CSC 344 – Algorithms and Complexity

Lecture #11 – Numerical Computation, Numerical Integration and the Fast Fourier Transform

Calculating e^x

- $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \cdots + \frac{x^i}{i!} + \ldots$
- How do we write the function?

exp1()

```
double  exp1(int x)     {
        double sum = 0.0, term  = 1.0;
        int     i;

        for (i = 0;  term >= sum/1.0e7; i++)    {
                term = power(x, i)/fact(i);
                sum += term;
        }
        return sum;
}
```

- ***What wrong with this function?***

exp3()

```
double  exp3(int x)     {
        double sum = 1.0, term  = 1.0;
        int     i;

        for (i = 1;  term >= sum/1.0e7; i++)    {
                term = term * x / (double)i;
                sum += term;
        }
        return sum;
}
```

- *Is this faster?*

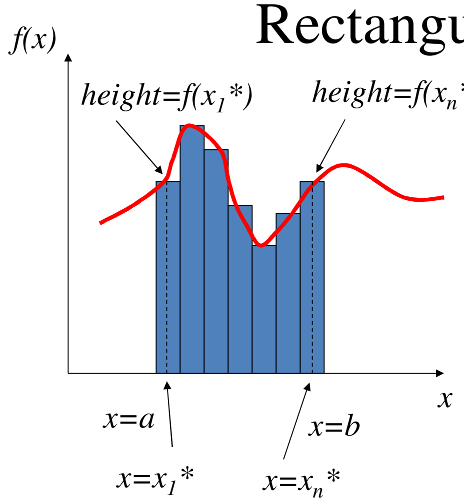
Numerical Integration

- In general, a numerical integration is the approximation of a definite integration by a "weighted" sum of function values at discretized points within the interval of integration.

$$\int_a^b f(x)dx \approx \sum_{i=0}^{N} w_i f(x_i)$$

where w_i is the weighted factor depending on the integration schemes used, and $f(x_i)$ is the function value evaluated at the given point x_i

Rectangular Rule



Approximate the integration, $\int_a^b f(x)dx$, that is the area under the curve by a series of rectangles as shown.
The base of each of these rectangles is $\Delta x=(b-a)/n$ and its height can be expressed as $f(x_i^*)$ where x_i^* is the midpoint of each rectangle

$$\int_a^b f(x)dx = f(x_1^*)\Delta x + f(x_2^*)\Delta x + ..f(x_n^*)\Delta x$$
$$= \Delta x[f(x_1^*) + f(x_2^*) + ..f(x_n^*)]$$

rect()

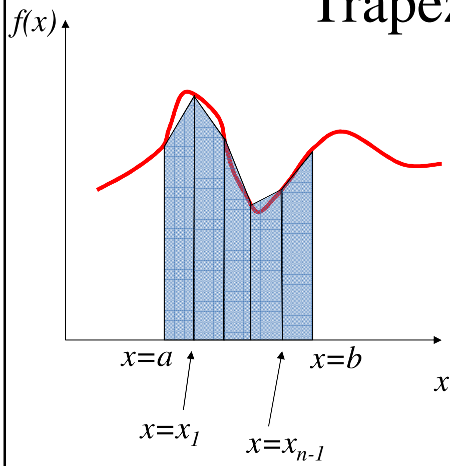
```
// rect() - Uses the Rectangle's rule to find the
//          definite integral of f(x).  Takes the
//          bounds as parameters
//          Uses f(x) that appears below.
float rect(int lowBound, int hiBound){
      int numDivisions = 4;
      float x, increment, integral = 0.0;

      // Get the increment and the midpoint of
      //the first rectangle
      increment = (float)(hiBound-lowBound)
                        / (float) numDivisions;
      x = lowBound + increment / 2.0;
```

```
      // Calculate f(x) and increment x to the
      // next value
      for (int i = 0; i < numDivisions; i++){
            integral = integral + f(x);
            x += increment;
      }

      // Multiply the sum by delta x
      integral = integral / (float) numDivisions;
      return (integral);
}
```

Trapezoidal Rule



The rectangular rule can be made more accurate by using trapezoids to replace the rectangles as shown. A linear approximation of the function locally sometimes work much better than using the averaged value like the rectangular rule does.

