Math

Chapter 9

Accuracy and Precision

- Accuracy: how correct
  - Accuracy is limited
- Precision: how many digits
  - Precision could be noise

Mass of an electron	Is it precise?	Is it accurate?
12.12345124 kg	Stupidly so	No
10^{-30} kg	No	Far more so than 12 kg, and accurate enough for most conversations
9.109×10^{-31} kg	Yes	Even more so
$9.1090000001 \times 10^{-31}$ kg	Uselessly so	Same accuracy as the previous answer
$(9.1092 +/- 0.0002) \times 10^{-31}$ kg	Yes	Yes, best answer yet

Speedy Operations

- Bit operation
- Addition, subtraction
- Multiplication
- Division
- Modulo
- Case25.math

Speedy Operations

- Bad

```
for (i=0; i<100; i++) {
  if ((i%10) == 0) {
    printf("%d percent done.", i);
  }
}
```

- Good

```
for (i=0; i<100; i++) {
  if (!(i & 0x07)) { // this will print out every 8th pass
    printf("%d percent done.", i);
```

Some Magic Numbers

- Powers of 2
  - Multiplication and division
    - Convert to bit shifting
- Unsigned numbers
  - Faster in math operations
    - No signs to track or extend
- Constants
  - Constants as #define are operands.
  - ~~Constants as const are variables.~~

Common Arithmetic

- Arithmetic operations
  - Average
  - Variance / Deviation
  - Sine / Cosine / Tangent
  - Logarithm
  - Exponentiation
- Methods
  - Direct calculation
  - Polynomial approximation
  - Lookup table + interpolation

Average

- Average with a rolling window

```
newAverage = lastAverage + (newSample/length) - (oldestSample/length);
```

- Problems
  - Overflow
  - Underflow: division precision

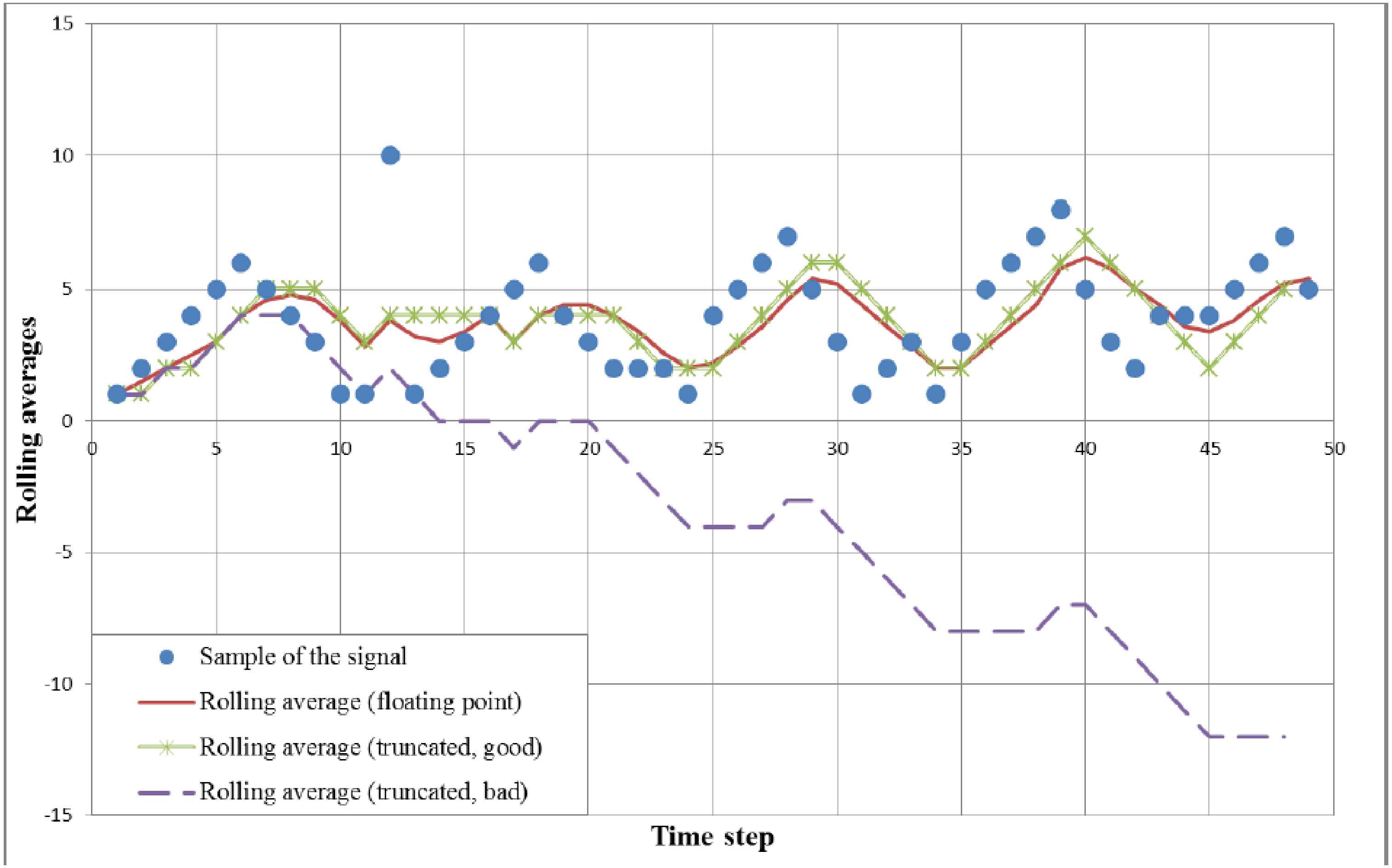
```
newAverage = lastAverage + ((newSample- oldestSample)/length);
```

