IYTE EE204 Lecture 2
Numerical Analysis

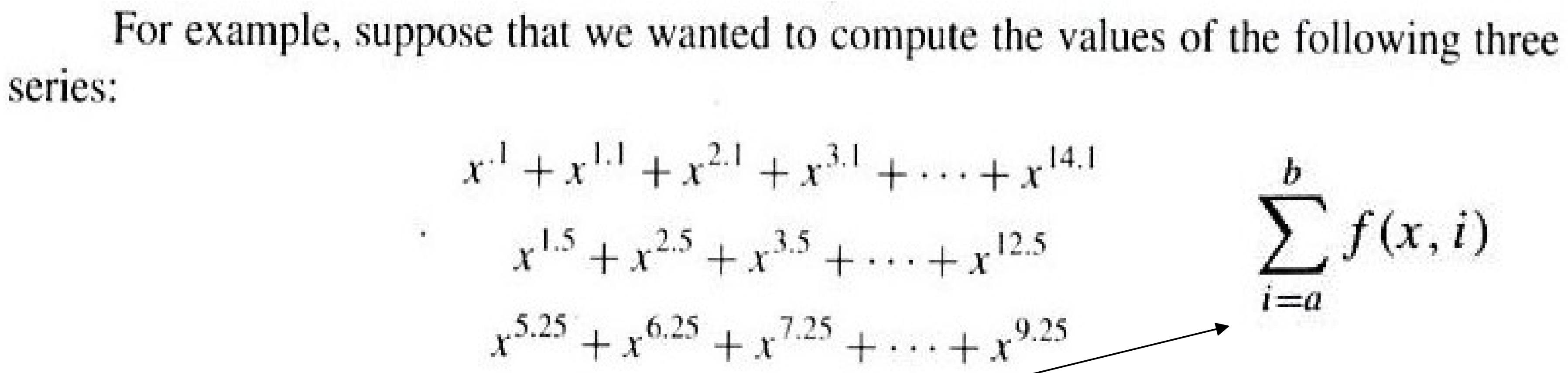
- Root Finding
- Integration
- Differential Equations

Şevket Gümüştekin
& Barış Bozkurt

Function pointers

Why would we need pointers to functions?

Imagine you have written a program that finds roots of a cost function that is derived in an optimization problem. You want to be able to use your program without changing it (keeping it independent of the cost function to be processed). Then you would like to call your program with an input that is a function.



For example, suppose that we wanted to compute the values of the following three series:

$$x^{.1}+x^{1.1}+x^{2.1}+x^{3.1}+\cdots+x^{14.1}$$

$$x^{1.5}+x^{2.5}+x^{3.5}+\cdots+x^{12.5}$$

$$x^{5.25}+x^{6.25}+x^{7.25}+\cdots+x^{9.25}$$

$$\sum_{i=a}^{b} f(x,i)$$

We would like to write a general summation function that accepts functions (which serve computation of powers) as input

Several examples will serve to demonstrate how pointers to functions are declared as arguments. Consider, for example, an argument **f** which is a function that has a **double** and an **int** as arguments and returns a **double**. This function might evaluate a single term in a series such as $x^{i+0.1}$. It would have two arguments: the values of x (in a **double**) and i (in an **int**). If this function is an argument to some other function, it would be declared in that other function's prototype as follows:

```
double (*f) (double, int)
```

Note the use of the parentheses in **(*f)()**. This is because the declaration

```
double *f(double, int)
```

declares **f()** to be a function that returns a *pointer to a* **double** rather than a pointer to a function that returns a **double**. The parentheses denoting a function have higher precedence than the *****. The use of the first pair of the parentheses in **(*f)()** overrides the normal order of precedence.

```
/* general-purpose summation routine */

double summation(
   double x,                       /* value of x */
   double (*f)(double, int),       /* function for individual
                                                        terms */
   int a,                          /* initial index for sum */
   int b)                          /* final index for sum */
{
  double sum;

  for(sum = 0.0; a <= b; a=a+1)
    sum = sum + f(x, a);
  return(sum);
}
```

The function prototype for **summation()** would be as follows:

```
double summation(double,double (*)(double,int),int,int);
```

```
\* example of program using summation */

#include <stdio.h> /* I/O header file */
#include <math.h>  /* math header file */
double summation(double,double (*)(double,int),int,int);
double f1(double,int), f2(double,int), f3(double,int);

main()
{
  double total,x;

  printf("Enter value of x for series:");
  scanf("%lf", &x);

  total = summation(x, f1, 0, 14);
  printf("First series sum is %lf\n", total);

  total = summation(x, f2, 1, 12);
  printf("Second series sum is %lf\n", total);

  total = summation(x, f3, 5, 9);
  printf("Third series sum is %lf\n", total);

}
```

```
/* terms for first series */

double f1(double x, int i)
{
  return( pow(x, i+.1));
}


/* terms for second series */

double f2(double x, int i)
{
  return( pow(x, i+.5));
}


/* terms for third series */

double f3(double x, int i)
{
  return( pow(x, i+.25));
}
```

This example uses the standard C mathematical library function `pow()`. This function has two `double`s as its arguments. `pow(x,y)` computes x^y.

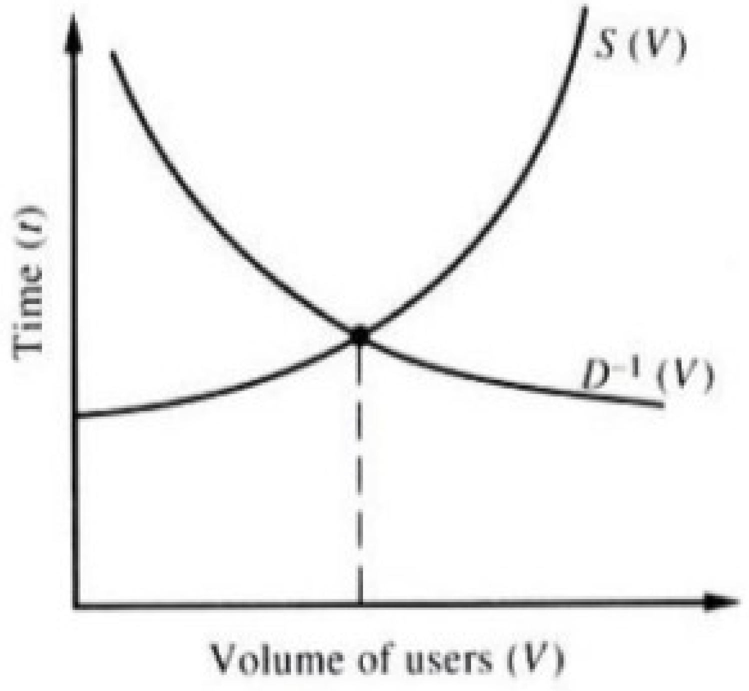
Some declarations may become complicated when we start using function pointers

```
int *f();        /* f: function returning pointer to int */

int (*pf)();   /* pf: pointer to function returning int */

char **argv
    argv:  pointer to pointer to char
int (*daytab)[13]
    daytab:  pointer to array[13] of int
int *daytab[13]
    daytab:  array[13] of pointer to int
void *comp()
    comp:  function returning pointer to void
void (*comp)()
    comp:  pointer to function returning void
char (*(*x())[])()
    x: function returning pointer to array[] of
    pointer to function returning char
char (*(*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
```

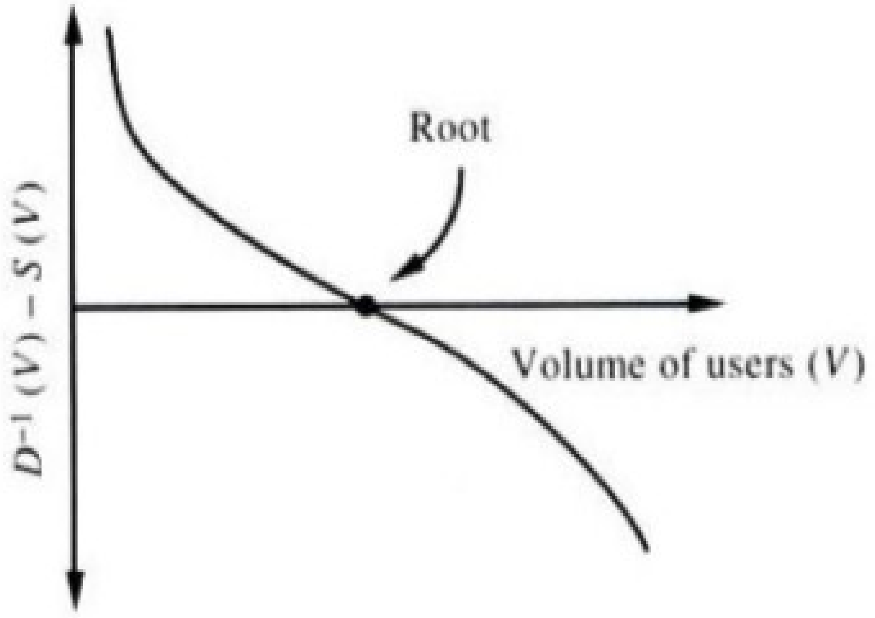
Root finding

Root finding is an important problem since many engineering optimization problems can be reduced to finding zeros of a cost function.

