APPROXIMATING WITH LOOK-UP TABLES AND POLYNOMIALS

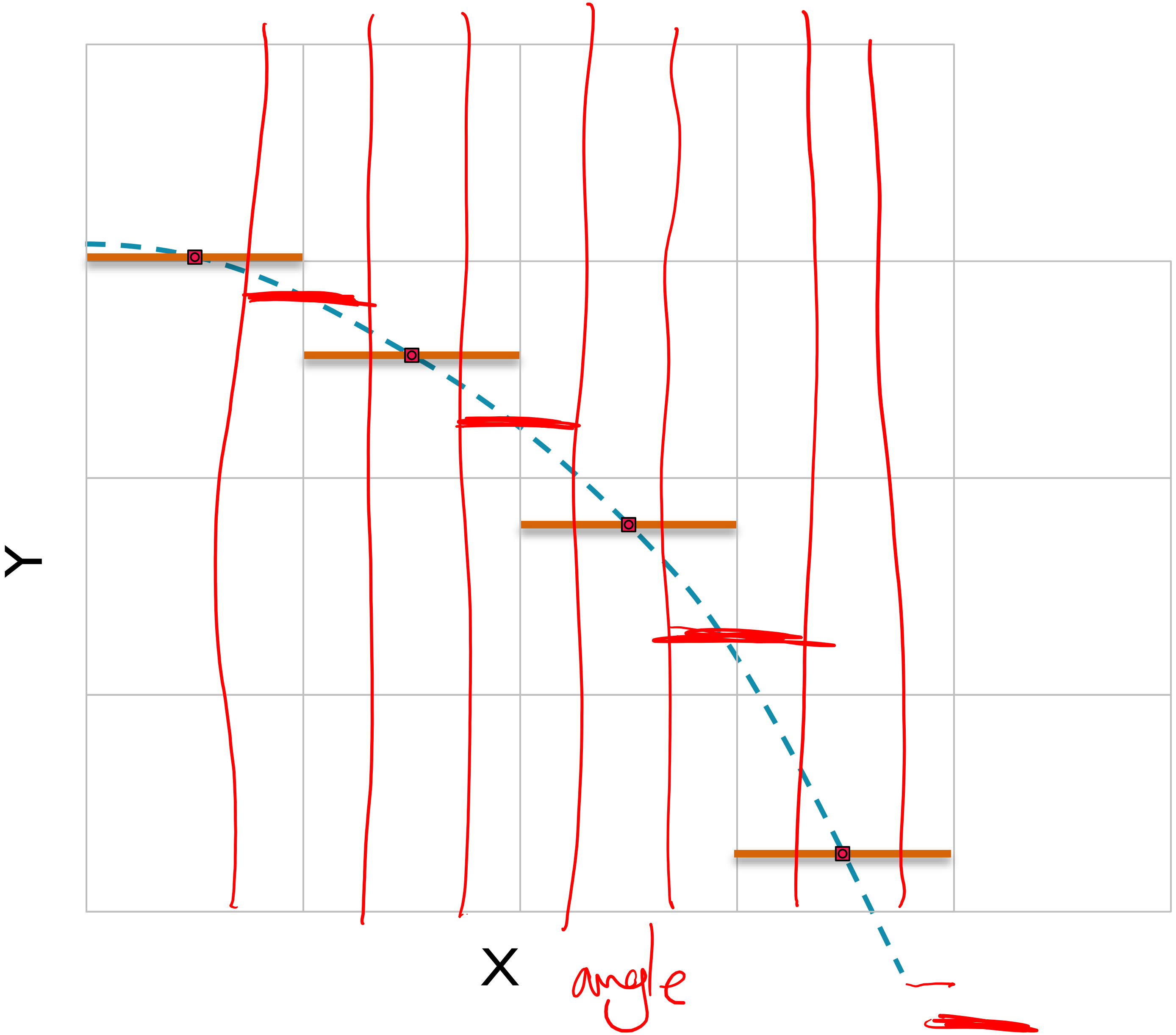
Approximations

- C math library has very accurate mathematical functions
  - Sin, cos, sqrt, etc. calculated with approximations
  - Accuracy takes computation time
  - May be more accurate than needed for your application
- Can simplify approximation of functions to save time
- Consider cosine function

