McGill

COMP 208 – Computers in Engineering

COMP 208
Computers in Engineering

Lecture 20

Jun Wang
School of Computer Science
McGill University
Fall 2007

Merge Sort (Variation)

```
static void _merge_sort(int arr[], int size, int temporary[])
{
  int half = size / 2;

  if(size <= 1) return;

  _merge_sort(arr, half, temporary);

  _merge_sort(arr + half, size - half, temporary + half);

  merge(arr, half, arr + half, size - half, temporary);

  memcpy(arr, temporary, size * sizeof (int));

  return;
}
```

memcpy declared in string.h

The False Position Method

The secant method uses the most recent approximation and the previous one

Even if the root was bracketed, it may not remain bracketed

The False Position Method combines the secant method with bisection to guarantee that the root remains bracketed

Regula Falsi Method

- Regula falsi is another common name for "false position"
- It can be thought of as a refinement of bisection
- It starts with 2 points x1,x2, such that $f(x1)*f(x2) < 0$
- Instead of using the midpoint of the interval we use the secant to interpolate the value of the root

The False Position Method

```
double fixedpoint_rf(DfD f, double x1, double x2,
                     double tol){
   double f1 = f(x1), f2 = f(x2),
          slope = (f2 - f1) / (x2 - x1),
          distance = -f1 / slope,
          point = x1 + distance;

   if(fabs(f(point)) < tol)
       return point;
   if((f1 * f(point)) < 0)
       return fixedpoint_rf(f, x1, point, tol);
   else
       return fixedpoint_rf(f, point, x2, tol);
}
```

Newton-Raphson

- Newton's method starts with a single initial guess at the root, x_0
- It is like the secant method with the value of the derivative replacing the slope
- At each step, the next value is given by

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

where f′ is the derivative of f

using a tangent line instead of secant

Convergence Criteria

This method could use the third convergence condition to terminate

That is, it could terminates when x_i and x_j are very close to each other

In our implementation we use the first, that $f(x_i)$ is close to zero

Newton-Raphson

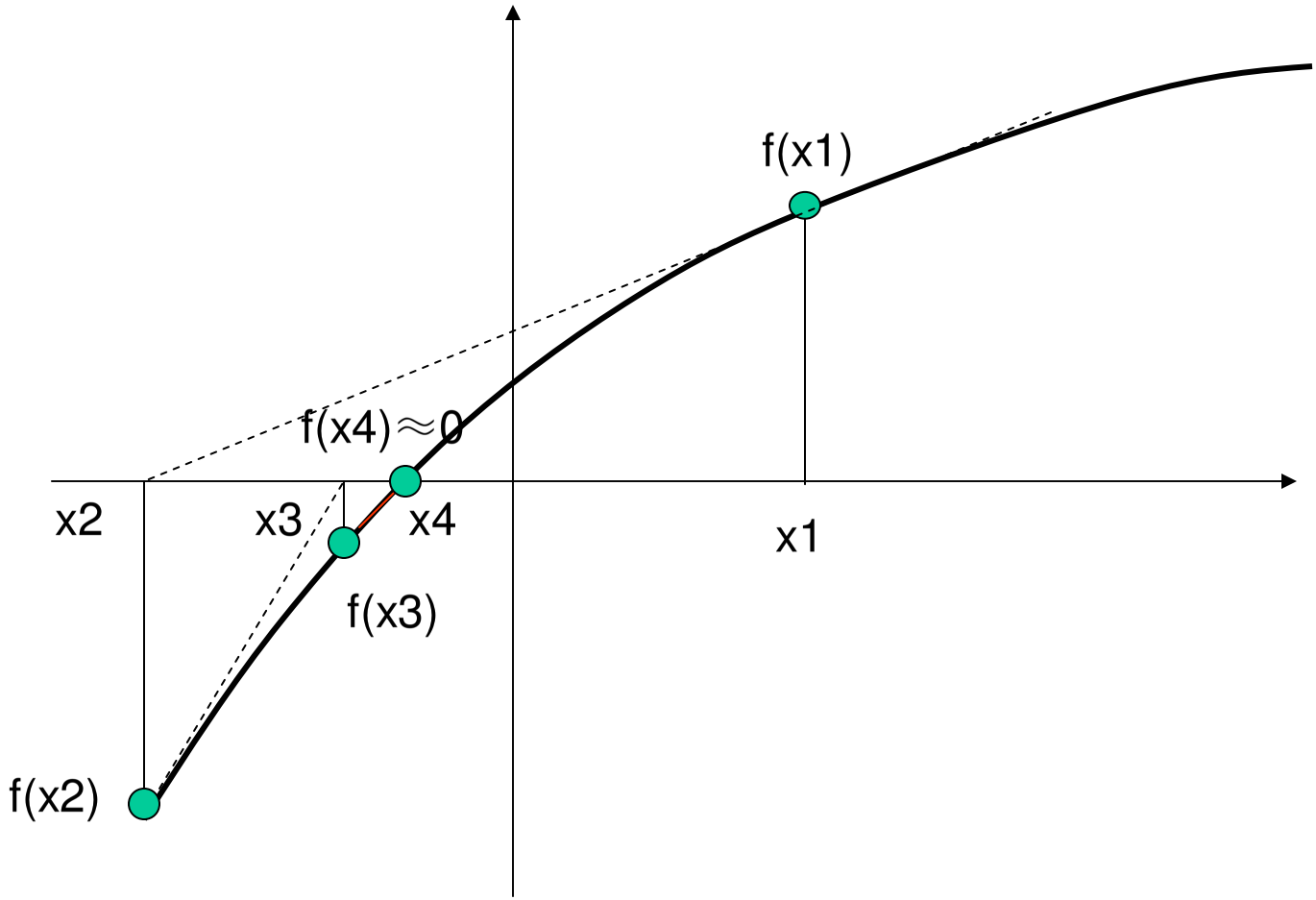
For example, to compute the square root of a number, a, we can find the root of the function x^2-a $(x_{i+1} = (x_i + a/x_i)/2)$

