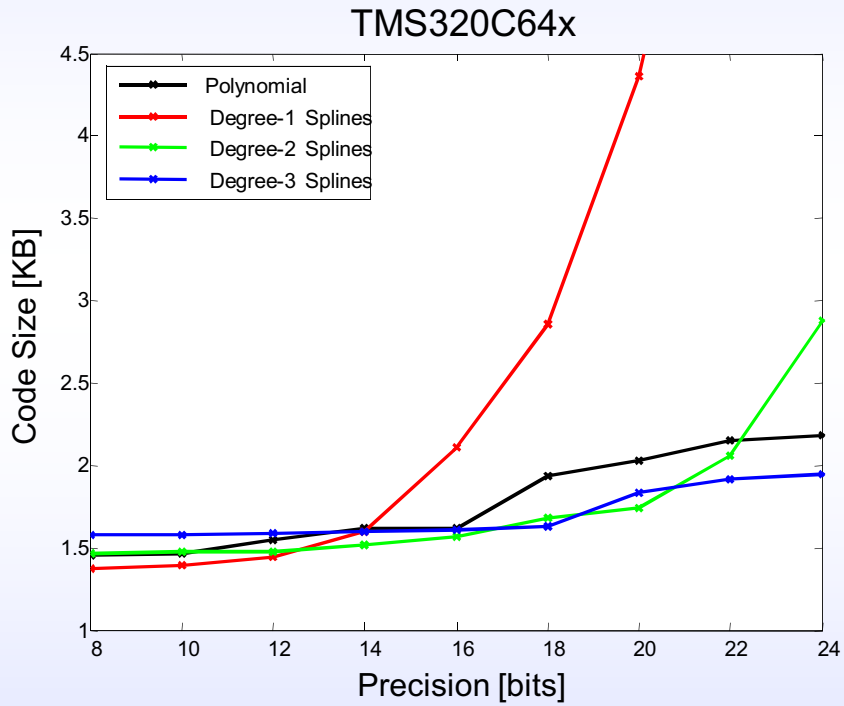
Latency Variations



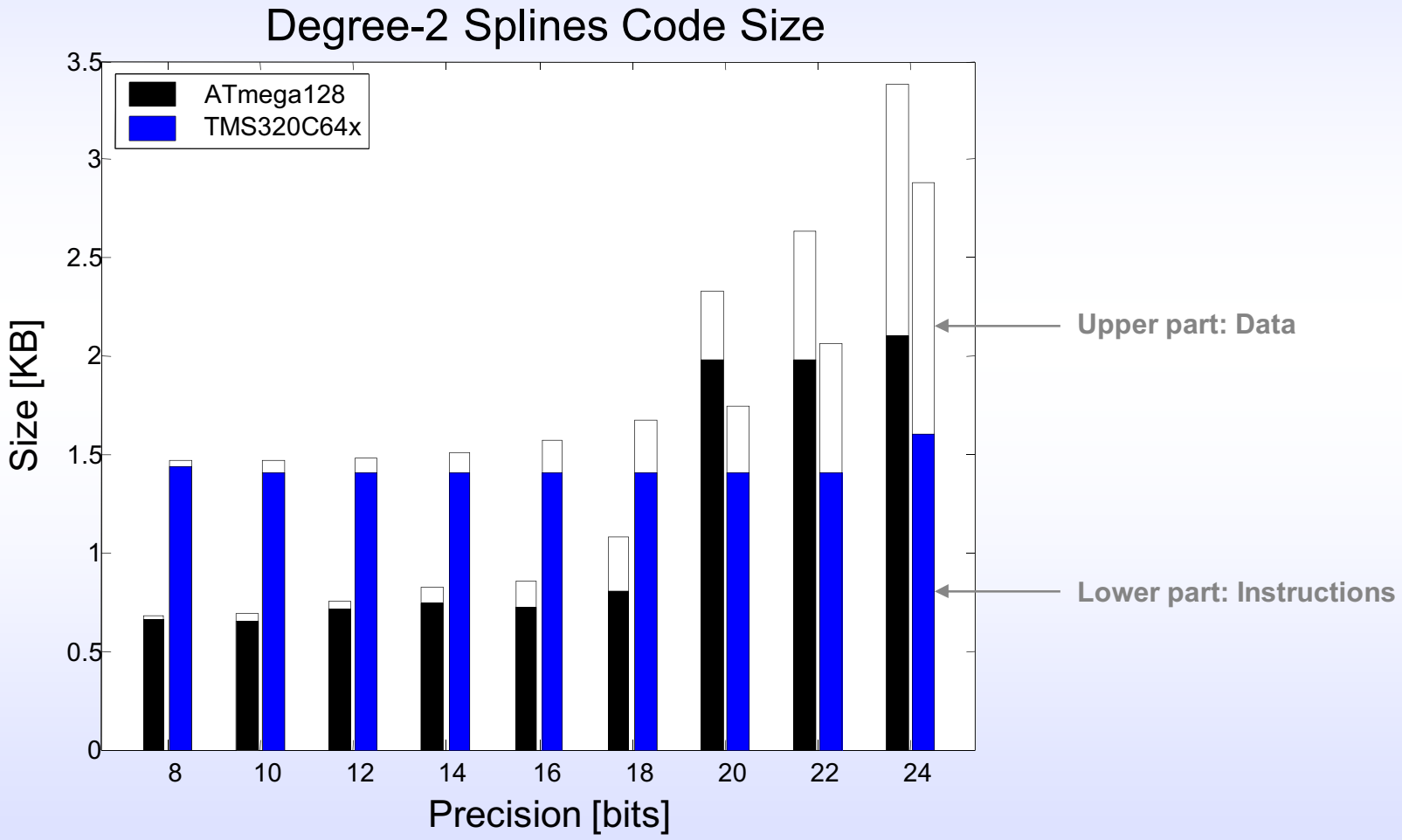


Code Size Variations





Code Size: Data and Instructions



Comparisons Against Floating-Point

- Significant savings in both latency and code size

Function	Atmel ATmega128 8-bit MCU					
	12-bit Precision (Degree-1)		16-bit Precision (Degree-2)		32-bit Floating-Point (C Math Library)	
	Latency [cycles]	Code Size [KB]	Latency [cycles]	Code Size [KB]	Latency [cycles]	Code Size [KB]
$\ln(x)$	115	0.56	497	0.86	5659	1.15
$\sin(x)$	114	0.56	383	0.94	5736	1.30
2^x	109	0.55	458	0.90	9316	1.57

Function	TI TMS320C64x 16-bit DSP					
	16-bit Precision (Degree-2)		24-bit Precision (Degree-3)		32-bit Floating-Point (C Math Library)	
	Latency [cycles]	Code Size [KB]	Latency [cycles]	Code Size [KB]	Latency [cycles]	Code Size [KB]
$\ln(x)$	106	1.57	177	1.95	2577	6.45
$\sin(x)$	104	1.49	173	1.93	2282	5.74
2^x	103	1.47	174	1.94	8602	10.61

Application to Gamma Correction

- Evaluation of $f(x) = x^{0.8}$ on ATmega128 MCU

Method	Simulated Latency		Measured Energy	
	Cycles	Normalized	Joules	Normalized
32-bit floating-point	10784	*13.6*	32 µJ	*13.0*
2^{-8} fixed-point*	791	*1*	2.4 µJ	*1*

No visible difference

* Degree-1 splines