Look-Up Table

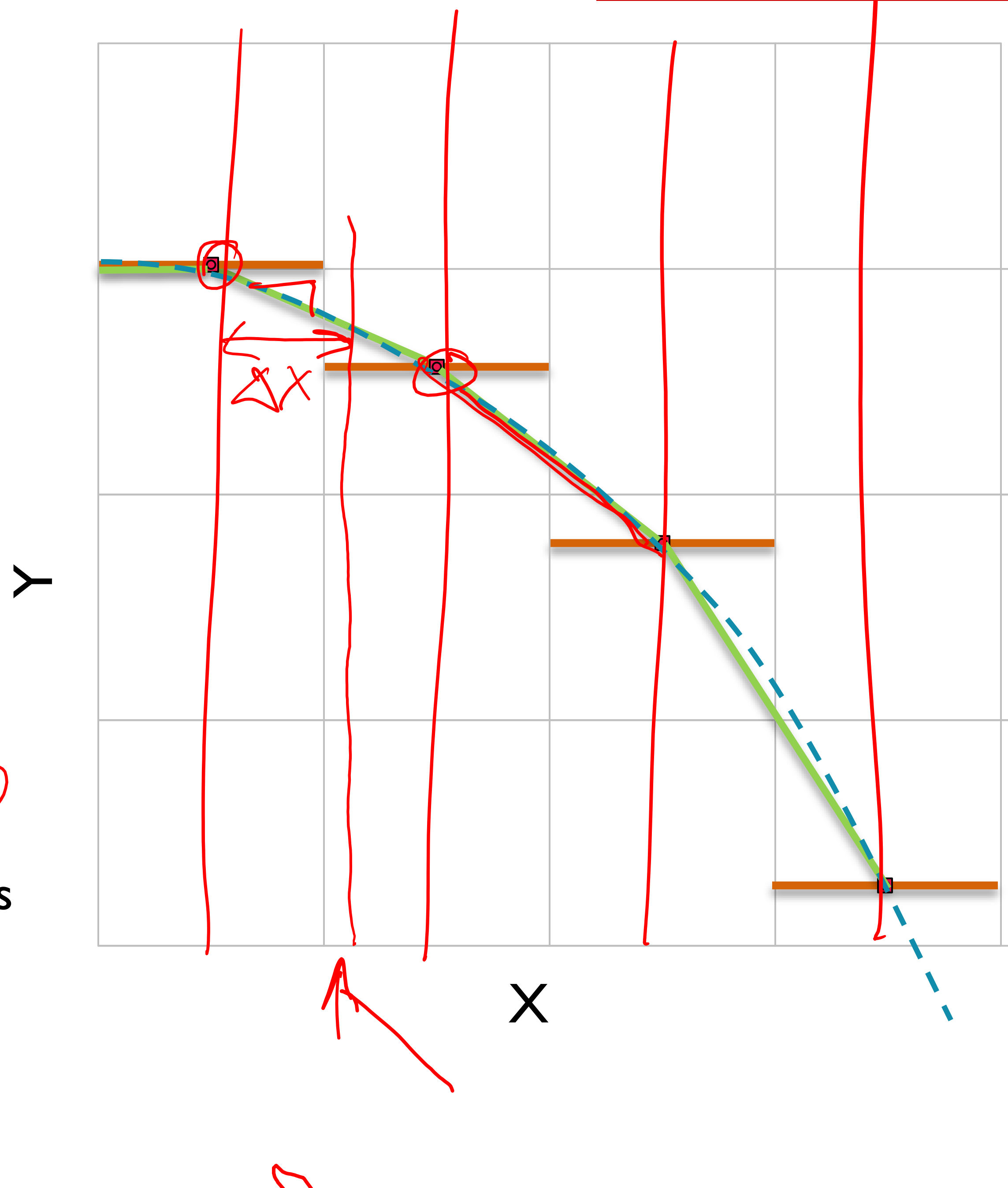
- Very fast
  - Convert input X to index i of table element
  - Read value from table[i]
- Potentially large memory requirements
  - Element size * number of elements
- Number of elements depends on accuracy required and how quickly function changes (derivative)

Double Accuracy



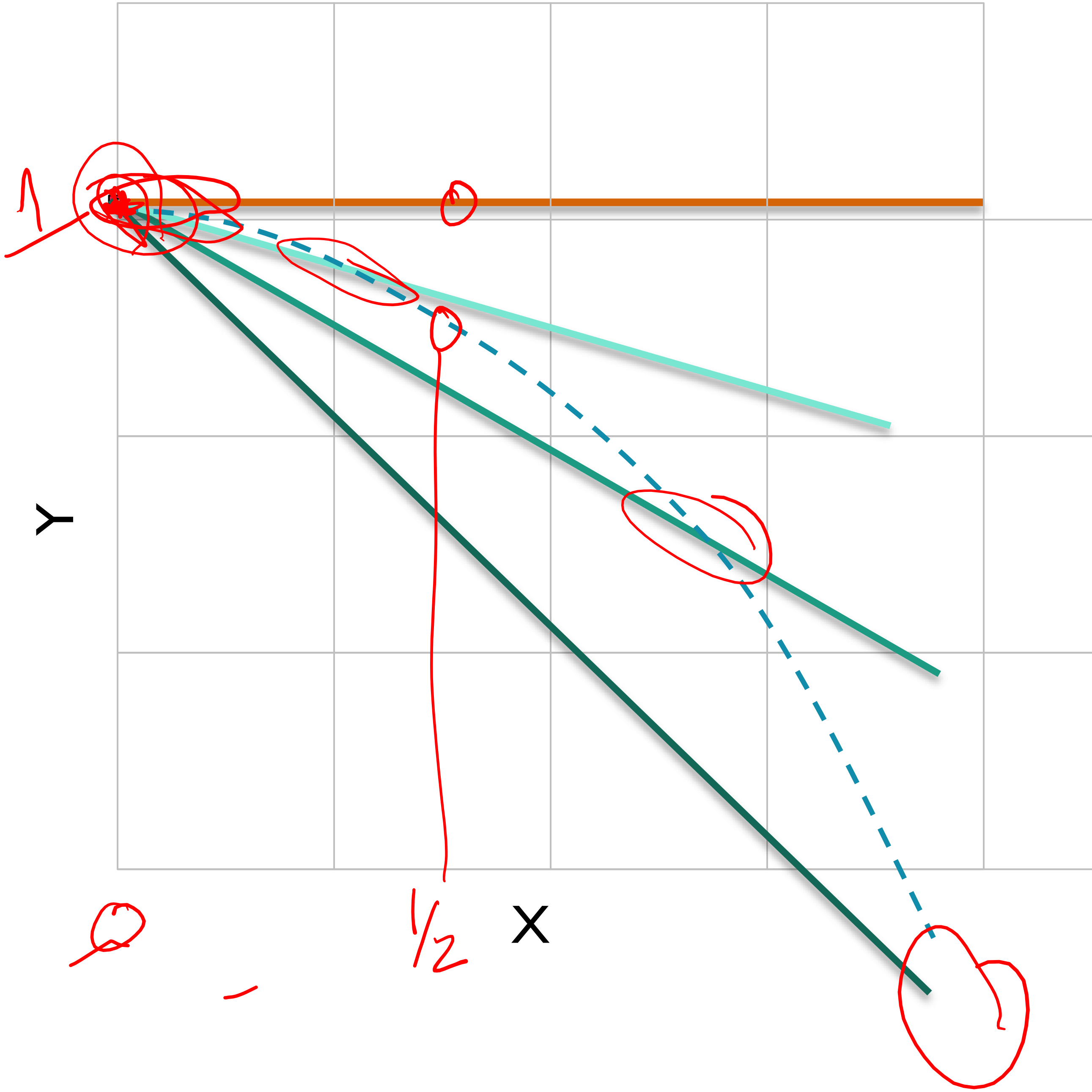
Look-Up Table with Interpolation

- Optimize by *interpolating* between adjacent data points
- A little slower
  - Find table entry i containing X – divide, or multiply by reciprocal
  - Subtract to find X offset of sample from table entry i
  - Multiply X offset by slope for table entry i
  - Add Y offset for table entry i $\cos(X) = mX + b$
- Much less error
  - Can reduce table size and memory requirements
- Example of approximation using linear interpolation