Example: Supply/Demand Cuves in economics. Interest point is the root.



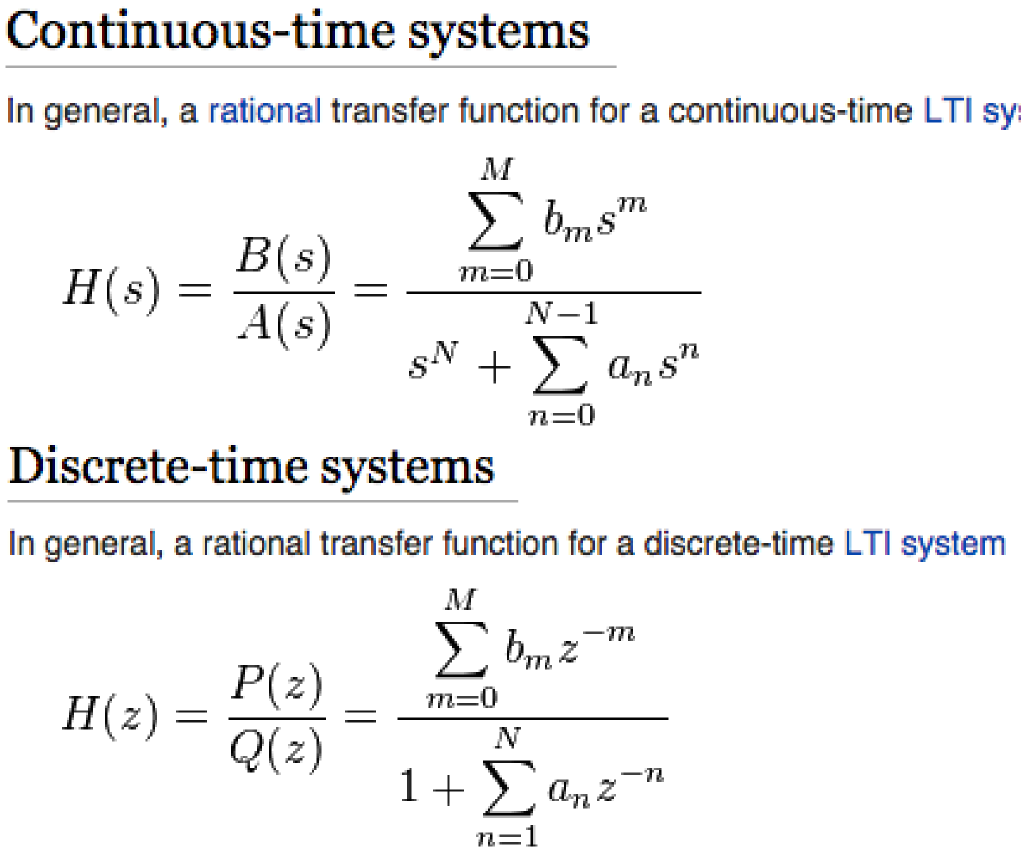
(a) Supply and inverse demand functions



(b) Difference between inverse demand and supply function

Root finding

For electrical engineers, the concept is familiar: poles of a transfer function are points where a system's response becomes infinite.Obtaining pole-zero plots for transfer functions is critical for control purposes.



Continuous-time systems

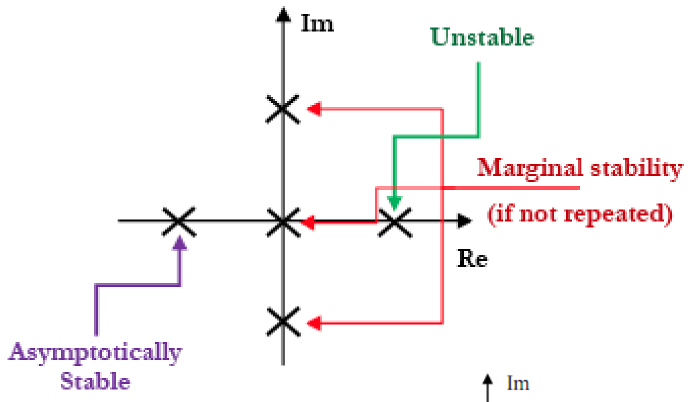
In general, a rational transfer function for a continuous-time LTI system has the form:

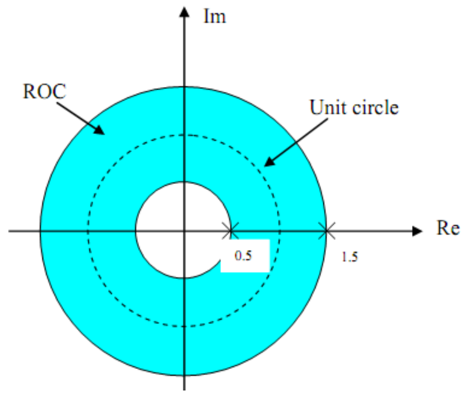
$$H(s) = \frac{B(s)}{A(s)} = \frac{\displaystyle\sum_{m=0}^{M} b_m s^m}{s^N + \displaystyle\sum_{n=0}^{N-1} a_n s^n}$$

Discrete-time systems

In general, a rational transfer function for a discrete-time LTI system

$$H(z) = \frac{P(z)}{Q(z)} = \frac{\displaystyle\sum_{m=0}^{M} b_m z^{-m}}{1 + \displaystyle\sum_{n=1}^{N} a_n z^{-n}}$$





Root-finding methods all share some common characteristics. In particular, the methods we will examine share the following:

1. They all require either a known interval $[x_1, x_2]$ in which a root is to be searched for or some initial starting value of x_0 from which a search is to begin.
2. None of the methods are guaranteed to find *all* the real roots of an equation. Instead, they find at most one root.
3. If they succeed in finding a root, they do so only approximately. The approximate nature of the solution is only in part because they rely on floating-point arithmetic. Another reason is that the methods are intrinsically iterative, converging upon a solution step by step. Even on a computer with infinite precision, the methods would not reach the true root in a finite number of steps.

Despite these limitations, numerical root-finding methods are often the only feasible means of solving many real-world problems.