$$\int_a^b f(x)dx = \frac{\Delta x}{2}[f(a)+f(x_1)] + \frac{\Delta x}{2}[f(x_1)+f(x_2)] + .. + \frac{\Delta x}{2}[f(x_{n-1})+f(b)]$$

$$= \Delta x[\frac{1}{2}f(a)+f(x_1)+..f(x_{n-1})+\frac{1}{2}f(b)]$$

`trapezoid()`

```
// trapezoid() -  Uses the Trapezoid rule to find
//                the definite integral of f(x)
//                Takes the bounds as parameters
//                Uses f(x) that appears below.
float trapezoid(int lowBound, int hiBound){
      int   numDivisions = 4;
      float x, increment, integral = 0.0;

      increment = (float)(hiBound-lowBound)
                          / (float) numDivisions;
      x = lowBound;

      // Add f(lowBound)/2 to the sum
      integral = 0.5*f(x);
```

```
        //    Increment x to the next value,
        //    calculate f(x) and add it to the sum
        for (int i = 1; i < numDivisions; i++)    {
              x += increment;
              integral = integral + f(x);
        }
        // Add f(hiBound)/2
        integral = integral + 0.5*f(hiBound);

        // Multiply the sum by delta x
        integral = integral /(float) numDivisions;
        return (integral);
}
```

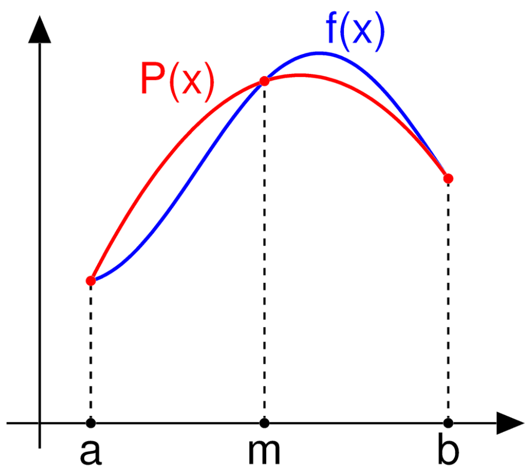
Simpson's Rule

Still, the more accurate integration formula can be achieved by approximating the local curve by a higher order function, such as a quadratic polynomial. This leads to the Simpson's rule and the formula is given as:

$$\int_a^b f(x)dx = \frac{\Delta x}{3}[f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + ..$$
$$..2f(x_{2m-2}) + 4f(x_{2m-1}) + f(b)]$$

It is to be noted that the total number of subdivisions has to be an even number in order for the Simpson's formula to work properly.

Simpson's Rule – A Quadratic Interpolation



simpson.c

```
#include    <iostream>

using namespace std;

float f(float x);
float simpson(int lowBound, int hiBound);

// main() - Get inputted values for lower and upper
//          bounds of integration, calls simpson()
//          to use Simpson's rule for numerical
//          integration and prints the result
```

```
int   main(void){
      int   lowBound, hiBound;
      float integral;

      // Input the bounds
      cout << "Enter lower bound\t?";
      cin >> lowBound;

      cout << "Enter upper bound\t?";
      cin >> hiBound;

      //Calls simpson and prints the integral
      integral = simpson(lowBound, hiBound);
      cout << "Integral is...." << integral;
      return(0);
}
```

```
// simpson() -    Uses Simpson's rule to find the
//                definite integral of f(x)
//                Takes the bounds as parameters
//                Uses f(x) that appears below.
float       simpson(int lowBound, int hiBound)  {
      int numDivisions = 4;
      float x, increment, integral = 0.0;

      increment = (float)(hiBound - lowBound)
                        / (float) numDivisions;
      x = lowBound;

      // Adds f(lowBound)
      integral = f(x);
```

```
    // Increment x to the next value, calculate
    // f(x)
    // Add 4f(x) for even numbered values
    // Add 2f(x) for odd numbered values
    for (int i = 1; i < numDivisions; i++){
        x += increment;
        if (i % 2 == 1)
            integral = integral + 4.0*f(x);
        else
            integral = integral + 2.0*f(x);
    }

    // Add f(hiBound)
    integral = integral + f(hiBound);
```

```
    // Multiply the sum by delta x/3
    integral = integral * increment/3.0;
    return (integral);
}

// f() -     The function being integrated
//           numerically
float f(float x){
    return(x * x * x);
}
```