  - If the difference is smaller than length, the new average will be the same.

Problems

- Overflow
  - Addition of 2^m n-bit numbers
  - (n+m)-bit result
  - 11+13=24
  - Integer multiplication
- Underflow: division precision
  - Integer division
  - n-bit number divided by 2^m
  - (n-m)-bit result
  - 59/4=14

Division Precision



Proper Average

- Case26.average
- Have a struct
  - ave
  - sum
  - length
- sum = sum + (newSample – oldestSample)
- ave = sum / length

- Notes:
  - Precision of samples
  - Precision of lastSum

Proper Average

- The solution is easy to state:
  - Large values should be divided by much smaller ones.
  - They shouldn’t be similar in magnitude.
- Choose a proper length of window
  - Addition can cause overflow
  - Division can cause underflow
- Scale numbers but choose a proper scale
  - Multiplication can cause overflow
- Know and depend upon a range of input values so that you can make sure your accuracy is not degraded by the lack of precision.

Standard Deviation

- Standard deviation
- Don't do square root
- Variance

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2} \qquad \sigma = \sqrt{\frac{1}{N-1}\left(\sum_{i=1}^{N} x_i^2\right) - \bar{x}^2}$$

```
uint16 GetVariance(int16_t* samples, uint16_t numSamples, int16_t mean) {
  uint32_t sumSquares = 0;
  int32_t tmp;
  uint32_t i = numSamples;

  while (i--) {
    tmp = *samples - mean;
    samples++;
    sumSquares += tmp*tmp;
  }
  return (sumSquares/(numSamples-1));
}
```

Variance

```
struct sVar {
  int32_t sum;
  uint64_t sumSquares;
  uint16_t numSamples;
};
void AddSampleToVariance (stuct sVar *var, int16_t newSample) {
  var->sum += newSample;
  var->sumSquares += newSample*newSample;
  var->numSamples++;
}
uint16_t GetVariance(struct sVar *var, int16_t *average) {
  uint16_t variance;
  // This method also outputs average
  *average = var->sum/var->numSamples;
  variance = (var->sumSquares - (var->sum * (*average)))
                       /(var->numSamples-1);
  // get ready for the next block
  var->sum = 0; var->numSamples = 0; var->sumSquares = 0;
  return variance;
}
```