3.5.1 Finding an Interval Containing a Root

Many of the algorithms that find a root of an equation require as one of their arguments an interval that contains a root. A simple algorithm for finding an interval that contains a root of a function $f(x)$ exploits the fact that any interval $[x_1, x_2]$ where

$$f(x_1) \cdot f(x_2) \leq 0$$

must contain at least one root. In fact, such an interval must contain an odd number of roots. To find such an interval, we take a broad range of x, divide it into subranges, and search for an interval that satisfies the condition above. A C function named **`find_interval()`** that does this search requires the following arguments:

- **`f`** — a pointer to the function for which an interval containing a root is to be found.
- **`xstart`** and **`xend`** — the starting and ending values of x defining the interval over which a search for an interval containing a root is to be done.
- **`stepsize`** — the size of the intervals to be tested.
- **`xleft`** and **`xright`** — pointers to variables of type **`double`** that point to the beginning and end of an interval containing a root.

```
/* function to find an interval containing a root */

#define TRUE 1
#define FALSE 0

int find_interval(
  double (*f)(double), /* function */
  double xstart,       /* start of search */
  double xend,         /* end of search */
  double stepsize,     /* interval of search */
  double  *xleft,      /* pointer to  left
                          of found interval */
  double *xright)      /* pointer to right
                          of found interval */
{
  int found_root = TRUE; /* initialize indicator */

  /* initialize interval */
  *xleft = xstart;
  *xright = *xleft + stepsize;

  /* search interval */
  while (*xleft < xend && f(*xleft) * f(*xright)>0.0) {
        *xleft = *xright;
        *xright += stepsize;
        if(*xright > xend)  *xright = xend;
      }

  /* check if root found */
  if(f(*xleft) * f(*xright)> 0.0)
        found_root = FALSE;

  return(found_root);
}
```

Use for function: $f(x) = x^3 - 4x^2 - 4x + 15$

```
/* program that uses find_interval() */

#include <stdio.h>
main()
{
  double polycubic(double);
  int find_interval(double (*f)(double), double, double,
                              double, double *, double *);
  double x1, x2;

  if(find_interval(polycubic, -10.0, 10.0,
                          0.1, &x1, &x2))
       printf("Interval is %lf to %lf\n", x1, x2);
  else
       printf("No interval was found\n");
}

/* function whose root to find */
double polycubic(double x)
{
  return(x*x*x - 4*x*x - 4*x + 15);
}
```

3.5.2 Bisection Method

Assuming that an interval containing a root has been found, the most straightforward approach to root finding is called *bisection*. This method searches over a given interval known to contain at least one root and successively halves that interval, restricting the search to the half known to contain the root. This successive halving converges to a very small interval in which a root must lie. The algorithm terminates when the interval is "small enough" by some criterion. What constitutes an interval which is small enough should be controlled by the user of the algorithm. However, it makes no sense to attempt to use bisection to obtain an interval which, as a fraction of the initial interval, is smaller than the machine accuracy.

Bisection is a classic example of a computational strategy often referred to as *divide and conquer*. The initial interval $[a, b]$ is successively halved, implying that at the nth iteration the size of the interval is $(b - a)/2^n$. Such algorithms converge very quickly to intervals that are small. If the floating-point representation of a `double` has d significant digits in binary form, then bisection will never require more than d iterations to converge to an interval that, as a fraction of $(b - a)$, is as small as the machine accuracy.

Bisection can be implemented either recursively or iteratively. The iterative implementation is as follows:

```
/* bisection method */
#include <math.h>
double bisect(
  double (*f)(double), /* function */
  double x1, /* start of interval */
  double x2, /* end of interval */
  double epsilon) /* convergence tolerance */
{
  double y;
  for(y=(x1+x2)/2.0; fabs(x1-y)>epsilon; y=(x1+x2)/2.0)
    if(f(x1)*f(y)<= 0.0)
        x2 = y; /* use left subinterval */
    else
        x1 = y; /* use right subinterval */
  return(y);
}
```

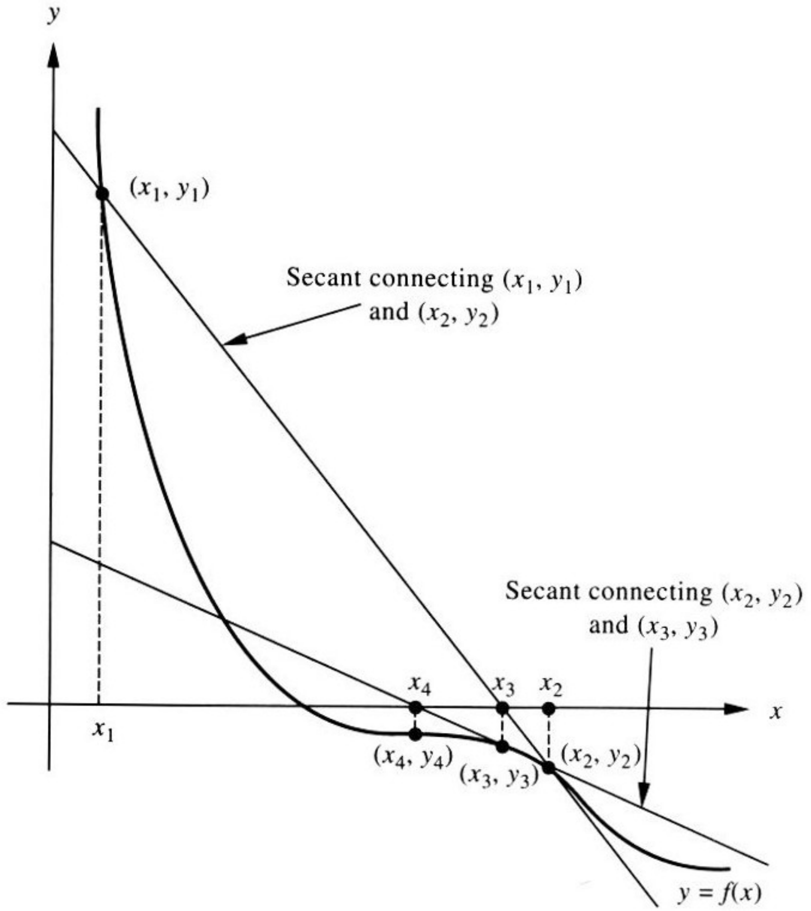
The function **`fabs()`** is a standard C mathematical library function. It takes a single argument of type **`double`** and returns a **`double`** which is the absolute value of the original argument. The prototype for **`fabs()`** is part of the header file **`math.h`**.

The recursive implementation of bisection is as follows:

```
/* recursive version of bisection */
#include <math.h>

double rbisect(
  double (*f)(double), /* function */
  double x1,          /* start of interval */
  double x2,          /* end of interval */
  double epsilon)     /* tolerance */
{
  double y;

  y = (x1+x2)/2.0;
  if( fabs(x1-y) > epsilon) /* check for convergence */
    if(f(x1)*f(y) <=0)
      y = rbisect(f,x1,y,epsilon);/* use left side */
    else
      y = rbisect(f,y,x2,epsilon);/* use right side */
  return(y);
}
```

3.5.3 Secant Method

The *secant method* uses an entirely different approach to root finding than the bisection method. The algorithm works by fitting a straight line between the points $(x_1, f(x_1))$ and $(x_2, f(x_2))$. It then finds x^*, the root of the straight line. If the result is sufficiently close to x_2, it returns that value as the root of the original equation. If not, it resets the value of x_1 to x_2 and x_2 to x^*, and tries the method again. This proceeds until a solution close to x_2 is found or some number of attempts to find a solution have failed.



y
(x_1, y_1)
Secant connecting (x_1, y_1) and (x_2, y_2)
Secant connecting (x_2, y_2) and (x_3, y_3)
x_4
x_3
x_2
x
x_1
(x_4, y_4)
(x_3, y_3)
(x_2, y_2)
$y = f(x)$

Figure 3.3 Secant method.

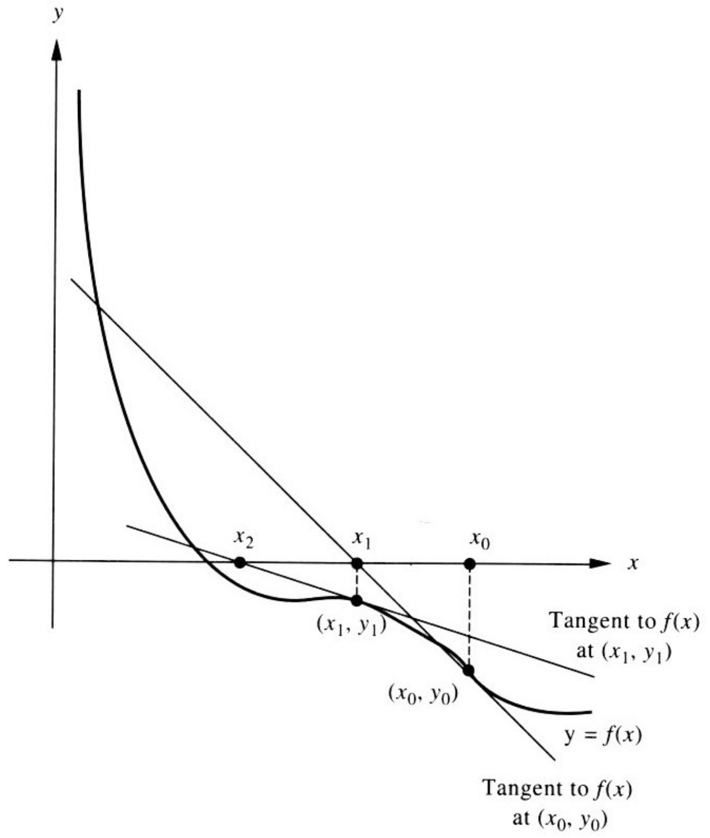
```
/* implementation of the secant method */
#include <math.h>
#define TRUE 1
#define FALSE 0


double secant(
  double (*f)(double),   /* function to find root of */
  double x1,             /* start of interval */
  double x2,             /* end of interval */
  double epsilon,        /* tolerance */
  int max_tries,         /* maximum number of tries */
  int *found_flag)       /* pointer to flag indicating
                                                success */
{


  int count = 0;          /* counter of loops */
  double root = x1;       /* root value */
  *found_flag = TRUE;     /* initialize flag to success */

  while(fabs(x2-x1)>epsilon && count<max_tries) {
    root = x1 - f(x1)*(x2-x1)/(f(x2)-f(x1));
    x1=x2;
    x2 = root;
    count = count + 1;
  }

  if(fabs(x2-x1) > epsilon)/* check for convergence */
    *found_flag = FALSE;

  return(root);           /* return root */

}
```

