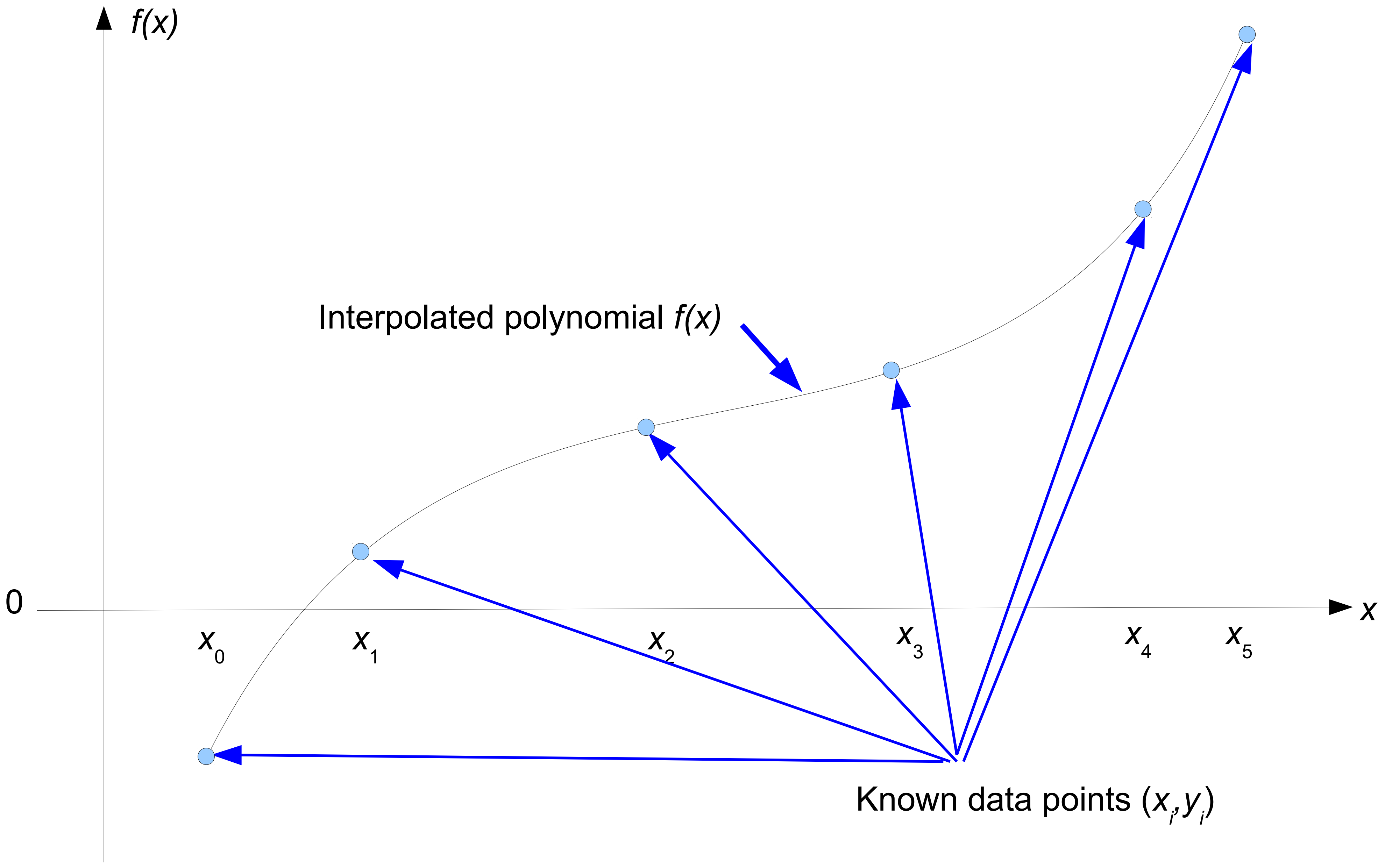
Class Progress

Basics of Linux, gnuplot, C
Visualization of numerical data
Roots of nonlinear equations
  (Midterm 1)
Solutions of systems of linear equations
Solutions of systems of nonlinear equations
Monte Carlo simulation
Interpolation of sparse data points
Numerical integration
  (Midterm 2)
Solutions of ordinary differential equations