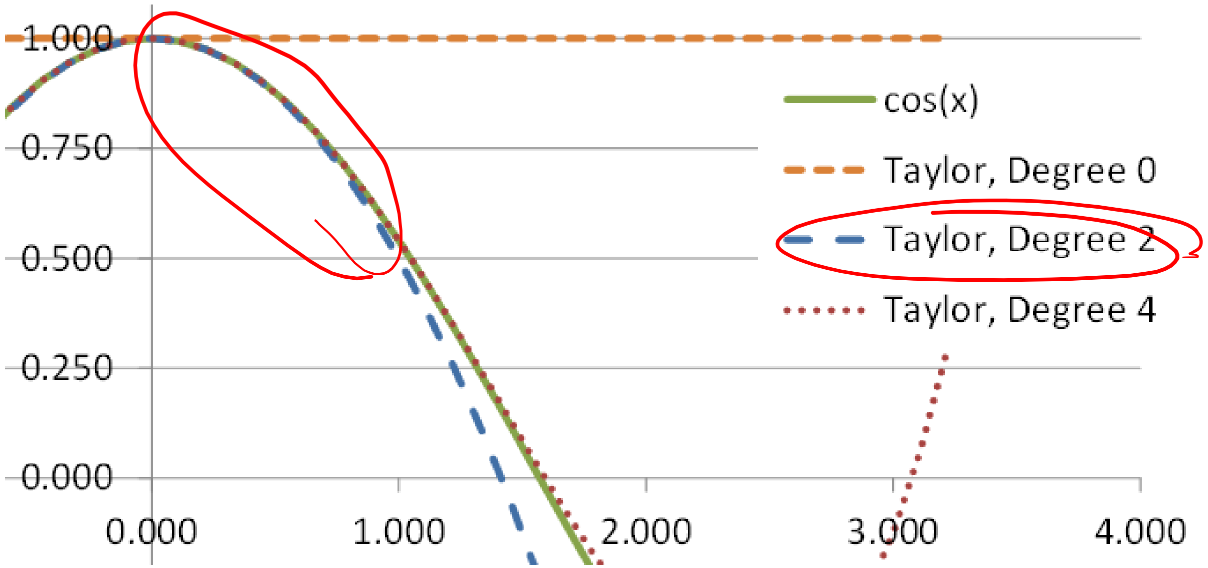
One-Element Look-Up Table

- How about a one-element look-up table?
- Constant approximation
  - cos(0) = 1
  - For very small values of x, cos(x) ≈ 1
  - Error increases quickly as x moves from 0, so limited use
- Linear approximations
  - cos(x) ≈ 1- x
  - Error still increases, but more slowly
  - How about cos(x) ≈ 1- 2x or cos(x) ≈ 1- x/2?
  - Or adding a constant?
- How about a better interpolation than linear?



Polynomial Approximations

- The Small Angle Approximation
  - $\cos(x) \approx 1 - x^2/2$
  - Is special case of Taylor series expansion (more soon)
- General case: Polynomial approximation
  - $f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + \cdots$
  - Can represent any arbitrary function Continuous
  - Improve accuracy by adding terms (a_5x^5, etc.)
  - Reduce accuracy by computing fewer terms
- Why use polynomials? Speed!
  - For a degree n polynomial, need n additions and $(n^2+n)/2$ multiplications



- Can reuse smaller terms (Horner's rule)
  - $x^{n+1} = x*x^n$
  - $f(x) = a_0 + x\left(a_1 + x\left(a_2 + x(a_3 + xa_4)\right)\right)$
  - For a degree n polynomial, need n additions and only n multiplications – much faster!