Examples

Integrate $f(x) = x^3$ between $x = 1$ and $x = 2$.

$$\int_1^2 x^3 dx = \frac{1}{4}x^4 \Big|_1^2 = \frac{1}{4}(2^4 - 1^4) = 3.75$$

Using 4 subdivisions for the numerical integration: $\Delta x = \frac{2-1}{4} = 0.25$

Rectangular rule:

i	x_i^*	$f(x_i^*)$
1	1.125	1.42
2	1.375	2.60
3	1.625	4.29
4	1.875	6.59

$$\int_1^2 x^3 dx$$
$$= \Delta x[f(1.125) + f(1.375) + f(1.625) + f(1.875)]$$
$$= 0.25(14.9) = 3.725$$

Trapezoidal Rule

i	x_i	$f(x_i)$
	1	1
1	1.25	1.95
2	1.5	3.38
3	1.75	5.36
	2	8

$$\int_1^2 x^3 dx$$
$$= \Delta x[\frac{1}{2}f(1) + f(1.25) + f(1.5) + f(1.75) + \frac{1}{2}f(2)]$$
$$= 0.25(15.19) = 3.80$$

Simpson's Rule

$$\int_1^2 x^3 dx = \frac{\Delta x}{3}[f(1) + 4f(1.25) + 2f(1.5) + 4f(1.75) + f(2)]$$
$$= \frac{0.25}{3}(45) = 3.75 \Rightarrow \text{perfect estimation}$$

Transforms

- Transform:
  - In mathematics, a function that results when a given function is multiplied by a so-called kernel function, and the product is integrated between suitable limits. (*Britannica*)
  - $G(y) = \int_{x_1}^{x_2} F(x)\underbrace{K(x,y)}_{Kernel}dx$

Fourier Transform

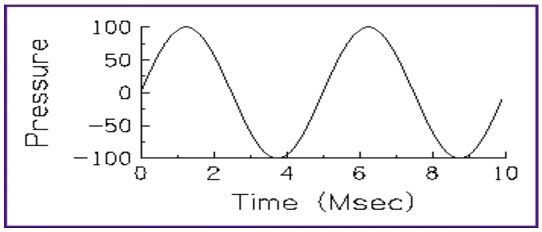
- Property of transforms:
  - They convert a function from one domain to another with no loss of information
- Fourier Transform:

$$F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t}\, dt$$

converts a function from the time (or spatial) domain to the frequency domain

Time Domain and Frequency Domain

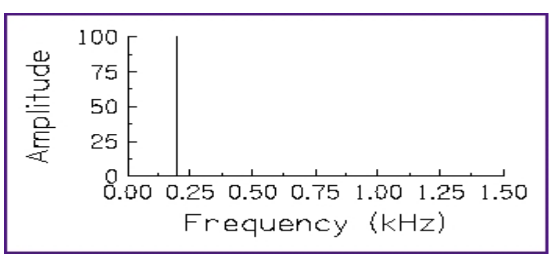
- Time Domain:
  - Tells us how properties (air pressure in a sound function, for example) change over time:



    - Amplitude = 100
    - Frequency = number of cycles in one second = 200 Hz

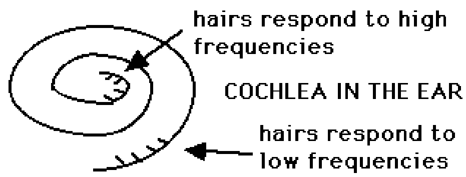
Time Domain and Frequency Domain

- Frequency domain:
  - Tells us how properties (amplitudes) change over frequencies:



Time Domain and Frequency Domain

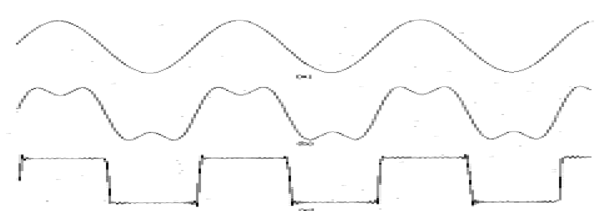
- Example:
  - Human ears do not hear wave-like oscilations, but constant tone