Fitting Polynomial to Known Data Points

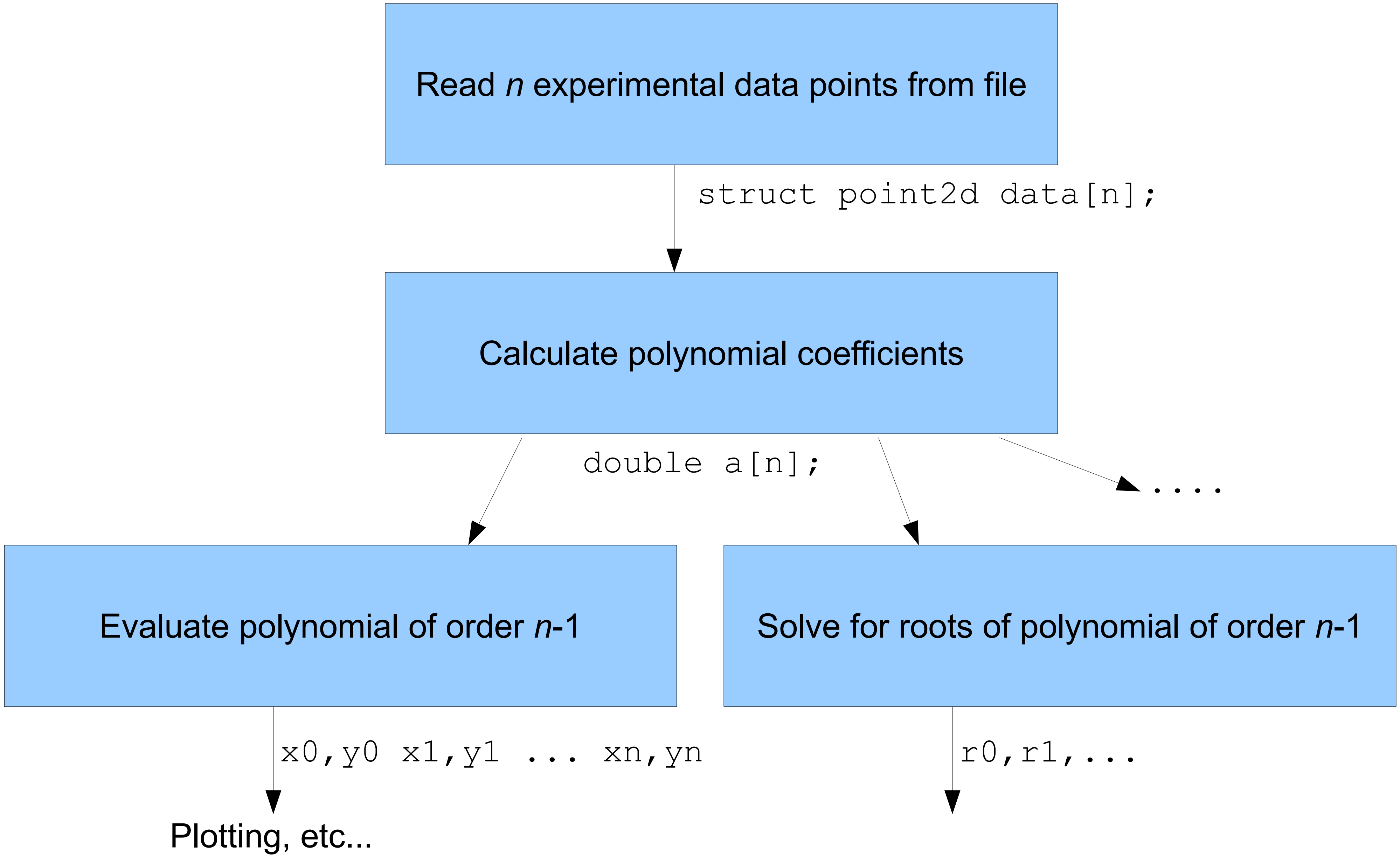

$f(x)$
Interpolated polynomial $f(x)$
0
x
x_0
x_1
x_2
x_3
x_4
x_5
Known data points (x_i, y_i)

When is Interpolation Used?