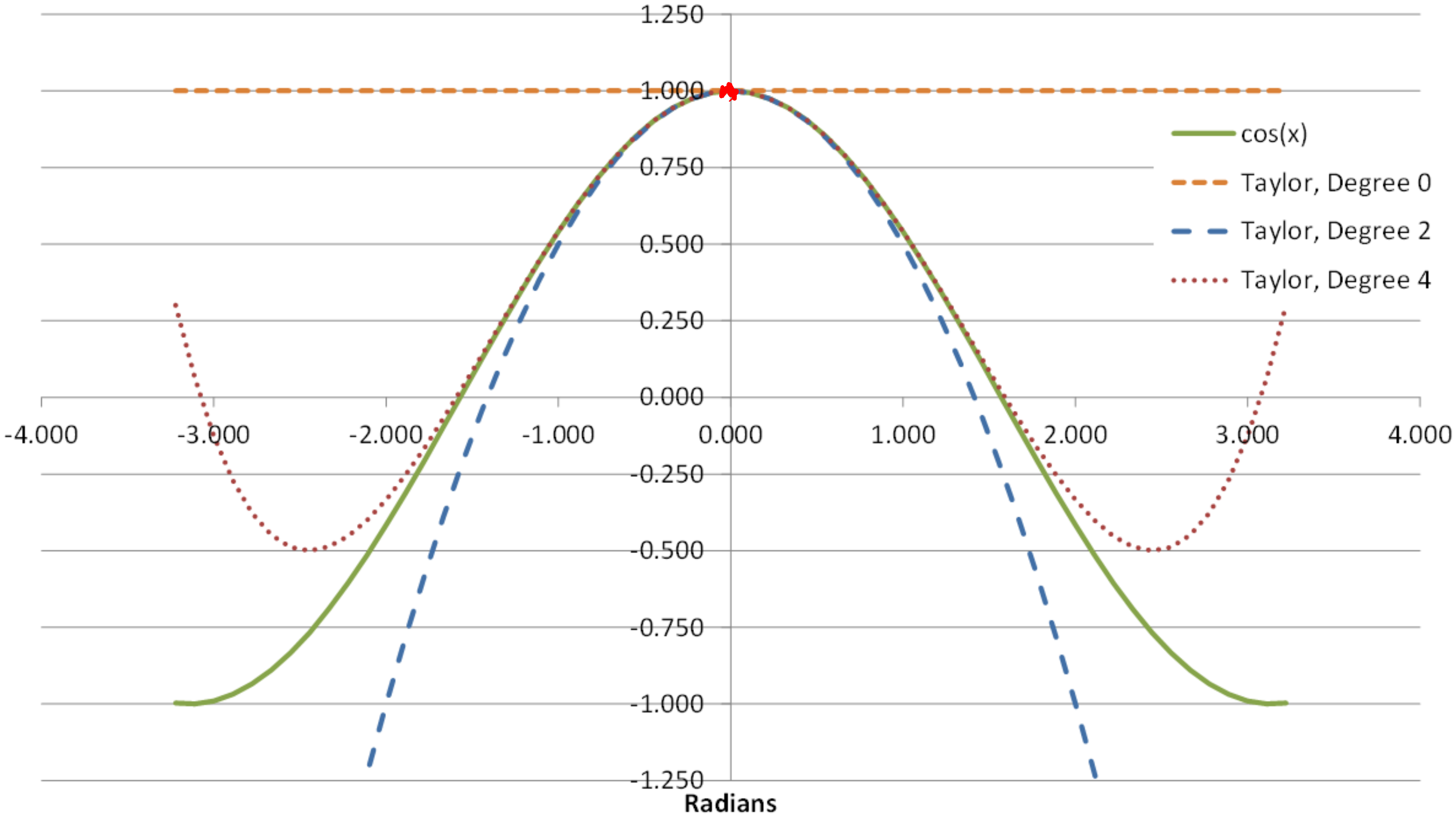
NC STATE UNIVERSITY

Determining coefficients

- Where do coefficients come from?
  - Can use Taylor or Maclaurin series
  - Other methods available too, which are more accurate or can use fewer terms
- Taylor Series
  - Coefficient a_n is based on *nth* derivative of the original function *f* at reference argument *r*
  - $\sum_{n=0}^{\infty} \frac{f^{(n)}(r)}{n!}(x-r)^n$
  - Factorials: 0! = 1
- Example: Taylor series for Cosine at r = 0
  - $\sum_{n=0}^{\infty} \frac{cos^{(n)}(r)}{n!}(x-r)^n = \sum_{n=0}^{\infty} \frac{cos^{(n)}(0)}{n!}(x-0)^n$
  - FYI: A Taylor series evaluated with r = 0 is called a Maclaurin series

- Derivative of cosine is –sine, derivative of sine is cosine
- $cos(x) = \frac{\cos(0)x^0}{0!} + \frac{-\sin(0)x^1}{1!} + \frac{-\cos(0)x^2}{2!} + \frac{\sin(0)x^3}{3!} + \frac{cos(0)x^4}{4!} + \frac{-\sin(0)x^5}{5!} + \frac{-\cos(0)x^6}{6!} + \cdots$
- Odd derivatives of cos are sin, and sin(0) = 0
  - So, no terms with odd exponent
- $cos(x) = \frac{\cos(0)x^0}{0!} + \frac{-\cos(0)x^2}{2!} + \frac{cos(0)x^4}{4!} + \frac{-\cos(0)x^6}{6!} + \ldots$
- cos(0) = 1, so simplify
- $cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$
- Note: signs of terms are alternating, and terms get closer to 0, so maximum error from truncation can be no larger than first truncated term

Accuracy



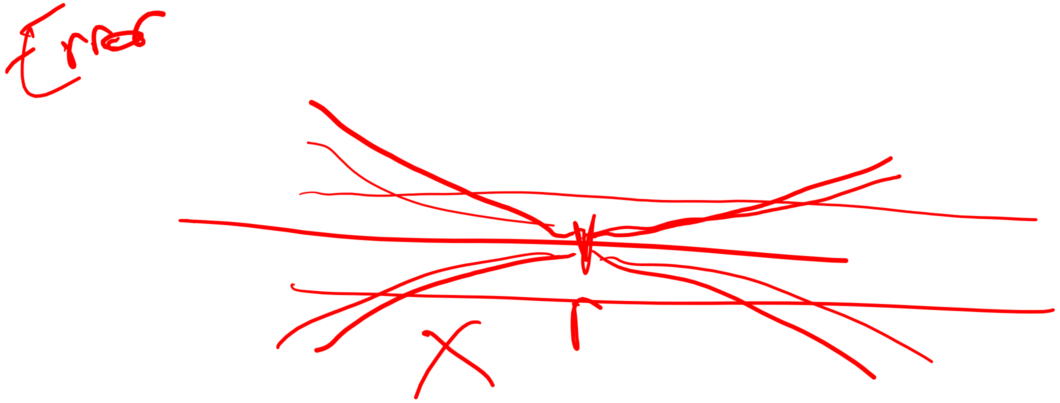
$$= 1 - \frac{x^2}{2}$$

$$= 1 - \frac{x^2}{2} + \frac{x^4}{4!}$$

- Accuracy increases with degree of approximation
- Accuracy decreases with increased distance from reference input r (r=0)