3.5.4 Newton's Method

Unlike bisection and the secant method, Newton's method does not require an initial interval. Instead, it begins the search for a root with some initial guess. It then fits a tangent to $f(x)$ through the guess and finds the root of the tangent. This then becomes the next guess. This process continues until the difference between the guess and the root of the tangent line is small.

Newton's method uses the derivative of $f(x)$ to find the slope of the tangent line. Thus, it requires two functions as inputs: the original function $f(x)$ and the derivative of $f(x)$, which we will denote as $f'(x)$. Each iteration of Newton's method is simply

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

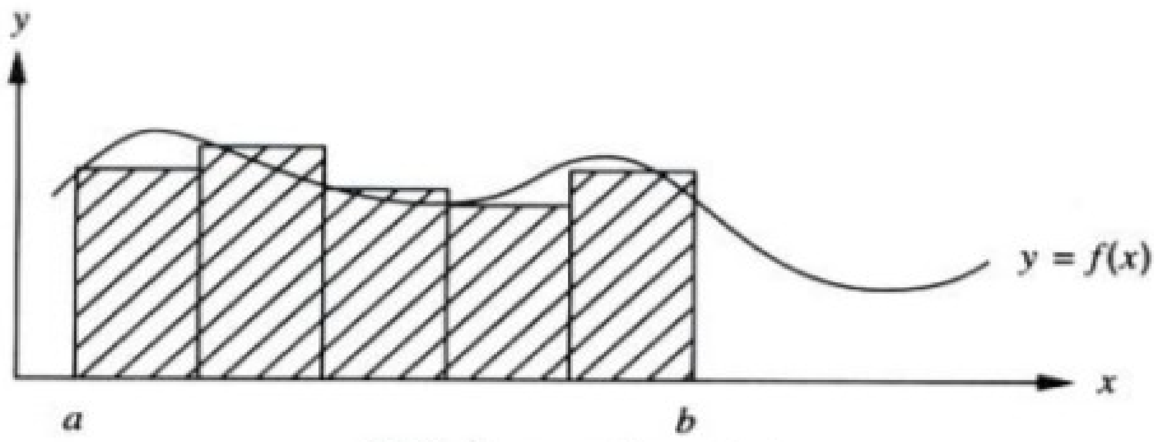


y
x_2
x_1
x_0
x
(x_1, y_1)
Tangent to $f(x)$ at (x_1, y_1)
(x_0, y_0)
$y = f(x)$
Tangent to $f(x)$ at (x_0, y_0)

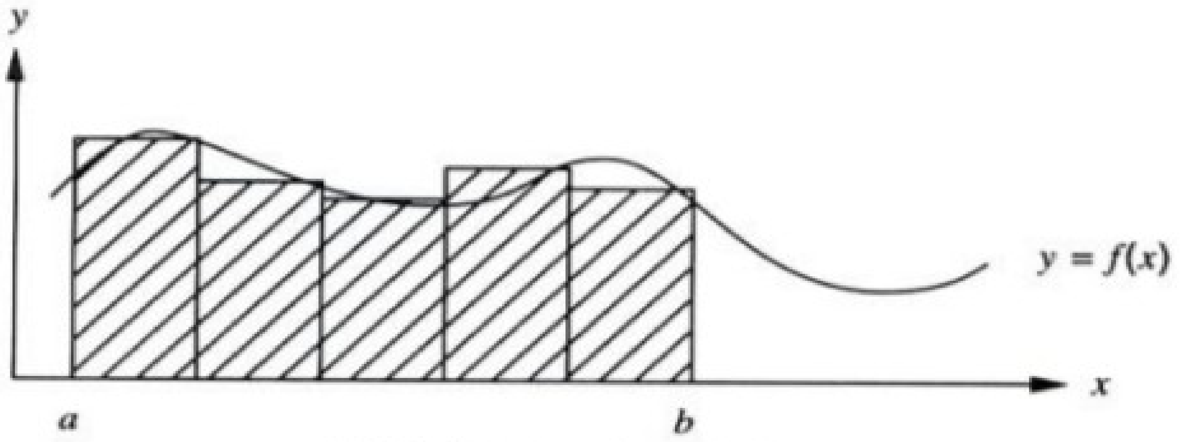
```
/* implementation of Newton's method */

#include <math.h>
#define TRUE 1
#define FALSE 0

double newton(
  double (*f)(double),      /* function */
  double (*fprime)(double),/* derivative of f */
  double dmin,                /* minimum allowed value of
                                          fprime */
  double x0,             /* initial guess of solution */
  double epsilon,        /* convergence tolerance */
  int *error)            /* pointer to error indicator */
{
  double deltax;
  deltax = 2.0 * epsilon; /*initialize deltax>epsilon */
  *error = FALSE;          /* initialize error indicator */

  while( !(*error) && fabs(deltax) > epsilon)
    if(fabs(fprime(x0)) >dmin)  {
      deltax = f(x0)/fprime(x0);
      x0 = x0 - deltax;
  }
    else
      *error = TRUE;

  return(x0);
}
```