When a relatively small number of data points are “expensive” to obtain, either from experiment or computation, costing large amounts of time or effort, and

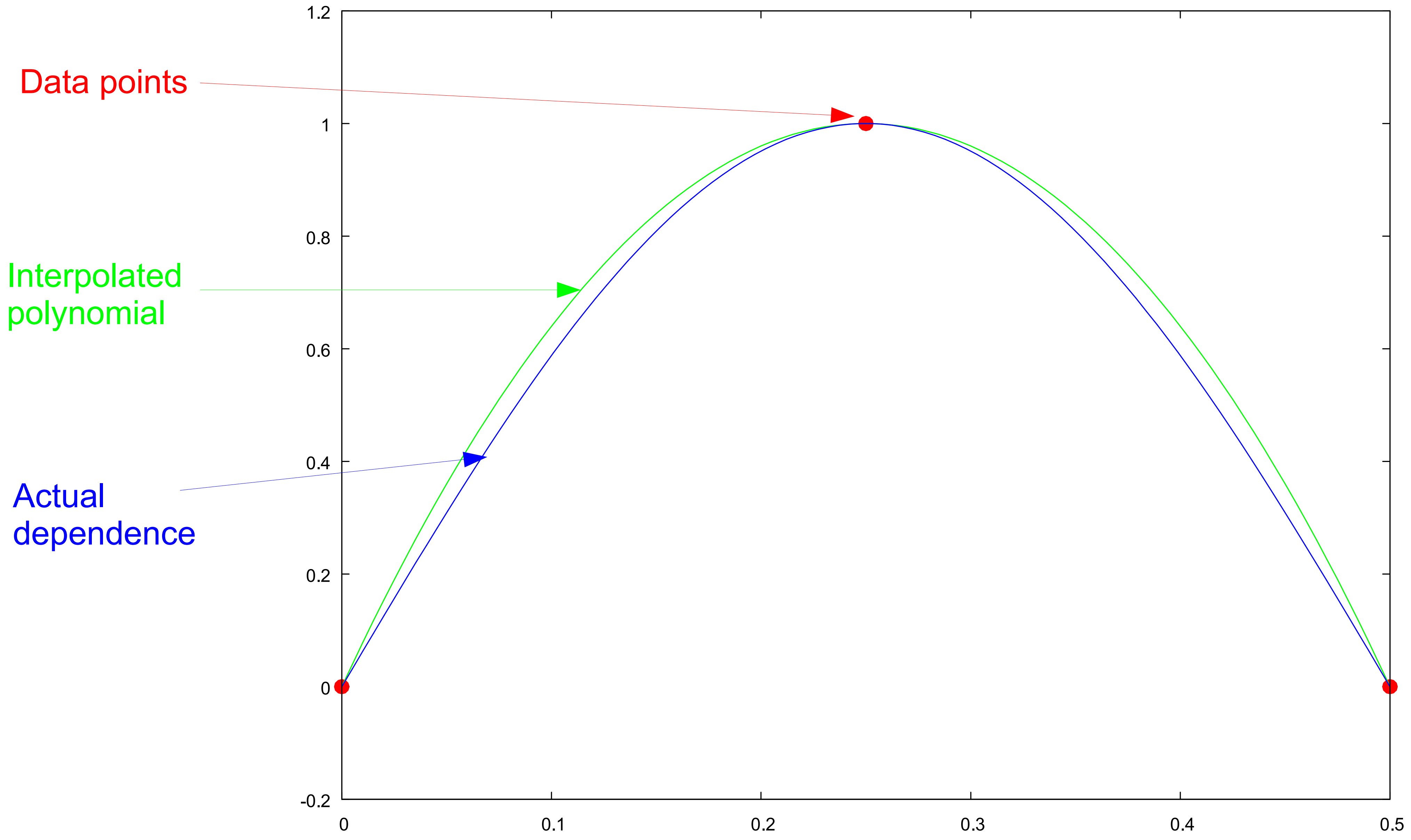
A finer density of intermediate points are needed for plotting, solving, or extracting a derivative