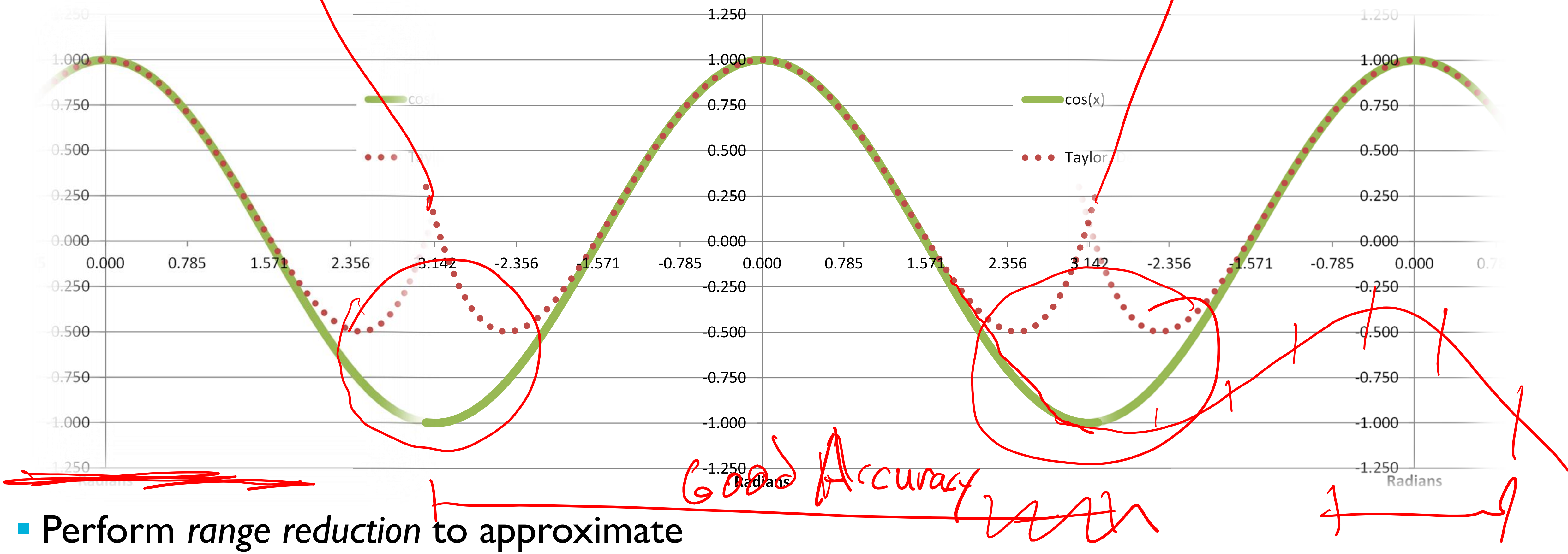
Improving Accuracy without Adding Terms

- Using Taylor series expansions for coefficients is simple and easy to understand, but not as good as other methods
  - Error is distributed unevenly: small near r, large far from r
- Can use other methods to determine coefficients
  - Get better accuracy
  - Distribute error more evenly over input range



- Typical methods
  - Chebyshev polynomials
  - Bessel functions
  - Minimax optimizations

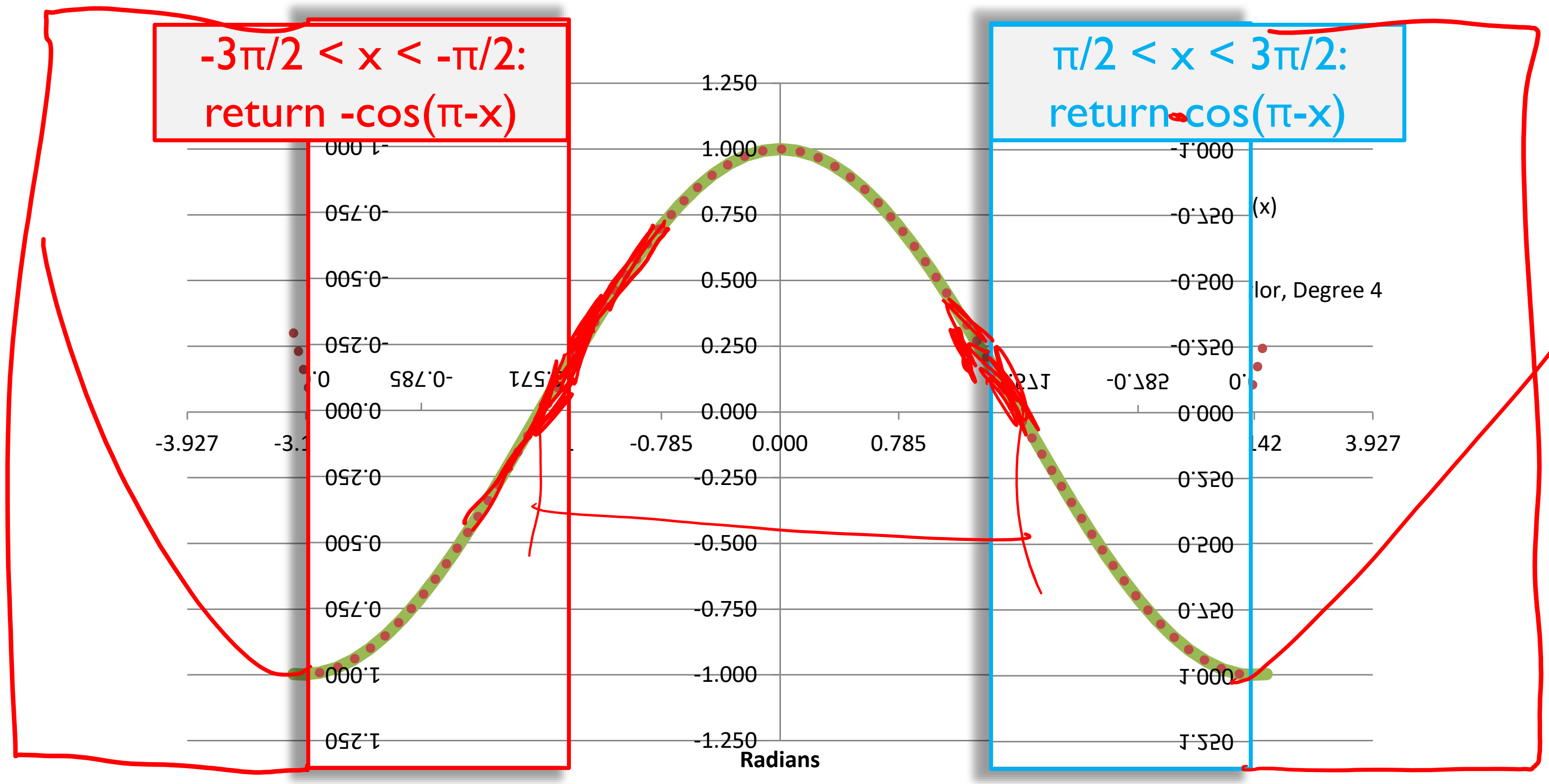
Approximating Periodic Functions



- Perform *range reduction* to approximate periodic functions
- For example, cos is periodic: *cos(x) = cos(x – n2π)*