Variance

- Two-pass variance

```
struct sVar {
  int16_t mean;
  int32_t M2;
  uint16_t numSamples;
};
void AddSampleToVariance(struct sVar *var, int16_t newSample) {
  int16_t delta = newSample - var->mean;
  var->numSamples++;
  var->mean += delta/var->numSamples;
  var->M2 += delta * (newSample - var->mean); // uses the new mean
}
uint16_t GetVariance(struct sVar *var, int16_t *average) {
  uint16_t variance = var->M2/var->numSamples;
  *average = var->mean; // running average already calculated
  // get ready for the next block
  var->numSamples =0; var->mean = 0; var->M2 = 0;
  return variance;
}
```

Variance

- Case27.variance
  - Classical
    - Integer vs double
  - Single pass
    - Good vs bad
  - Two pass
    - Good vs bad

Divide a Constant

- Divide 6
- Divide 7?

Table 9-1. Approximating 1/6 with a power-of-two divisor

Multiplier	Divisor	Equivalent shift	Result	% Error
1	6	None	0.166666667	0
3	16	4	0.1875	12.5
5	23	5	0.15625	6.2
11	64	6	0.171875	3.1
21	128	7	0.164063	1.5
43	256	8	0.167969	0.78
85	512	9	0.166016	0.39
171	1024	10	0.166992	0.19
341	2048	11	0.166504	0.09
683	4096	12	0.166748	0.04

Factor Polynomial

- 3 multi + 2 add

```
y = A*x + B*x + C*x;
```

- 2 add + 1 multi

```
y = (A + B + C)*x;
```

- Power + multi + add
- 3 mtlti + 2 add

$A*x^3+B*x^2+Cx \implies ((Ax+B)x+C)x$

- Tyler series
  - Sine

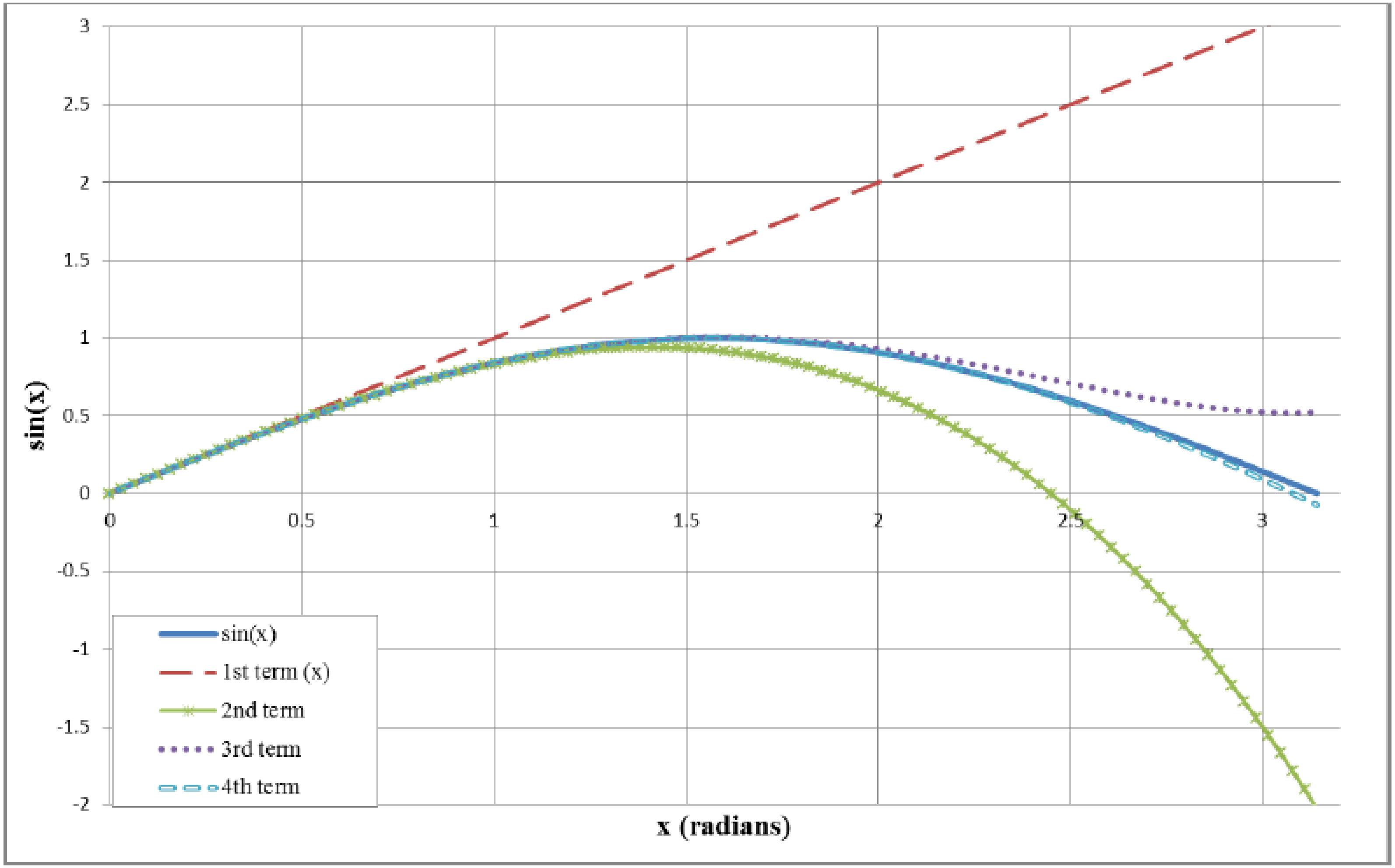
$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