Interpolation Flow

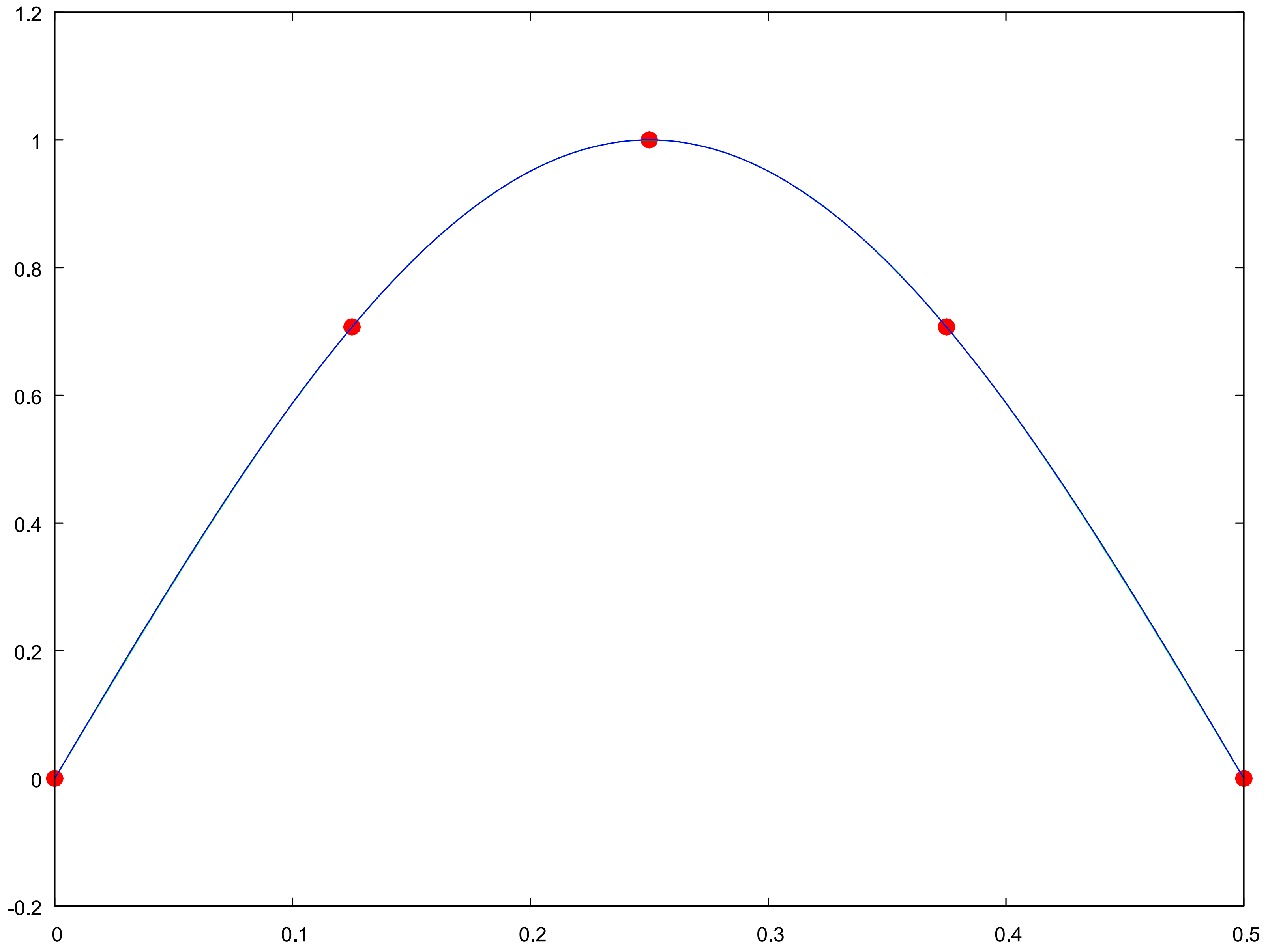

Read *n* experimental data points from file
Calculate polynomial coefficients
Evaluate polynomial of order *n*-1
Solve for roots of polynomial of order *n*-1
Plotting, etc...

```
struct point2d data[n];
double a[n];
....
x0,y0 x1,y1 ... xn,yn
r0,r1,...
```