- Subtract n2*π* (or perform modulo operation) to reduce input range to [-*π*, *π*]
- Much better, but still bad near n*3*π*/2

Approximating Symmetric Functions



- Perform *range reduction* to approximate symmetric functions
- For example, cos is symmetric: cos(x) = -cos(π-x)
- So we can approximate cos(x) from -π/2 to π/2, where accuracy is high

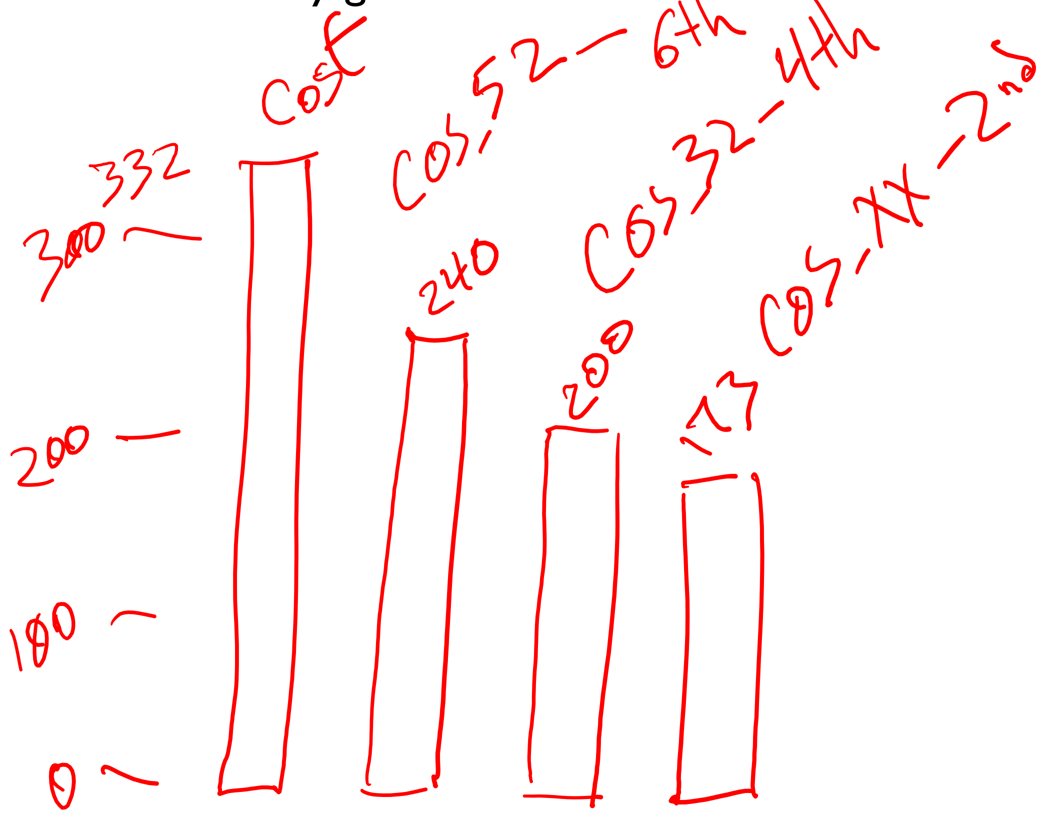
- If - 3π/2 < x < -π/2, return -cos(π-x)
- If - π/2 ≤ x ≤ π/2, return cos(x)
- If π/2 < x < 3π/2, return -cos(π-x)

Performance Evaluation

- How much faster than cosf() is the polynomial approximation?
- It depends…

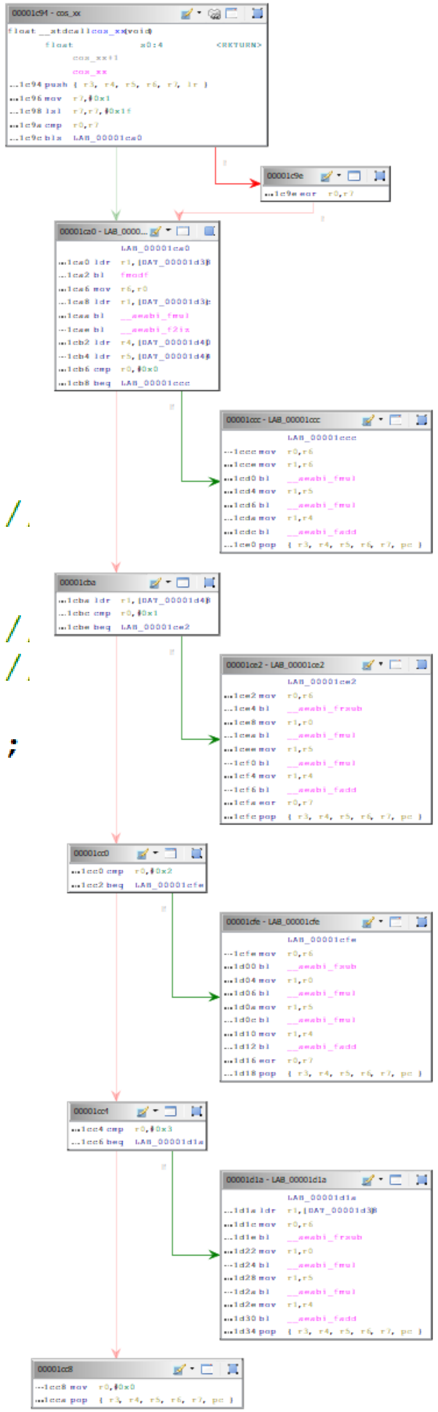
Details of Polynomial Trig Approximations