Numerical integration

3.7.1 Rectangular Rule



(b) Left rectangle method



(c) Right rectangle method

Numerical integration

3.7.2 Trapezoidal Rule

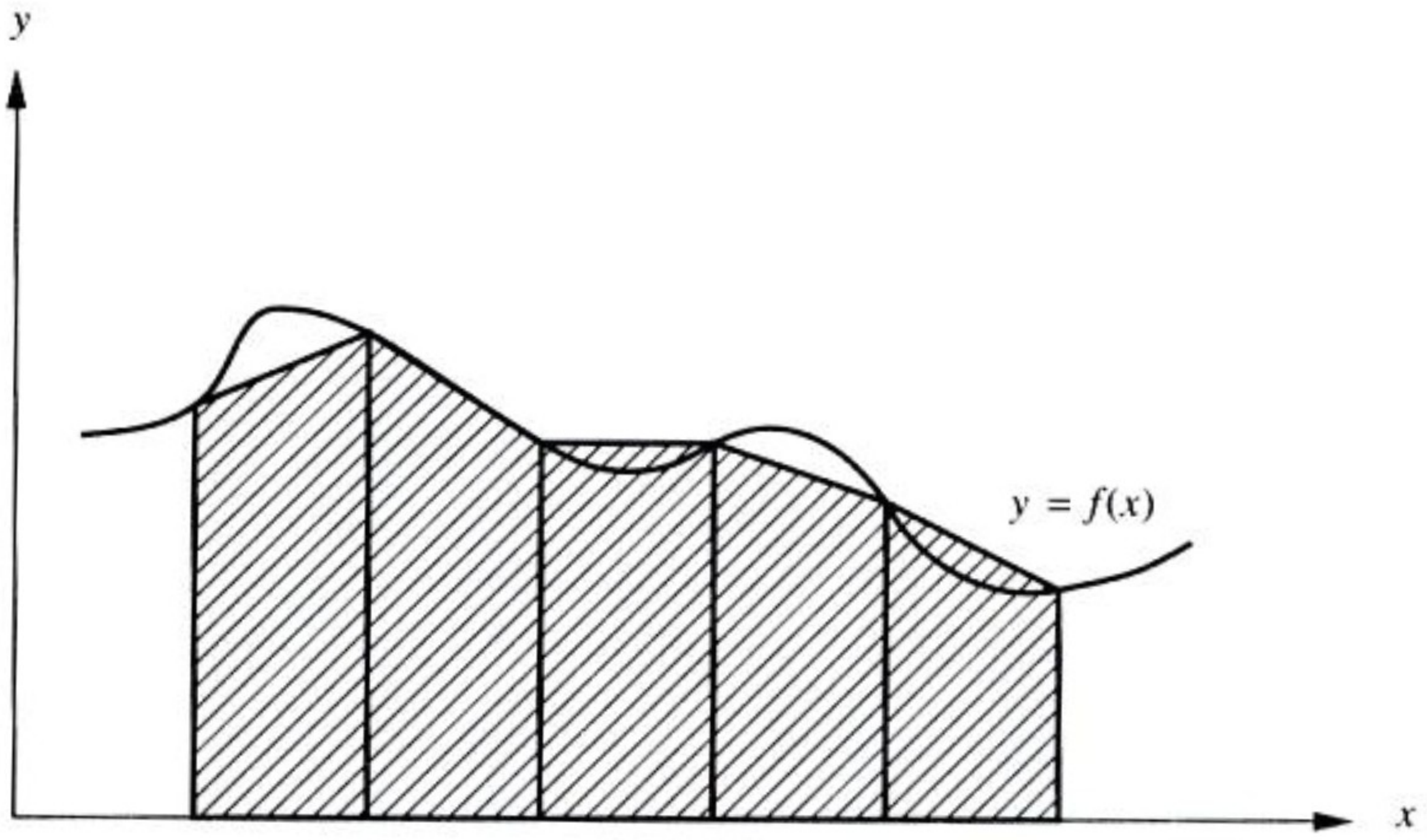


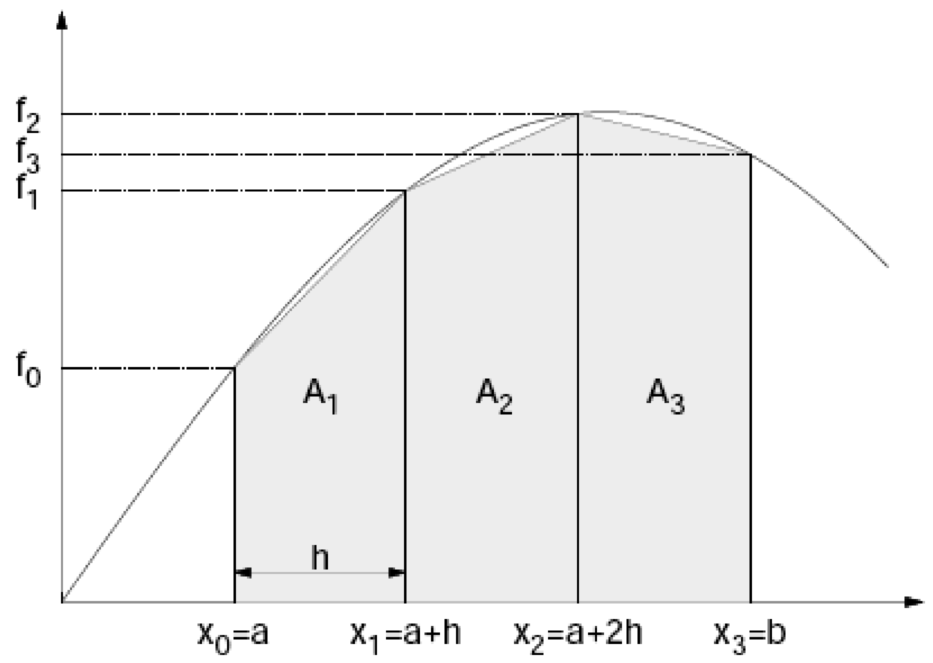
Figure 3.10 Graphical interpretation of the trapezoidal method.

```
/* implementaton of trapezoidal method */
double trapezoidal(
  double (*f)(double),    /* function to be integrated */
  double a,               /* starting x */
  double b,               /* ending x   */
  int n)                  /* number of panels */
{
  double answer, h;       /* result and panel width */
  int i;                  /* counter for intervals */

  answer = f(a)/2;
  h = (b-a)/n;

  /* sum panel areas */
  for(i=1; i<=n; i++)
    answer = answer + f(a+i*h);

  answer = answer - f(b)/2;
  return(h*answer);
}
```

The function prototype for `trapezoidal()` is

```
double trapezoidal(double (*)(double),
                   double, double, int);
```

Generalized trapezoidal method



$$\begin{aligned} A \approx A_1 + A_2 + A_3 &= \frac{1}{2}h(f_0+f_1) + \frac{1}{2}h(f_1+f_2) + \frac{1}{2}h(f_2+f_3) \\ &= h\left[\frac{1}{2}(f_0+f_3) + f_1 + f_2\right]. \end{aligned}$$

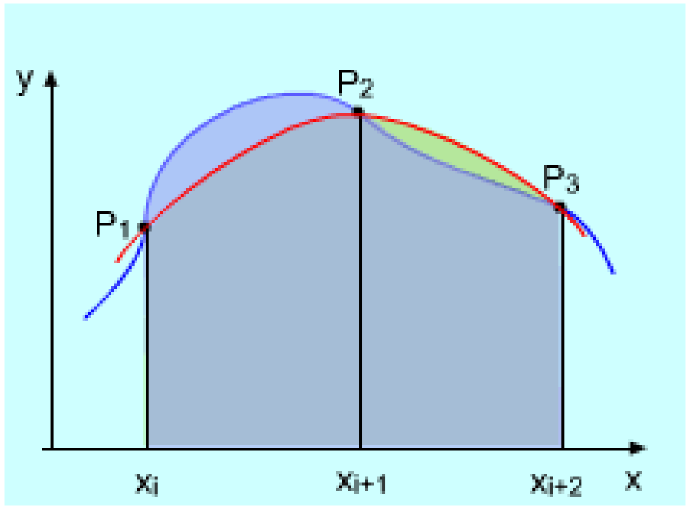
$$\begin{aligned} A = \int_a^b f(x)dx &= h\left[\frac{1}{2}(f_0+f_{N-1}) + f_1 + f_2 + \cdots + f_{N-2}\right] \\ &= h\left[\frac{1}{2}(f_o+f_{N-1}) + \sum_{i=1}^{N-2} f_i\right]. \end{aligned}$$

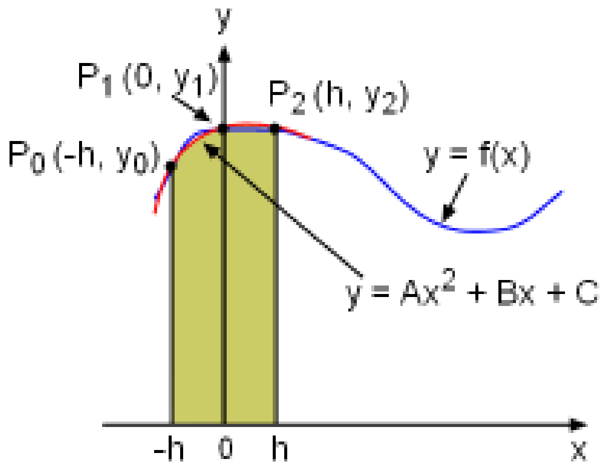
3.7.3 Simpson's Rule

A considerable improvement over the trapezoidal rule can be achieved by approximating the function within each of the n intervals by some polynomial. When second-order polynomials are used, the resulting algorithm is called *Simpson's rule*.