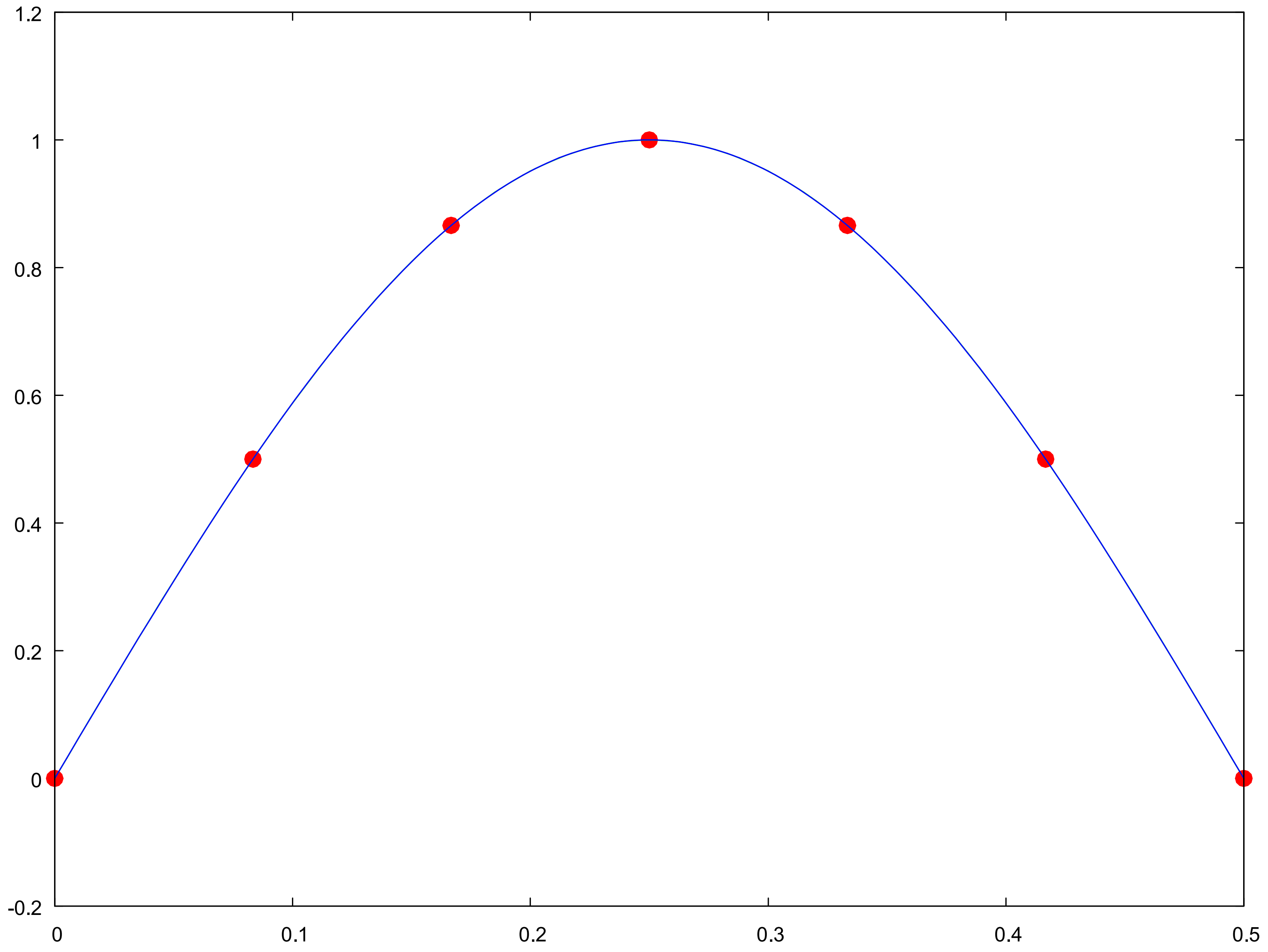
Example Polynomial Interpolation, *n*=3