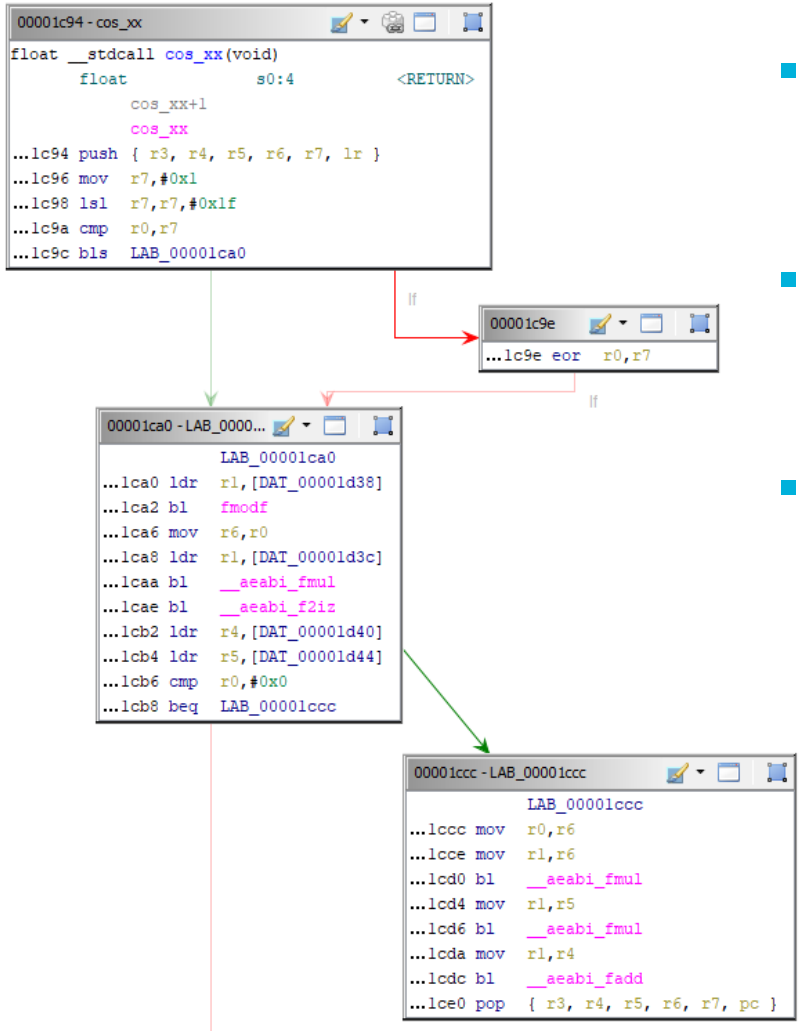
- A Guide to Approximations, Jack Ganssle
  - http://www.ganssle.com/item/approximations-for-trig-c-code.htm
  - What object code do we actually get? Source code is in sincos.c



CFG for cos_xx

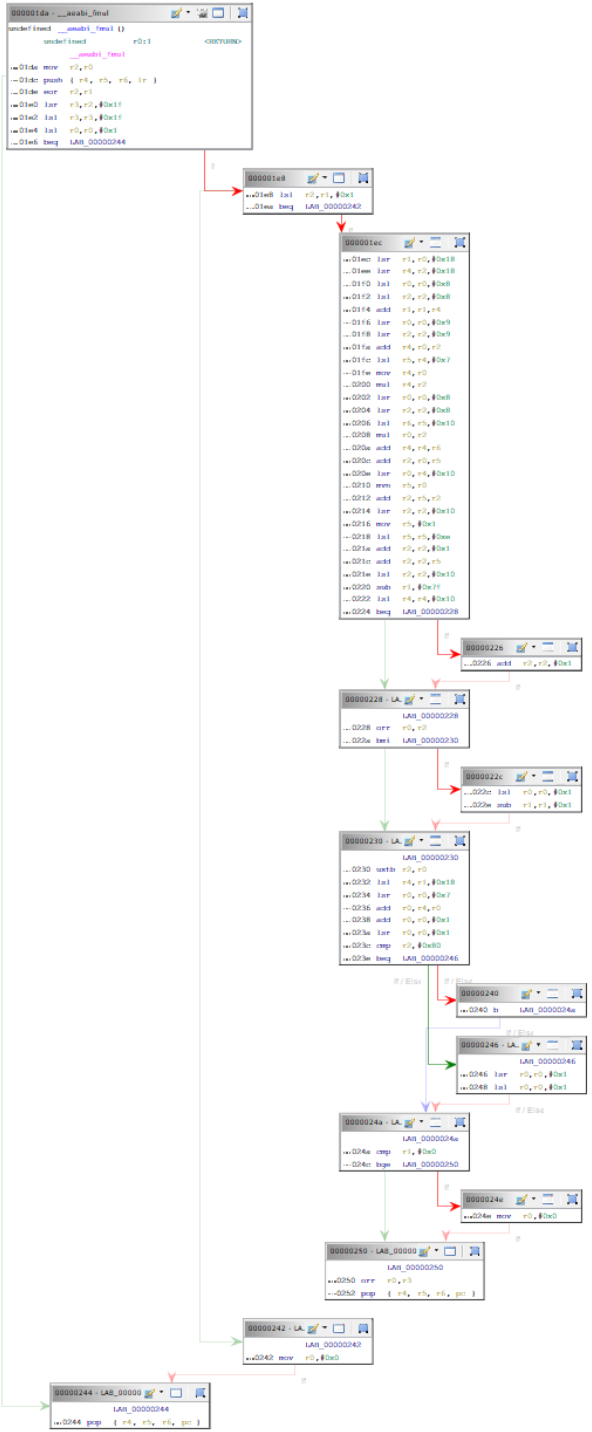
```
float cos_xxs(float x) {
  float x2;

  x2 = x * x;
  return (c1 + x2 * c2);
}

float cos_xx(float x) {
  int quad;                              /.

  if (x < 0)
    x = -x;                              /.
  x = fmod(x, twopi);                    /.

  quad = (int) (x * two_over_pi);
  switch (quad) {
  case 0:
    return cos_xxs(x);
  case 1:
    return -cos_xxs(DP_PI - x);
  case 2:
    return -cos_xxs(x - DP_PI);
  case 3:
    return cos_xxs(twopi - x);
  }
  return 0.0;
}
```