If we define h as the width of each of the subdivisions on the x axis and apply Simpson's rule, then the area under the curve in the interval $(x, x+h)$ is approximated by

$$\frac{h}{6}\left[f(x)+4f\left(x+\frac{h}{2}\right)+f(x+h)\right]$$





For details, see: http://en.wikipedia.org/wiki/Simpson's_rule

```
/* implementation of Simpson's rule */
double simpson(
  double (*f)(double),   /* function to be integrated */
  double a,              /* starting x */
  double b,              /* ending x   */
  int n)                 /* number of panels */
{
  double answer, h;      /* result and panel width */
  double x;
  int i;

  answer = f(a);
  h = (b-a)/n;

/* sum panel areas */
  for(i=1; i<=n; i++) {
    x = a+i*h;
    answer = answer + 4*f(x-h/2) + 2*f(x);
  }
  answer = answer - f(b);
  return(h*answer/6);
}
```

Adaptive setting of number of bins

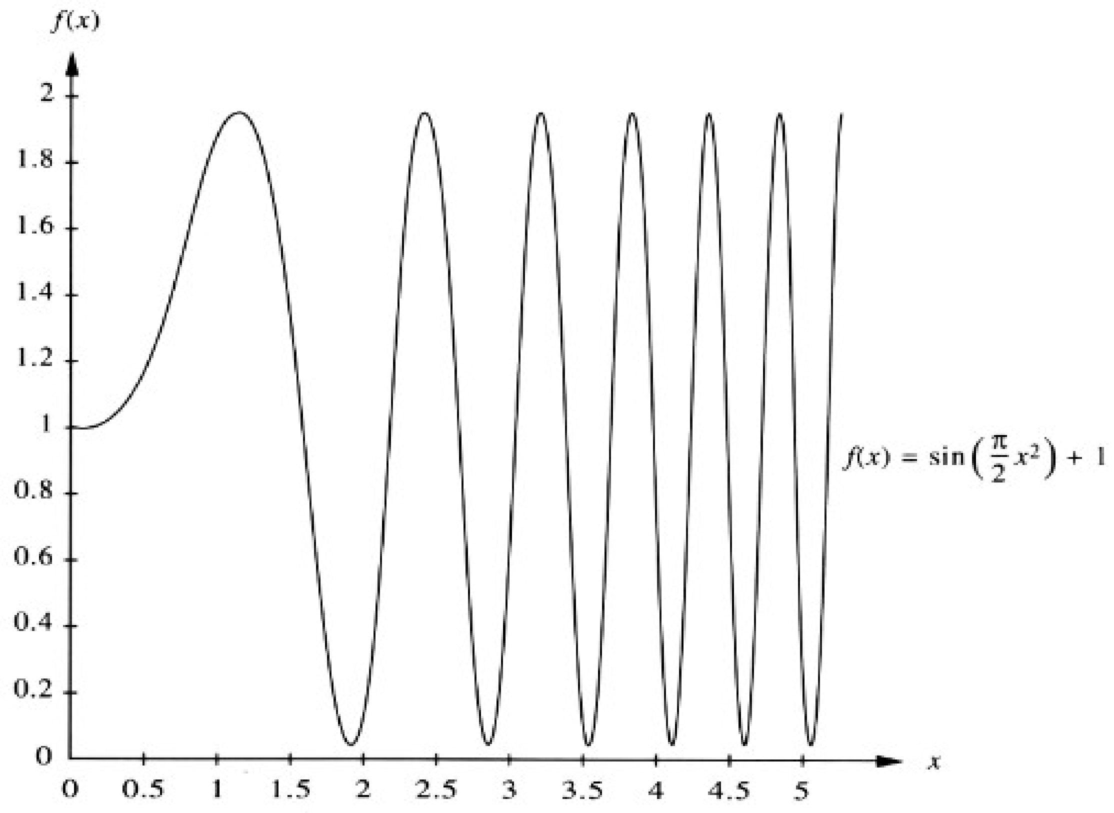
The number of bins, n, can be adaptively selected: starting with an initial value and doubling it until some desirable precision is achieved.

```c
#include <math.h>
double new_simpson(
  double (*f)(double),   /* function to be integrated */
  double a,              /* starting x */
  double b,              /* ending x   */
  int n0,                /* number of panels */
  double tolerance)      /* measure of convergence */

{
  double check= tolerance +1.0;
  double lowval, val;

  lowval = simpson(f, a, b, n0);
  while(check  > tolerance) {
    n0 = 2 * n0;
    val = simpson(f, a, b, n0);
    check = fabs((val-lowval)/val);
    lowval = val;
  }
  return(val);
}
```

Example:

$$f(x) = \int_{x=a}^{b} sin\left(\frac{\pi}{2}x^2\right)dx + 1$$

One way to view this function is as the area below the curve shown in Figure 3.11. This function cannot be integrated symbolically, so solution of its value for some particular interval $[a, b]$ requires numerical methods. In this section we explore how accurate various alternative approaches are in solving this integral for various intervals.



Example:

The resulting output of this program is given below.

```
  a    b     Trap.     Simpson's Adaptive  Reference
 ---  ---  ---------  --------- ---------  ---------
   0    1  1.43826    1.43826   1.43826    1.43826
   1    2  0.910423   0.905149  0.905156   0.905157
   2    3  1.14764    1.1529    1.1529     1.1529
   3    4  0.934949   0.924135  0.9242     0.924204
   4    5  1.06794    1.07874   1.07868    1.07867
   5    6  0.96447    0.947518  0.947771   0.947771
   6    7  1.03606    1.05299   1.05274    1.05274
   7    8  0.98401    0.959849  0.96051    0.960512
   8    9  1.01618    1.0403    1.03965    1.03964
   9   10  1.00006    0.966832  0.968312   0.968313
  10   11  1.00007    1.03321   1.03175    1.03175
  11   12  1.01618    0.970491  0.973554   0.973555
  12   13  0.984014   1.02949   1.02647    1.02647
  13   14  1.03606    0.971057  0.977315   0.977315
  14   15  0.964484   1.02879   1.0227     1.0227
  15   16  1.06793    0.966563  0.980141   0.980142
  16   17  0.934986   1.03248   1.01987    1.01987
  17   18   1.1476     0.945541  0.982342   0.982344
  18   19  0.91056    1.03509   1.01767    1.01766
  19   20  1.43787    0.854051  0.984108   0.984107
```

Ordinary differential equations

A vast range of problems in science and engineering are characterized by *differential equations*, where the rate of change of some quantity is a function of the values of other variables, potentially including other rates of change. It is quite common for these differential equations to be mathematically unsolvable, requiring that numerical methods be applied.

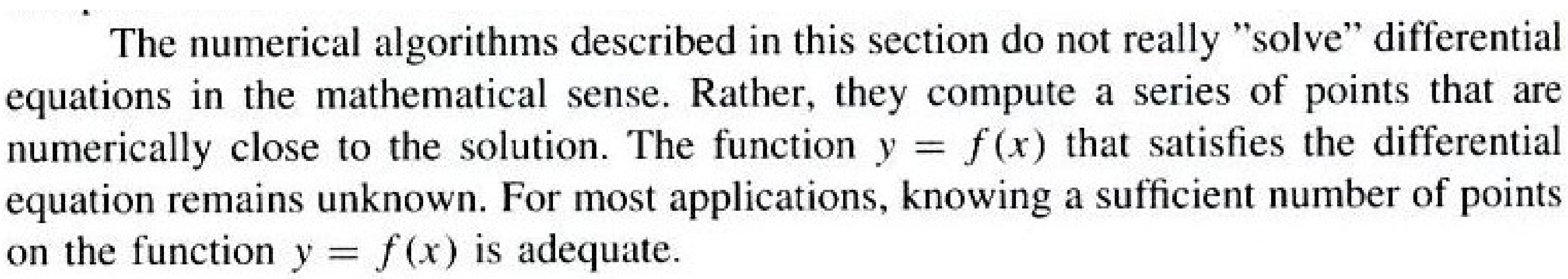
We restrict our presentation here to the solution of first-order ordinary differential equations. These are of the form