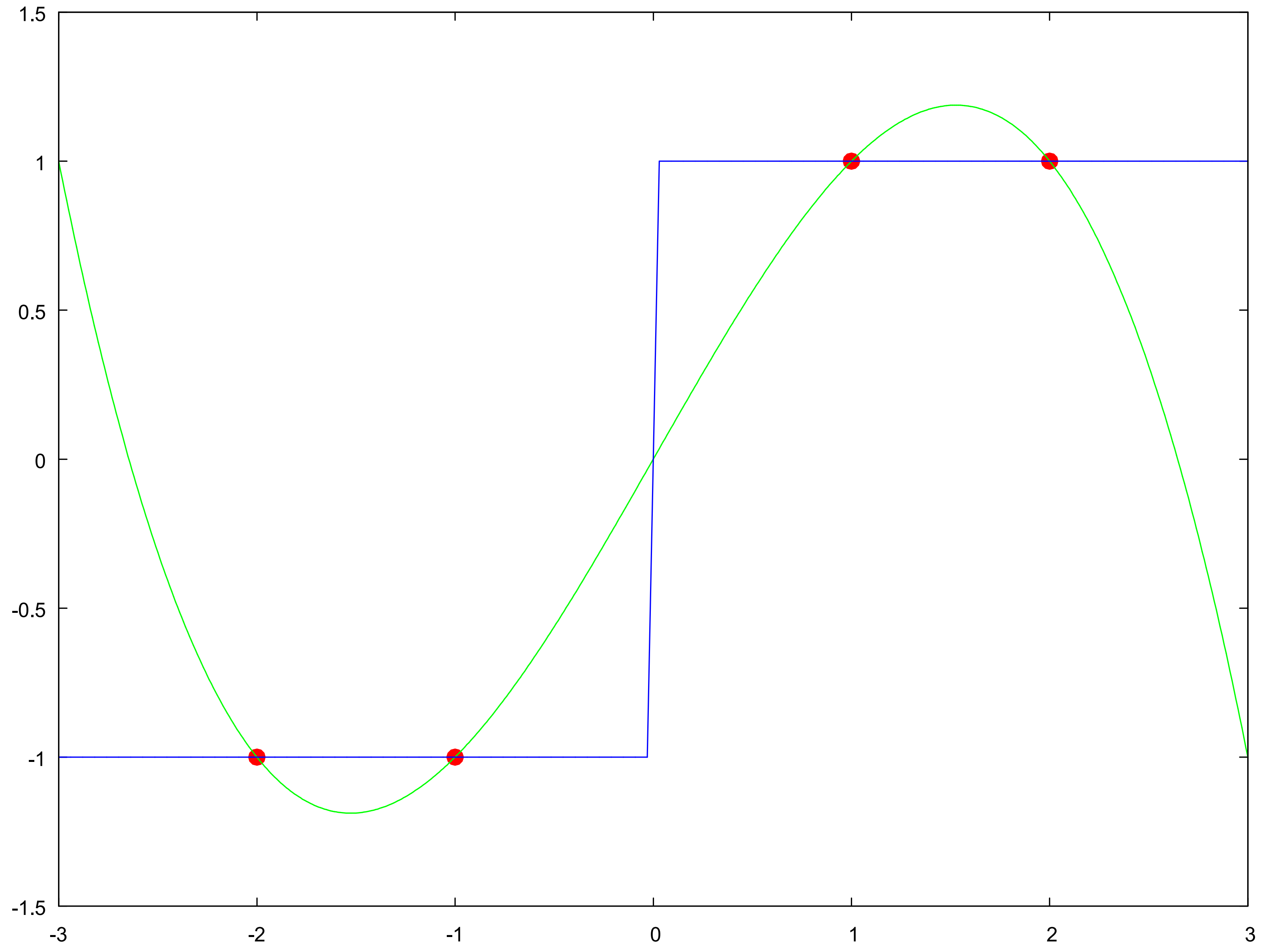
Example Polynomial Interpolation, *n*=5