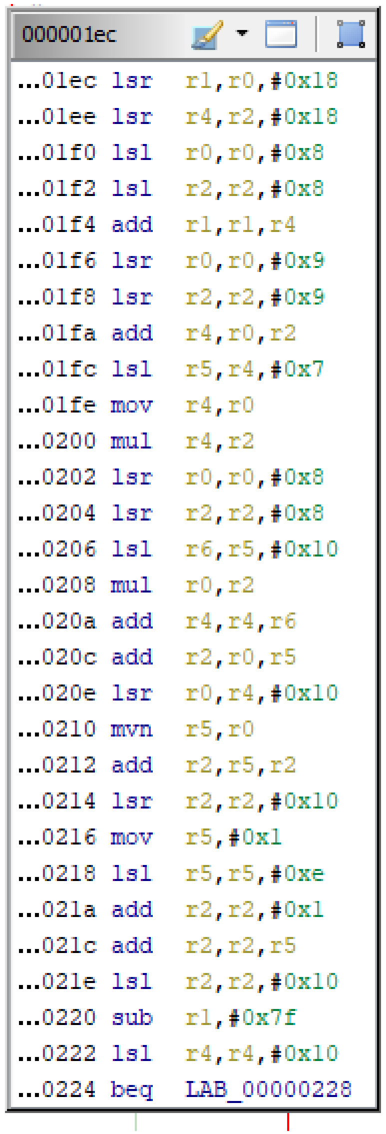




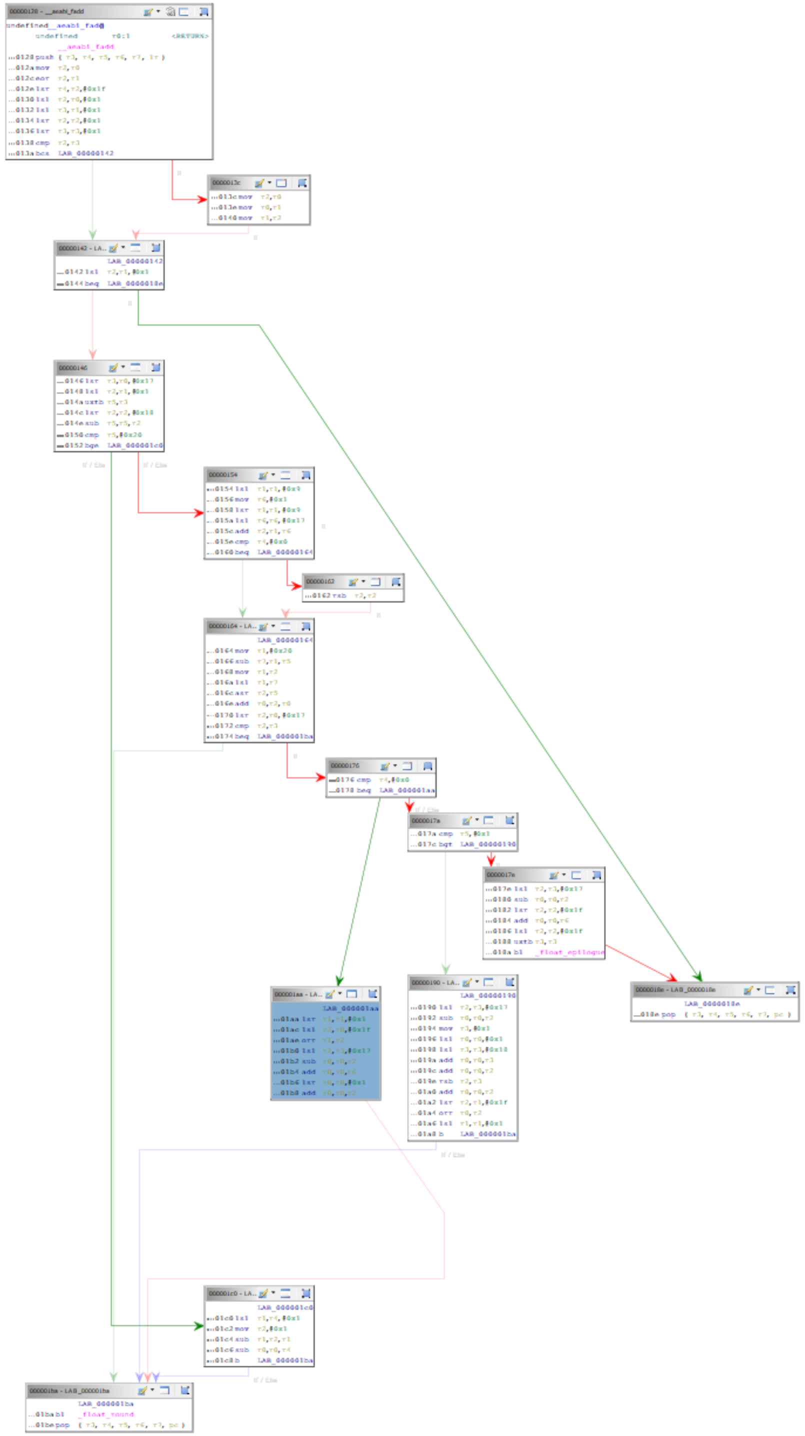
- Switch statement uses if/elseif ladder
- cos_xxs is inlined into cos_xx
- Generated code looks efficient

CFG for __aeabi_fmul





CFG for __aeabi_fadd



NC STATE UNIVERSITY