$$\frac{dy}{dx} = g(x, y)$$

or, defining $y' = dy/dx$,

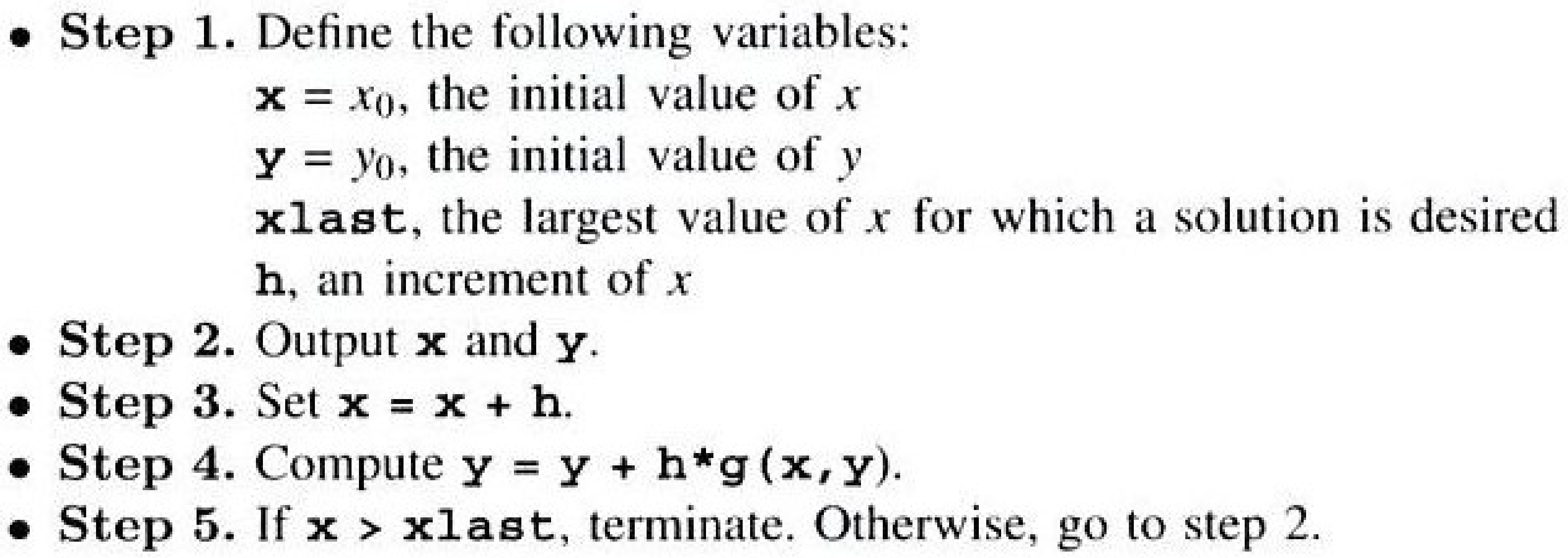
$$y' = g(x, y)$$

Restricting attention to only first-order differential equations is not as limiting as it first may seem. It is quite simple to convert an mth-order differential equation into a system of m first-order equations.



The numerical algorithms described in this section do not really "solve" differential equations in the mathematical sense. Rather, they compute a series of points that are numerically close to the solution. The function $y = f(x)$ that satisfies the differential equation remains unknown. For most applications, knowing a sufficient number of points on the function $y = f(x)$ is adequate.

Euler's algorithm



- **Step 1.** Define the following variables:
  `x` = x_0, the initial value of x
  `y` = y_0, the initial value of y
  `xlast`, the largest value of x for which a solution is desired
  `h`, an increment of x
- **Step 2.** Output **`x`** and **`y`**.
- **Step 3.** Set **`x = x + h`**.
- **Step 4.** Compute **`y = y + h*g(x,y)`**.
- **Step 5.** If **`x > xlast`**, terminate. Otherwise, go to step 2.

```
#include <stdio.h>

void euler(
  double x0,      /* Initial x */
  double y0,      /* Initial y */
  double (*g)(double,double),
            /*pointer to derivative function */
  double h,       /* increment */
  double xlast)   /* largest x */

{

  printf("       x              f(x)\n");
  for ( ; x0 <= xlast; x0 = x0+h) {
    printf("%15.6g  %15.6g\n", x0, y0 );
    y0 = y0 + h * g(x0, y0);
  }
}
```

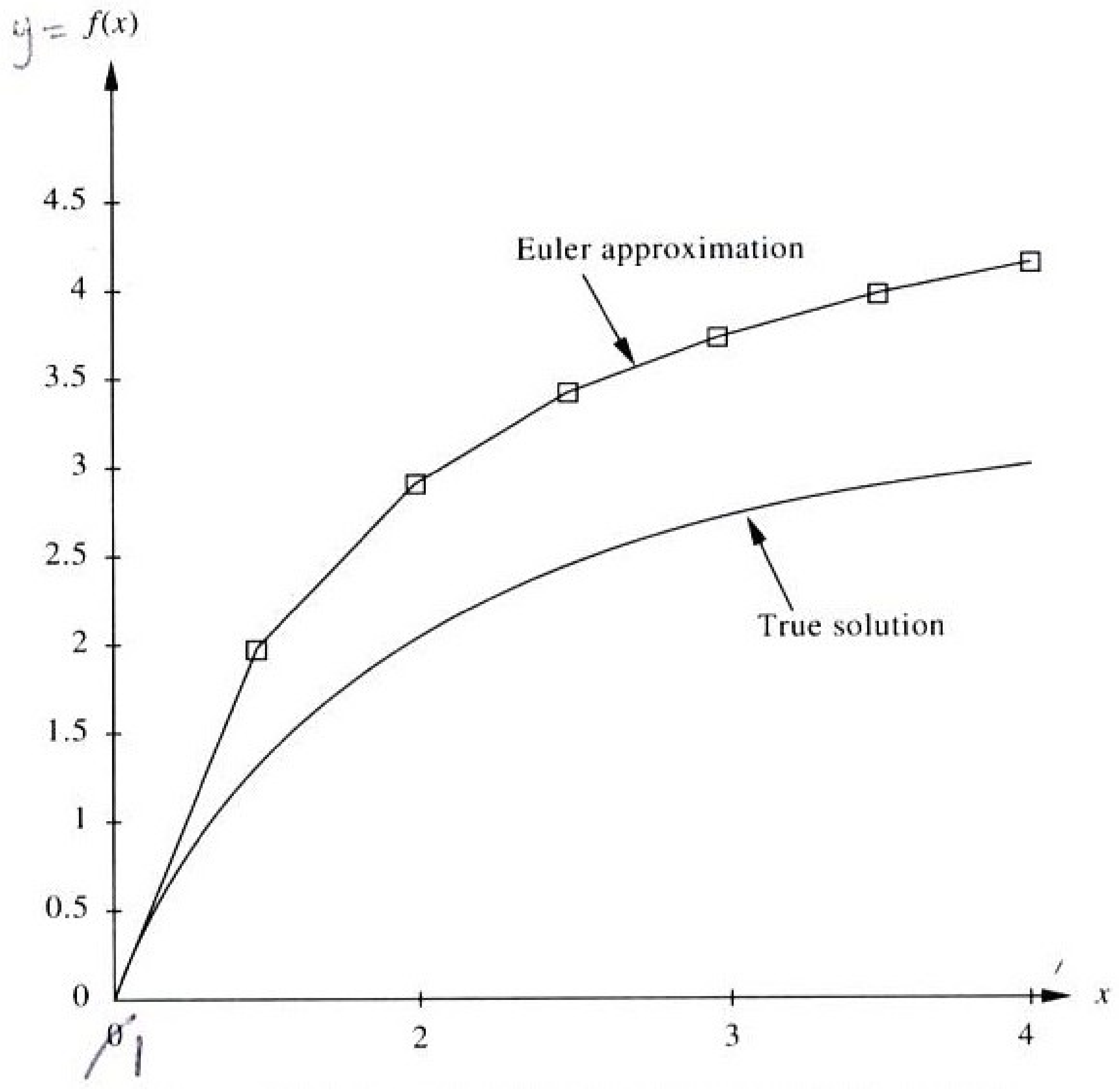
For example, if the differential equation to be solved is

$$y' = -\frac{y-4}{x}, \qquad y(1) = 0$$

then the C function for the differential equation would be

```
double func(double x, double y)
{
  return( -(y-4)/ x );
}
```

Errors in Euler's algorithm



A local error is inherent in each estimation and then the derivative is computed at a point different than the true solution.

Result: Euler's method will quickly diverge from the true solution even with small step sizes

Runge-Kutta methods

Runge-Kutta methods are based on the observation that each step of Euler's method relies entirely on the derivative evaluated at a single point. If one could compute a better estimate of the derivative, it would be possible to improve on each Euler method step. Runge-Kutta methods do this by evaluating the derivative at more than one point for each step and combining the various computed derivatives to obtain a better approximation of the solution. The goal is to reduce the local error at each step to something considerably smaller than $O(h^2)$.