Newton's method guarantees that we keep getting closer to the root

However, in general we may not converge to a root

Therefore we stop after a certain number of steps (count) if we have not yet found a root

Newton-Raphson



Newton_Raphson

```
double newton_rf(DfD f, DfD df,
                 double x,
                 double tol, int count){

  double distance = -f(x) / df(x);
  x += distance;
  if((fabs(f(x)) < tol) || !count)
      return x;
  return newton_rf(f, df, x, tol, count - 1);
}
```

The Derivative

This algorithm assumes that we know the derivative of the function

If the function is complex or we generate values of the function empirically without having an explicit analytic form for the function, we may have to estimate the derivative

Using a "centered three point" method, we can rewrite the algorithm as follows

Newton-Raphson with Numerical Differentiation

```
double newton_rf(DfD f, double x,
                 double tol, int count){

  double distance =
          -f(x) / centered3_diff(f,x,1e-6);
  x += distance;
  if((fabs(f(x)) < tol) || !count)
     return x;
  return newton_rf(f, x, tol, count - 1);
}
```

Numerical Differentiation

- Engineers often deal with functions represented as a collection of data points
- We might not have an analytic closed form for the function
- For example, we might measure the position of a vehicle at different points in time

Numerical Differentiation

- Given the position of a vehicle at different points in time:
  - To compute the velocity, i.e. the derivative, we must compute an estimate based on the observed position
  - Knowing the value of the function at two different points in time allows us to approximate the derivative

Two Point Approximations

Central Difference Formula:

```
f'(t) = (f(t+h)-f(t-h))/2h
```

Forward Difference Formula:

```
f'(t) = (f(t+h)-f(t))/h
```

Backward Difference Formula:

```
f'(t) = (f(t)-f(t-h))/h
```

Two Point Differentiation

- The error is O(h)
- If h is big, the approximation will not be very accurate
- If h is small, there may be large roundoff errors.
- It might not be possible to sample the data at intervals for which h is very small

Three Point Differentiation

Three point methods are more precise

We look at the formulae for three methods but in this course we will not derive them

- Forward 3 point differentiation
- Backward 3 point differentiation
- Centered 3 point differentiation