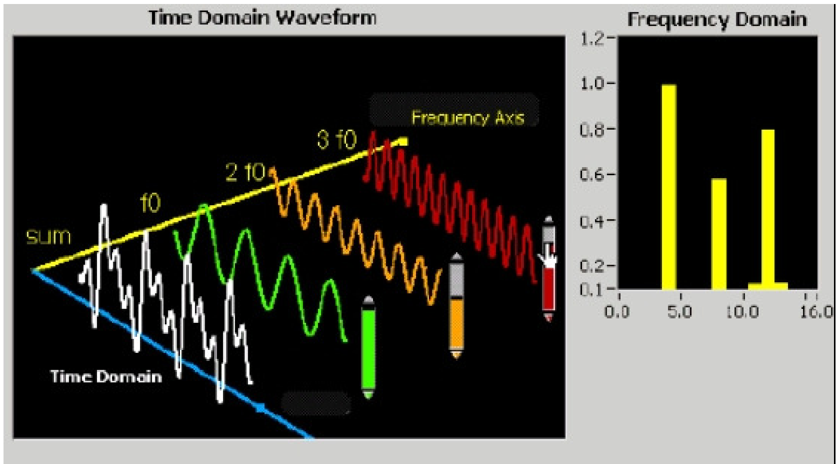
- Often it is easier to work in the frequency domain

Time Domain and Frequency Domain

- In 1807, Jean Baptiste Joseph Fourier showed that any periodic signal could be represented by a series of sinusoidal functions



Time Domain and Frequency Domain



Fourier Transform

- Because of the property:

EULER's FORMULA

$$e^{i\theta} = \cos\theta + i\sin\theta$$

$$e^{i\omega t} = \cos\omega t + i\sin\omega t$$

$$\text{where } i = \sqrt{-1}$$

- Fourier Transform takes us to the frequency domain:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt$$

the Fourier transform; strength of frequency ω contained in f(t)

scale factor for the Fourier Transform F(ω); the original signal in the time domain; the "inverse Fourier transform".

sinusoidally varying "basis" function for the expansion

Discrete Fourier Transform

- In practice, we often deal with discrete functions (digital signals, for example)
- Discrete version of the Fourier Transform is much more useful in computer science:
  - $W \equiv e^{2\pi i/N}$
  - $F_n = \sum_{k=0}^{N-1} W^{nk} f_k$, n = 0, 1, 2, … N-1
- Calculating all the values of the vector F requires $O(n^2)$ time complexity

Effect of Sampling in Time and Frequency

- By sampling in time, we get a periodic spectrum with the sampling frequency f_s. The approximation of a Fourier transform by a DFT is reasonable only if the frequency components of x(t) are concentrated on a smaller range than the Nyquist frequency $f_s/2$

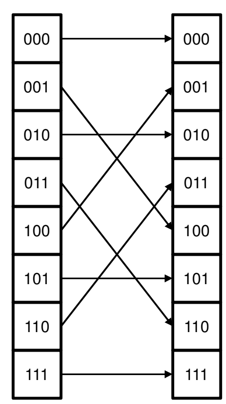
Dividing the Transform in 2

- $F_k = \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j$
- $= \sum_{j=0}^{\frac{N}{2}-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{\frac{N}{2}-1} e^{2\pi ik(2j+1)/N} f_{2j+1}$
- $= \sum_{j=0}^{\frac{N}{2}-1} e^{2\pi ikj/(\frac{N}{2})} f_{2j} + W^k \sum_{j=0}^{\frac{N}{2}-1} e^{2\pi ikj/(\frac{N}{2})} f_{2j+1}$
- $= F_k^e + W^k F_k^o$

This can be used recursively.

As We Continue To Divide…

- This works best if $N=2^n$
- We re-order the elements in the array.
  - Let $e = 0$ and $o = 1$
  - We reverse the bits



Now, we combine them..