Second order:

- **Step 1.** Define the following variables:
  - `x` = x_0, the initial value of x
  - `y` = y_0, the initial value of y
  - `xlast`, the largest value of x for which a solution is desired
  - `h`, an increment of x
  - `temp`, a temporary variable
- **Step 2.** Output `x` and `y`.
- **Step 3.** Compute `temp = h*g(x,y)`.
- **Step 4.** Compute `y = y + h*g(x + h/2, y+temp/2)`.
- **Step 5.** Set `x = x + h`.
- **Step 6.** If `x > xlast`, terminate. Otherwise, go to step 2.

In this algorithm, the derivative at the midpoint between x and $x + h$ is used to compute the direction for the next step. This reduces the local error per step from $O(h^2)$ in Euler's method to $O(h^3)$. This is called the *second-order* Runge-Kutta method. (Euler's method can be viewed as the first-order Runge-Kutta method.)

Fourth order Runge-Kutta:

- **Step 1.** Define the following variables:
  - `x` = x_0, the initial value of x
  - `y` = y_0, the initial value of y
  - `xlast`, the largest value of x for which a solution is desired
  - `h`, an increment of x
  - `ta, tb, tc,` and `td`, temporary variables
- **Step 2.** Output `x` and `y`.
- **Step 3.** Compute:

```
ta =   h*g(x, y)
tb =   h*g(x+h/2, y+ta/2)
tc =   h*g(x+h/2,y+tb/2)
td =   h*g(x+h, y+tc)
```

- **Step 4.** Compute `y = y + ta/6 + tb/3 + tc/3 + td/6`.
- **Step 5.** Set `x = x + h`.
- **Step 6.** If `x > xlast`, terminate. Otherwise, go to step 2.

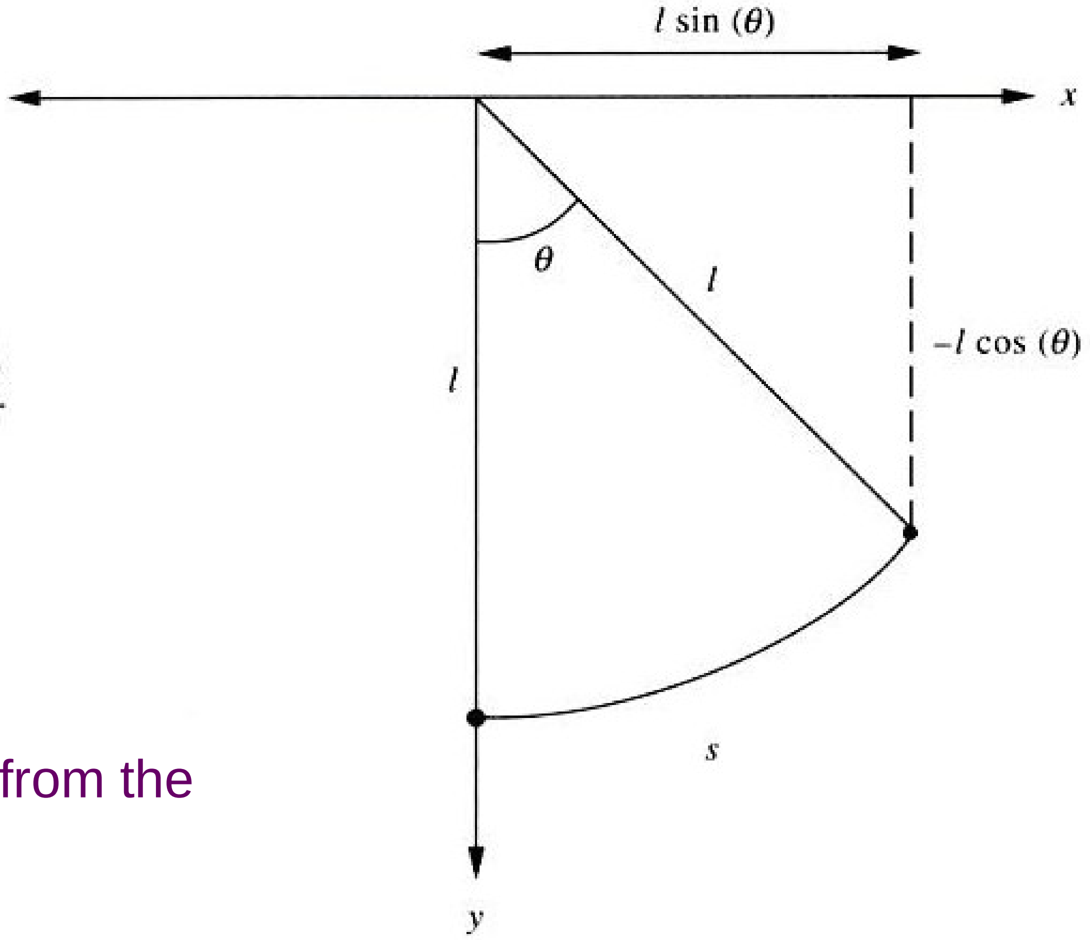
```
/* fourth-order Runge-Kutta method */
void runge_kutta(
  double x0,      /* Initial x */
  double y0,      /* Initial y */
  /*pointer to derivative function */
  double (*g)(double,double),
  double h,       /* increment */
  double xlast)   /* largest x */
{

  double ta, tb, tc, td;
  printf("       x           f(x)\n");
  for ( ; x0 <= xlast; x0 = x0+h) {
    printf("%15.6g %15.6g\n", x0, y0 );
    ta =  h * g(x0, y0);
    tb =  h * g(x0+h/2.0, y0+ta/2.0);
    tc =  h * g(x0+h/2.0,y0+tb/2.0);
    td =  h * g(x0+h, y0+tc);
    y0 = y0 + (ta +2.0* tb +2.0*tc + td)/6.0;

  }
}
```

Application: Motion of a pendulum

$$x(t) = l \sin \theta(t) = l \sin \frac{s(t)}{l}$$

$$y(t) = -l \cos \theta(t) = -l \cos \frac{s(t)}{l}$$

$$\frac{d^2 s}{dt^2} = -g \sin \frac{s(t)}{l}$$



First order equations obtained from the second order equation:

$$\frac{dz(t)}{dt} = -g \sin \frac{s(t)}{l}$$

$$\frac{ds(t)}{dt} = z(t)$$

```
#include <stdio.h>
#include <math.h>
#define GRAVITY -9.8  /* gravitational constant */

main()
{
  double h;       /* time increments for computation */
  double s0;      /* initial  pendulum position */
  double tlast;   /* time limit for computation */
  double length;  /* length of pendulum */
  double s,z,t;          /* values of s,z and t */
  double sa, sb, sc, sd; /* intermediate values of s */
  double za, zb, zc, zd; /* intermediate values of z */

/* prompt for and read input values */
  printf("Enter initial value of s:");
  scanf("%lf", &s0);
  printf("Enter last value of t:");
  scanf("%lf", &tlast);
  printf("Enter value of time increments:");
  scanf("%lf", &h);
  printf("Enter length of pendulum:");
  scanf("%lf", &length);

/* print labels */
  printf("    t     s(t)     x(t)     y(t)\n");

/* initialize values */
  s = s0;
  z = 0;
/* perform runge-kutta iterations */
  for (t=0 ; t <= tlast; t = t+h) {

  /* output the values of key variables */
    printf("%12.6g  %12.6g %12.6g %12.6g\n", t, s,
      length*sin(s/length), -length*cos(s/length));
  /* update z and s */
      za = h * GRAVITY * sin(s/length);
      sa = h * z;
      zb = h * GRAVITY * sin((s+sa/2)/length);
      sb = h * (z+za/2);
      zc = h *GRAVITY * sin((s+sb/2)/length);
      sc = h * (z + zb/2);
      zd = h * GRAVITY * sin((s+sc)/length);
      sd = h * (z + zc);
      z = z + za/6 + zb/3 + zc/3 + zd/6;
      s = s + sa/6 + sb/3 + sc/3 + sd/6;
    }
 }
```