$$\sin(x) \approx x - Ax^3 + Bx^5 - Cx^7, where\ A = \frac{1}{3!}, B = \frac{1}{5!}, C = \frac{1}{7!}$$

$$\sin(x) = x * (1 - x^2(A + x^2(B - Cx^2)))$$

$$\sin(x) = x * (1 - x^2\left(\frac{1}{3!} + x^2\left(\frac{1}{5!} - \frac{1}{7!}x^2\right)\right))$$

Taylor Series: Sine



Taylor Series: Sine

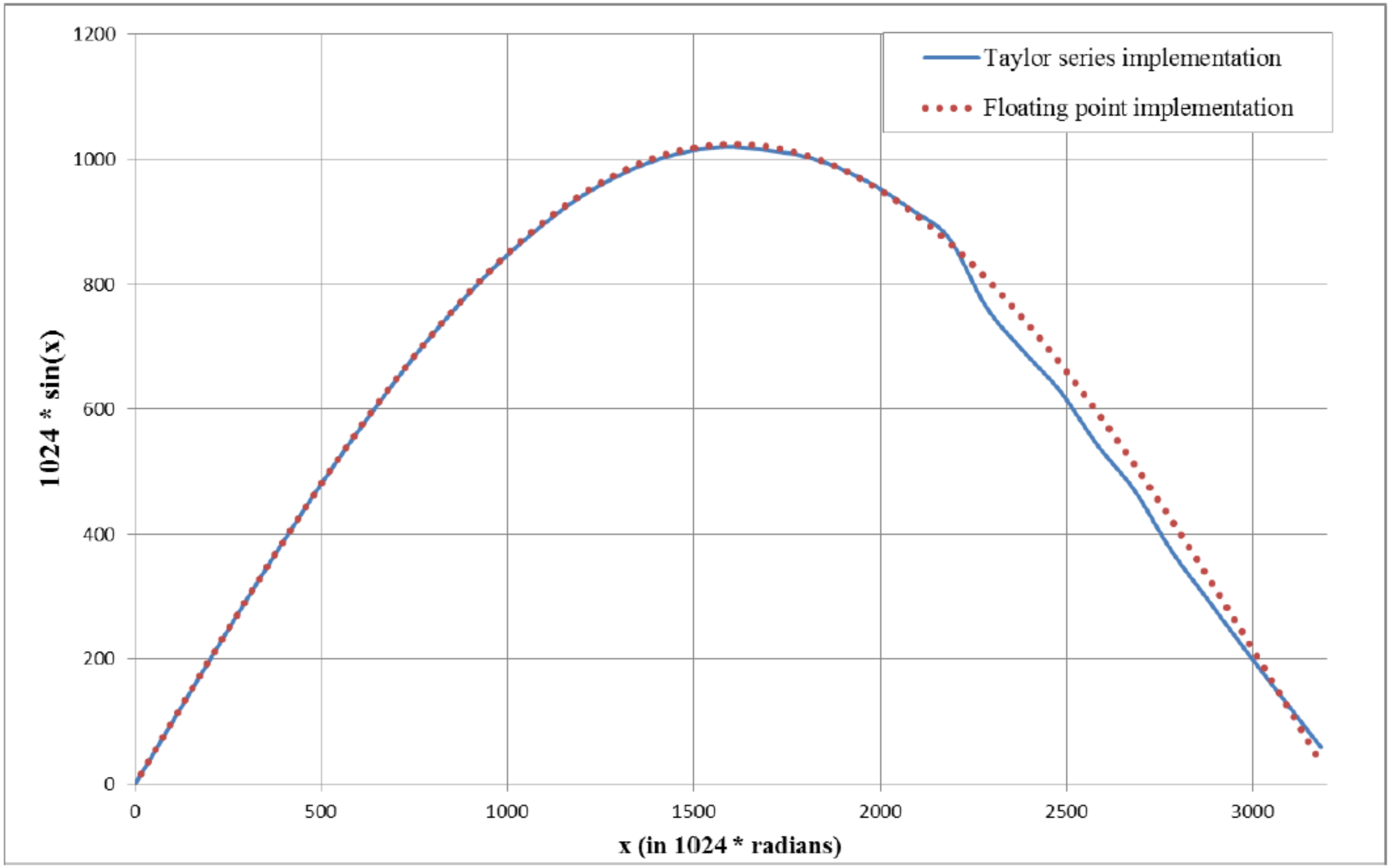
```
xSq = x*x;
sinX = x * (1 - xSq * (INVERSE_THREE_FACTORIAL + xSq *
            (INVERSE_FIVE_FACTORIAL - INVERSE_SEVEN_FACTORIAL * xSq)));

// or taking it apart and writing multiple steps, stating with
// the end and moving toward the front:
tmp = - INVERSE_SEVEN_FACTORIAL * xSq;
tmp = xSq * (INVERSE_FIVE_FACTORIAL + tmp);
tmp = xSq * (-INVERSE_THREE_FACTORIAL + tmp);
sinX = x * (1 - tmp);
```