Forward three point

$$\frac{dy}{dx} = \frac{-3*f(x)+4*f(x+\Delta x)-f(x+2*\Delta x)}{2*\Delta x}$$

```
double forward3_diff(DfD f, double x,
                     double h){
  return (-3*f(x) + 4*f(x+h) - f(x+2*h)) /
         (2 * h);
}
```

Backward three point

$$\frac{dy}{dx} = \frac{f(x-2*\Delta x) - 4*f(x-\Delta x) + 3*f(x)}{2*\Delta x}$$

```
double backward3_diff(DfD f, double x,
                      double h){
  return (f(x - 2*h) - 4*f(x - h) + 3*f(x))
         / (2 * h);
}
```

Centered three point

In deriving the centered three point formula, terms cancel out and the final result is the same as the centered two point

This has the fewest function evaluations and is as accurate as the 3 point methods

```
double centered3_diff(DfD f, double x,
                      double h){
  return (-f(x - h) + f(x + h)) / (2 * h);
}
```

McGill

COMP 208 – Computers in Engineering

Initial Value Problems

Nathan Friedman

Fall, 2007

Ordinary Differential Equations

In many areas of application we can measure how things change in some process

From these measurements we would like to find the function that describes the process

Examples:

- Change in concentrations during chemical reaction
- Heating or cooling of objects
- Current flow in electrical circuits
- Population dynamics

ODE's

If we let y(x) be the function we would like to study, we are able to observe the rate of growth of the function

This leads to equations of the form

```
dy/dx = f(x,y)
```

ODE's

- An **ordinary differential equation** takes the form

  $G(t, x(t), x'(t), x''(t), ...) = 0$ for all t,

- Example
  - Here are some examples of ordinary differential equations:
  - $x'(t) - 1 = 0$ for all t (first-order)
  - $x'(t) + x(t) - 1 = 0$ for all t (first-order)
  - $x''(t) - 1 = 0$ for all t (second-order)
  - $x''(t) - 2\,t\,x(t) = 0$ for all t (second-order)

Solution to ODE

- A **solution** of an ordinary differential equation is a *function x* that satisfies the equation for all values of *t*.
- Example:
  - A solution of the equation $x'(t) - 1 = 0$, for example, is the function x defined by $x(t) = t$ for all t.
  - There are many solutions. In general: $x(t) = t + c$ (c is a constant)

Initial value problem

- If $x'(t) - 1 = 0$ is given an initial value $x(0)=4$, then the only solution is $x(t) = t + 4$
- The additional conditions on the values of the function are referred to as *initial conditions* (though these conditions may specify the value of *x* or the value of its derivatives at any value of *t*, not necessarily the "first" value).
- A differential equation together with a set of initial conditions is called an **initial value problem**

Analytic Solutions

Some ODE's have analytic solutions

$$dy/dx = x + y - 1$$

Has the solution

$$y(x) = e^x - x$$

Others have no analytic solution. For example:

$$dy/dx = x^2 + y^2$$

Initial Value Problems

When an ODE cannot be easily solved, we can use numerical methods to approximate the solution

The methods we consider are called initial value problems since we must know the value of the function, y_0, at some initial point x_0

The Euler Method

- We "grow" the function from the starting value one step at a time
- Think of the `dy/dx` in terms of discrete steps, `delta_y` and `delta_x`
- Then the derivative approximates the ratio of these two values for small values of delta_x
  i.e. y' is approximately dy/dx

The Euler Method

- Multipling `dy/dx` by `delta_x` gives an approximate value for `delta_y`
- The Euler method increments the independent variable by one stepsize, `delta_x`, at a time
- Using the derivative, we approximate `delta_y` and then the value of the function at the next step

Euler Method

Given:

```
dy/dx = f(x,y)      (e.g. y'=x + y -1)
y(x0) = y0          (e.g. y(0) = 1)
```

Since it's not easy to solve the OED to find y(x), we find the approximate value of y at x0+h, x0+2h, x0+3h, ...