- We start with our Fourier transforms of length one and we perform $\log_2 N$ combinations

The Fast Fourier Transform

```
//  Replaces data by its discrete Fourier transform
//  if sign is input as 1.
//  Replaces data by its inverse discrete Fourier
//  transform is sign is input as -1.
//  data is an array of complex values with the real
//  component stored in data[2j] and the imaginary
//  component stored in data[2j+1]
//  nn MUST be a power of 2; it is NOT checked.
void  four1(double data[], int nn, isign) {
  int i, istep, j, m, mmax, n;
  double    tempi, tempr;
  double    theta, wi, wpi, wpr, wr, wtemp;

  n = 2 * nn;
  j = i
```

```
// Do the bit reversal
for (i = 1; i <= n; i+=2)   {
  if (j > i)    {
    // Swap 2 complex values
    tempr = data[j];
    tempi = data[j+1];
    data[j] = data[i];
    data[j+1] = data[i+1];
    data[i] = tempr;
    data[i+1] = tempi;
  }

  m = n/2;
  while (m >= 2 && j > m)   {
    j = j - m;
    m = m / 2;
  }
```

```
  j = j + m;
}

// Here is where we combine terms
//  outer loop is performed log2 nn times
while (n > mmax)      {
  istep = 2 * mmax
  //Initialize trig recurrence
  theta = 2.0 * 3.141592653589/(isign*mmax);
  wpr = 2 * pow(sin(0.5*theta), 2);
  wpi = sin(theta);
  wr = 1.0;
  wi = 0.0;
```

```
// First of two nested loops
for (m = 1;  m <= mmax; m +=2)  {
  //Second of two nested loops
  for (i = m; i <= n;  i +=istep)     {
    // We combine them here
    j = i + mmax;
    tempr = wr* data[j] - wi *data[j+1];
    tempi = wr* data[j+1] + wi *data[j];
    data[j] = data[i] _ tempr;
    data[j+1] = data[i+1] - tempi;
    data[i] = data[i] + tempr;
    data[i+1] = data[i+1] + tempi;
  }
```

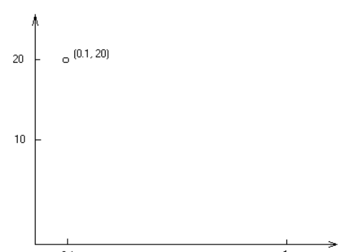
```
      // Trig recurrence
      wtemp = wr;
      wr = wr*wpr - wi*wpi + wr;
      wi = wi*wpi + wtemp*wpi + wi;
    }
    max = istep;
  }
}
```

Applications

- In image processing:
  - Instead of time domain: *spatial domain* (normal image space)
  - *frequency domain:* space in which each image value at image position F represents the amount that the intensity values in image I vary over a specific distance related to F

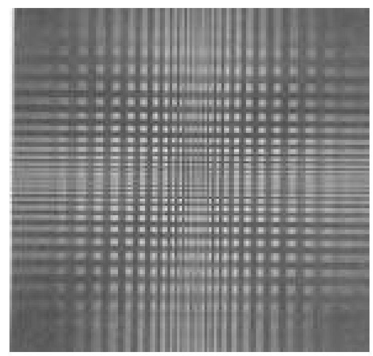
Applications: Frequency Domain In Images

- If there is value 20 at the point that represents the frequency 0.1 (or 1 period every 10 pixels). This means that in the corresponding spatial domain image I the intensity values vary from dark to light and back to dark over a distance of 10 pixels, and that the contrast between the lightest and darkest is 40 gray levels



Applications: Frequency Domain In Images

- *Spatial frequency* of an image refers to the rate at which the pixel intensities change
- In picture on right:
  - High frequences:
    - Near center
  - Low frequences:
    - Corners



Applications: Image Filtering

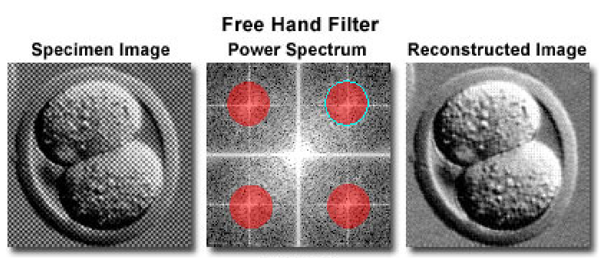


Figure 1