Scale Input

- Sine
  - Scale x by 1024.
  - All polynomial factors need to be scaled too.
    - 1/3! * 1024 = 171
    - 1/5! * 1024 = 8 or 9 (8.533)
    - 1/7! * 1024 = 1/5

```
xSq = (x*x) >> 10;  // right-shift by 10 is equal to divide by 1,024
tmp = - DivideSevenFactorial(xSq);
tmp = (xSq * (INVERSE_FIVE_FACTORIAL + tmp)) >> 10;
tmp = (xSq * (-INVERSE_THREE_FACTORIAL + tmp)) >> 10;
sinX = x - ((x * tmp)>>10);
```

Scale Input



1200
1000
800
600
400
200
0
0
500
1000
1500
2000
2500
3000
1024 * sin(x)
x (in 1024 * radians)
Taylor series implementation
Floating point implementation

Taylor Series: Sine

- Case28.taylor
  - Polynomial
  - Divide constants
    - 1/6 = 683/4096
    - 1/120 = 546/65536
    - 1/5040 = 13/65536
  - Scale input (and output)
    - 1024x
    - Polynomial scale
      - Factors divided by scale
      - Factors divided by scale*scale

Lookup Table

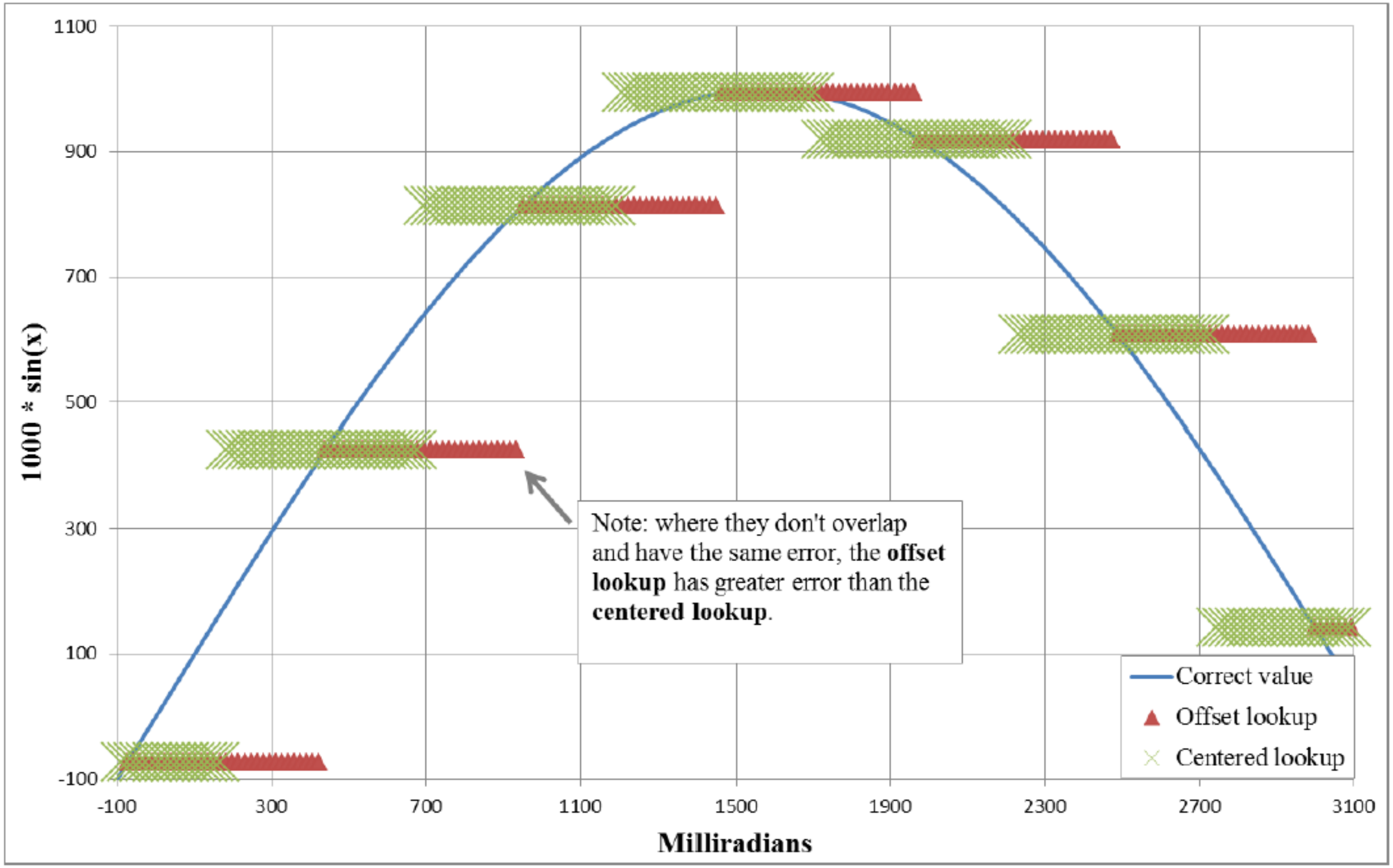
```
const int16_t sinLookup[] = {
  -58,   // x =  -3200
  -335,  // x =  -2800
  -676,  // x =  -2400
  -910,  // x =  -2000
  -1000, // x =  -1600
  -933,  // x =  -1200
  -718,  // x =  -800
  -390,  // x =  -400
   0,    // x =  0
   389,  // x =  400
   717,  // x =  800
   932,  // x =  1200
   999,  // x =  1600
   909,  // x =  2000
   675,  // x =  2400
   334,  // x =  2800
  -59    // x =  3200
};
uint8_t index = (x - (-3200))/400;
y = sinLookup[index];
```

```
const int16_t sinLookup[] = {
  3,     // x @ -3145, for range
  -487,  // x @ -2633, for range
  -853,  // x @ -2121, for range
  -1000, // x @ -1609, for range
  -890,  // x @ -1097, for range
  -553,  // x @ -585,  for range
  -73,   // x @ -73,   for range
  425,   // x @ 493,   for range
  813,   // x @ 951,   for range
  994,   // x @ 1463,  for range
  919,   // x @ 1975,  for range
  608,   // x @ 2487,  for range
  142,   // x @ 2999,  for range
};

uint8_t index = (x - (-3145 - 256))>>9;
y = sinLookup[index];
```