Now `f(x0,y0)` is the slope of the function at `(x0,y0)`

Approximate the function value at `x0+h` by

```
y0 + h*f(x0,y0)
```

Repeat this process so that

```
x_(n+1) = x_n + h
y_(n+1) = y_n + hf(x_n,y_n)
```

A Simple Example

```
// approximate y, given y' = x+y-1, and x0=0, y0=1
// The analytical solution is y=exp(x)-x

#include <stdio.h>
#include <math.h>
#define H    0.1
int main()
{
   double x, newx;
   double y, newy;
   int i, steps = 10;

   x = 0;
   y = 1;
   for (i = 0; i < steps; i++)
   {
      newx = x + H;
      newy = y + H * (x + y -1); //newy = y + H*y'
      x = newx; y = newy;
      printf ("  %f  %f  %f\n", newx, newy,exp(x)-x);
   }
   return 0;
}
```

`#define`

- After the `#include` commands, we can also define names using `#define`
- In the program, the name is replaced by the expression
- The name defined is called a macro, and the replacement is called macro substitution
- Note that the name is not a variable (that is the name of a memory location)
- The name is just an alias for the value

#define

- Syntax
  - No ; at end
  - No = between name and value

```
#define macro expression
```

- Like #include, this is also a pre-processor directive
- A most common use is to define constants
- Every occurrence of macro is replaced with the expression by the pre-processor
- This can cause problems if complex expressions are used

Example Revisited

```
void main(){
   double x, y;
   FILE * myfile = fopen("test2.csv","w+");
   int i,steps = 100;
   if (myfile){
      x = 0;
      y = 1;
      for (i = 0; i < steps; i++){
         fprintf (myfile," %f, %f, %f\n", x, y,exp(x)-x);
         Euler_step(H,x,&y,func);
         x += H;
      }
      fclose(myfile);
   }
   else printf("Could not open the file");
      return;
}
```

Writing to a File – A Review

We declare it to be of type FILE and use fopen to open it.

```
FILE * myfile = fopen("test2.csv","w+");
```

Then we can write to the file using fprintf

```
fprintf (myfile," %f, %f, %f\n",
          x, y,exp(x)-x);
```

When we are finished we close the file

```
fclose(myfile);
```

"w+": open for reading and writing;
If file does not exist, create it.
If file does exist, empty it.

Writing to a File

When we open a file, the filename is assigned a nonzero value if the operation is successful and a value of zero if it fails.

```
if (myfile)
    {
          . . .
    }
  else printf("Could not open the file");
```

Function Arguments – A Review

We can pass a function as an argument to a function that computes the next iteration from the previous one.

First give a name (fun) to the **type** of function we want to use

```
typedef double (*fun) (double,double);
```

Computing the Next Value

Then define a function that uses this argument to compute the next value.

We compute a new value for y, so we pass a pointer to it

```
void Euler_step(double h, double x,
                double *y, fun f){
  *y += h * f(x,*y);
  return;
}
```

Computing the Next Value

We can define a function that we want to pass to the Euler function.

```
double func(double x, double y){
  return x+y-1;
}
```

Functions are stored in the memory of the machine at a specific address. We can pass a pointer to the function.

```
Euler_step(H,x,&y,func);
```

The Final Program (a)

```
#include <stdio.h>
#include <math.h>

#define H    0.1

typedef double (*fun) (double,double);

double func(double x, double y){
   return x+y-1;
}

void Euler_step(double h, double x, double *y, fun f){
   *y += h * f(x,*y);
   return;
}
```

The Final Program (b)

```
int main(){
  double x, y;
  FILE * myfile = fopen("test2.csv","w+");
  int i,steps = 100;
    if (myfile){
      x = 0; y = 1;
      for (i = 0; i < steps; i++){
        fprintf (myfile," %f, %f, %f\n", x, y,exp(x)-x);
       Euler_step(H,x,&y,func);
        x += H;
      }
      fclose(myfile);
    }
  else printf("Could not open the file");
  return 0;
}
```

Part of the Output File

0	1	1
0.1	1.01	1.005171
0.2	1.031	1.021403
0.3	1.0641	1.049859
0.4	1.11051	1.091825
0.5	1.171561	1.148721
0.6	1.248717	1.222119
0.7	1.343589	1.313753
0.8	1.457948	1.425541
0.9	1.593742	1.559603
1	1.753117	1.718282
1.1	1.938428	1.904166
1.2	2.152271	2.120117
1.3	2.397498	2.369297
1.4	2.677248	2.6552
1.5	2.994973	2.981689

Excel Generated Graph