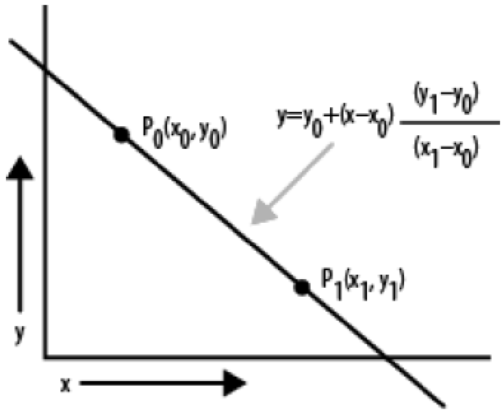
Lookup Table

- Scale of input
  - 1000x
- Scale of output
  - 1000x (1024x)
- Step size : determine # of entries
  - 200
  - Power of 2
  - Variable
- Position of entries
  - Left
  - Center

Lookup Table



Linear Interpolation




```
struct sPoint {
  int16_t x;
  int16_t y
};
```

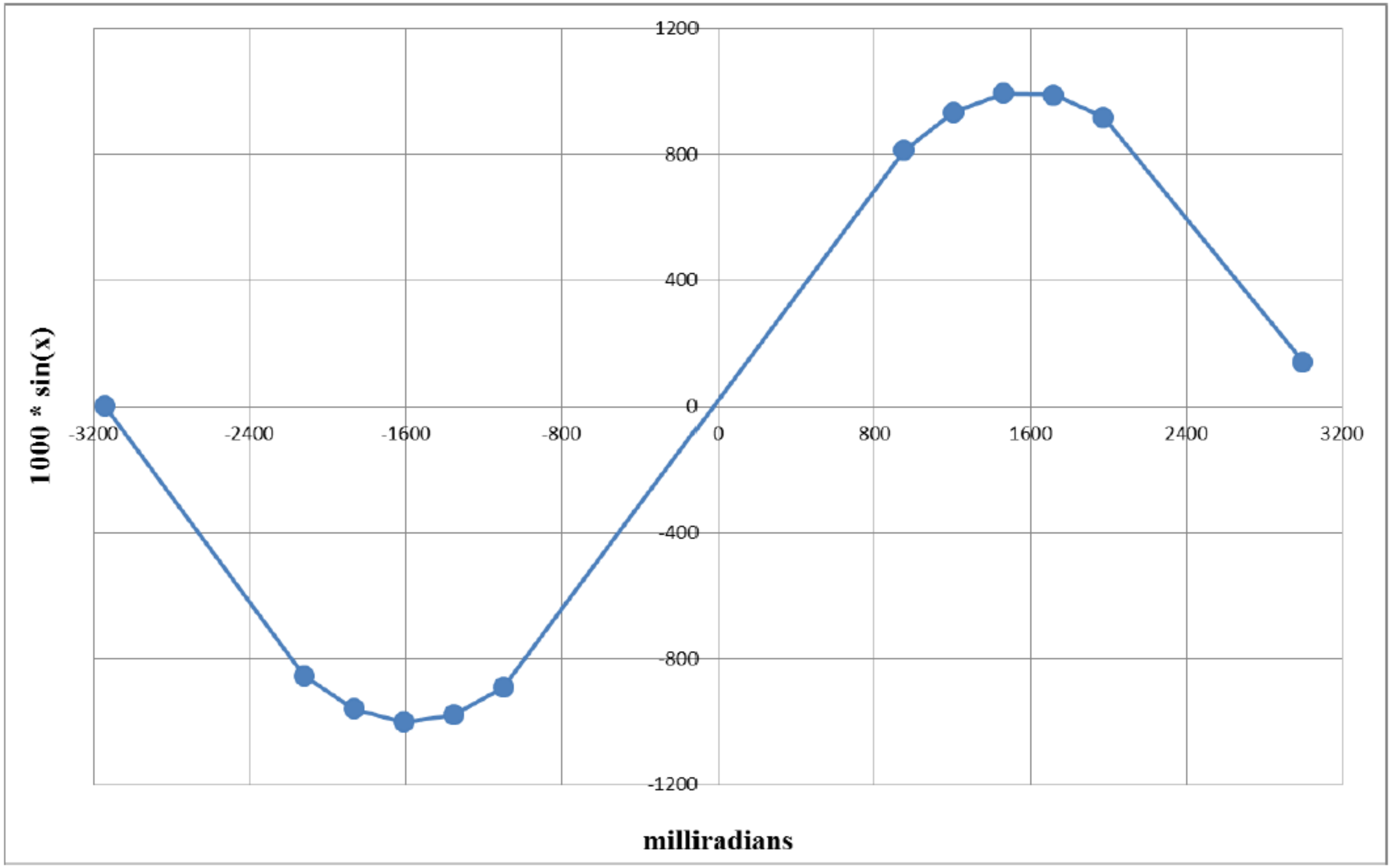
```
// This linear interpolation code does:
// y = p0.y + ((x-p0.x)*(p1.y-p0.y))/(p1.x-p0.x);
// but in a bit-safe way.
int16_t Interpolate(struct sPoint p0, struct sPoint p1, int16_t x) {
  int16_t y;
  int32_t tmp; // start and end with int16s, but can need a larger intermediate
  tmp = (x - p0.x);
  tmp *= (p1.y - p0.y);
  tmp /= (p1.x - p0.x);
  y = p0.y + tmp; // now safe to go back to 16 bits
  return (y);
}
```

Example: Linear Interpolation

- Table 1: (step size 200)
  - x=2100
  - x=300
  - Errors
- Table 2: (step size 256)
  - x=2100
  - x=300
  - Errors
- Uniform step size is not always good.
  - Smaller steps on curves
  - Larger steps on straight lines

Explicit Lookup Table

- A table has both inputs and outpus
- Variable step sizes to improve accuracy



Explicit Lookup Table

- $Y = c_0 + c_1 x + c_2 x^2 + ...$
- 1st tayler item
  - Linear interpolation
- 2nd tayler item
  - Error on linear interpolation
  - Bigger c2, larger error
  - Bigger c2, more curve
- Bigger step size along straight lines
- Smaller step size along curves

Explicit Lookup Table

```
int SeachLookupTable(int32_t target, struct sPoint const *table, int tableSize) {
  int i;
  int bestIndex = 0;

  for (i=0; i<tableSize; i++) {
    if (target > table[i].x) {
      bestIndex = i;
    } else {
      return bestIndex;
    }
  }
  return bestIndex;
}

index = SeachLookupTable (x, sinLookup, sizeof(sinLookup));
if (index+1 < sizeof(sinLookup)) {
  y = interpolate(x, sinLookup[index], sinLookup [index + 1]);
} else {
  y = interpolate(x, sinLookup[index-1], sinLookup [index]);
}
```

Fake Floating Numbers

- Binary scaling

```
struct sFakeFloat {
  int32_t num;  // numerator
  int8_t shift; // right-shift values (use negative for left-shift)
}
floatingPointValue = num >> shift;    // in the actual code
```

```
struct sFakeFloat oneFourth = {1, 2};
struct sFakeFloat four = {1, -2};
```

Precision

Table 9-2. Representing the number 12.345 using binary scaling

Numerator	Number of bits needed in numerator	Denominator shift values	Equivalent floating-point number	Error
12	4	0	12	0.345
25	5	1	12.5	0.155
99	7	3	12.375	0.030
395	9	5	12.34375	0.00125
12641	14	10	12.34472656	0.000273
12944671	24	20	12.34500027	2.67E–07
414229463	29	25	12.345	1.19E–09
1656917852	31	27	12.345	1.19E–09

Operations

- Addition (Subtraction)
  - Left shift the larger number
    - Both numbers have the same shift
  - Add two numbers
  - Keep the shift
- Multiplication (Division)
  - Multiply two numbers
  - Add two shifts
  - Right shift the result to make the result in range.
- Case30.fakefloat