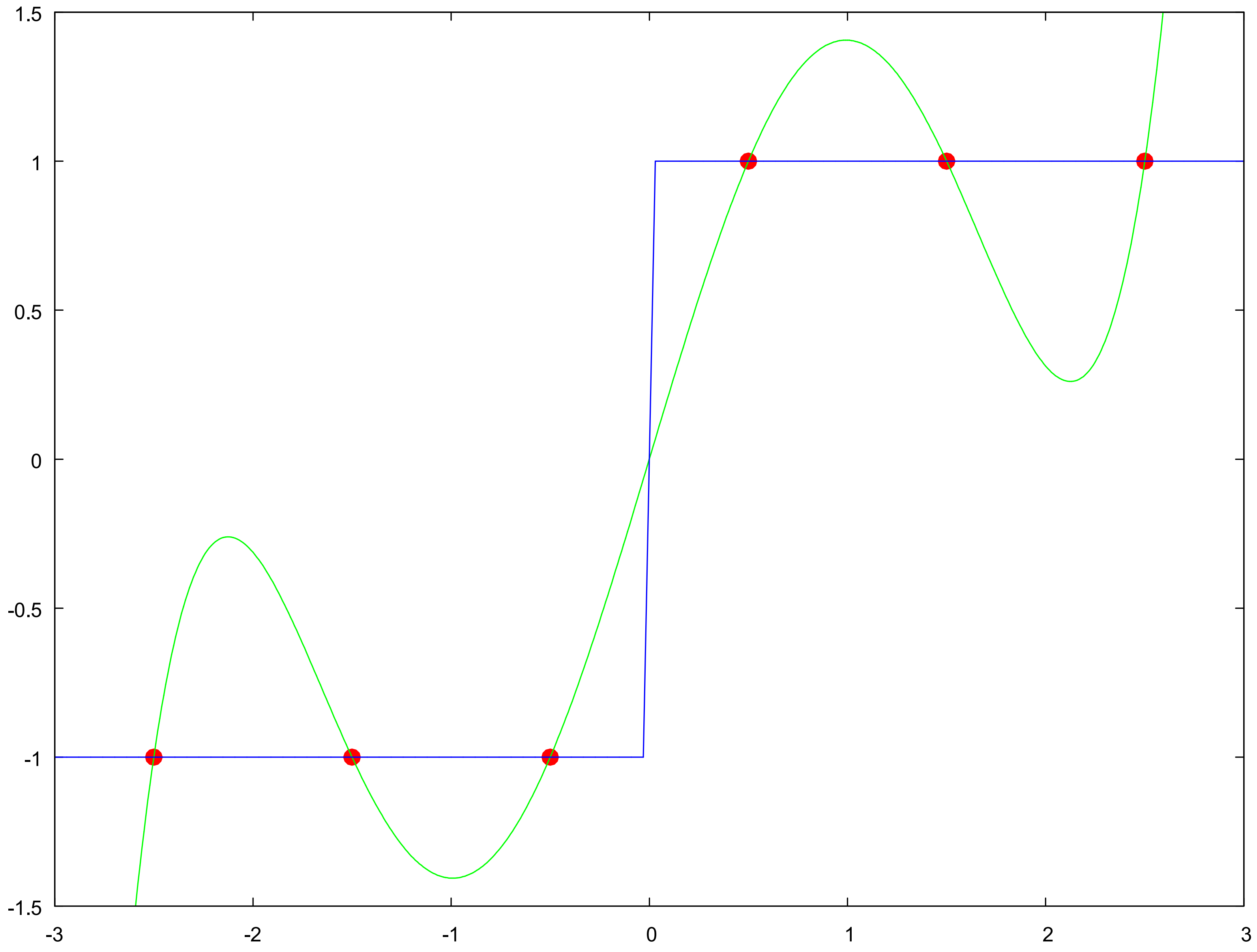
Example Polynomial Interpolation, *n*=7



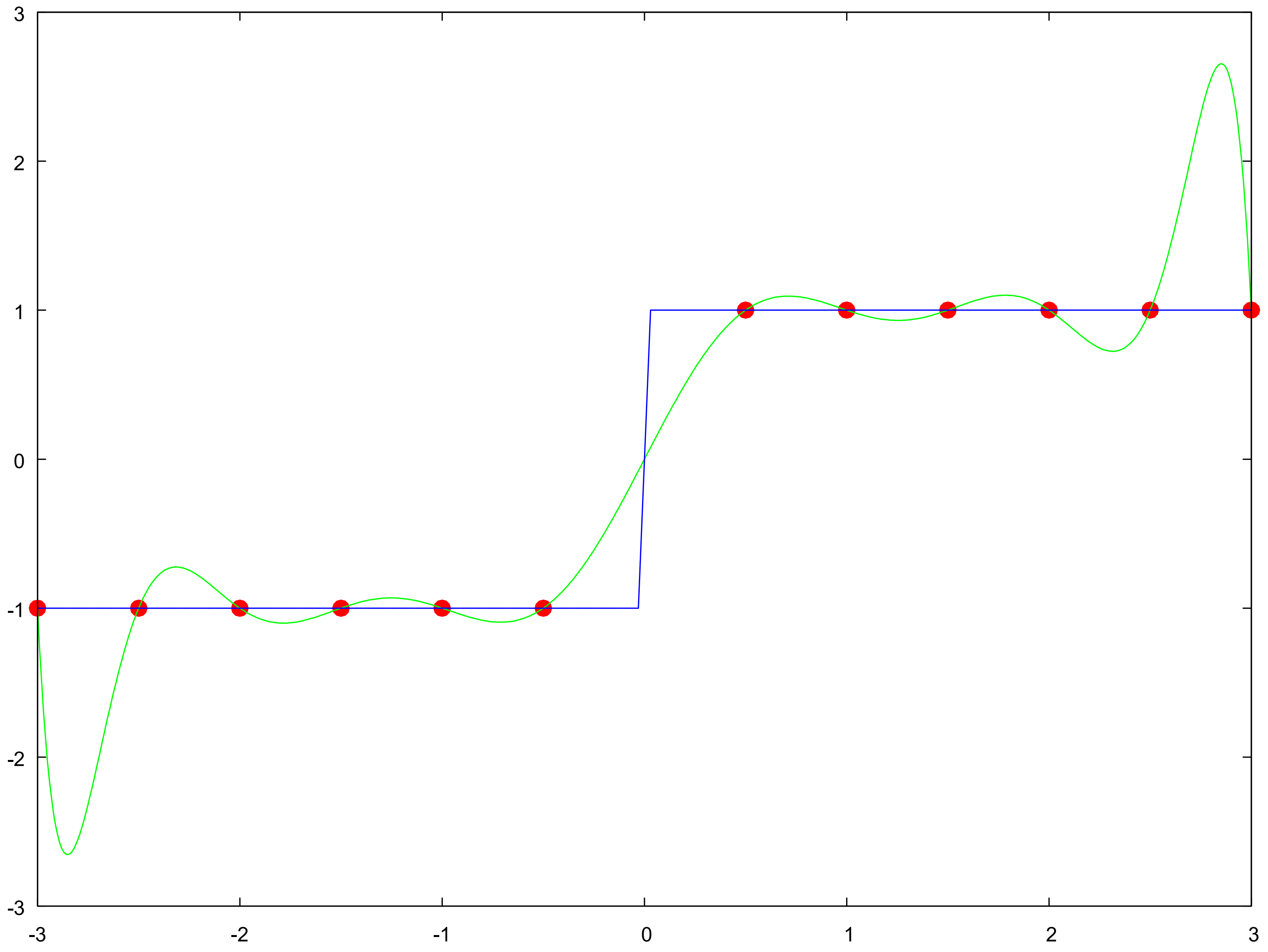
Example Polynomial Interpolation, *n*=4



Example Polynomial Interpolation, n=6



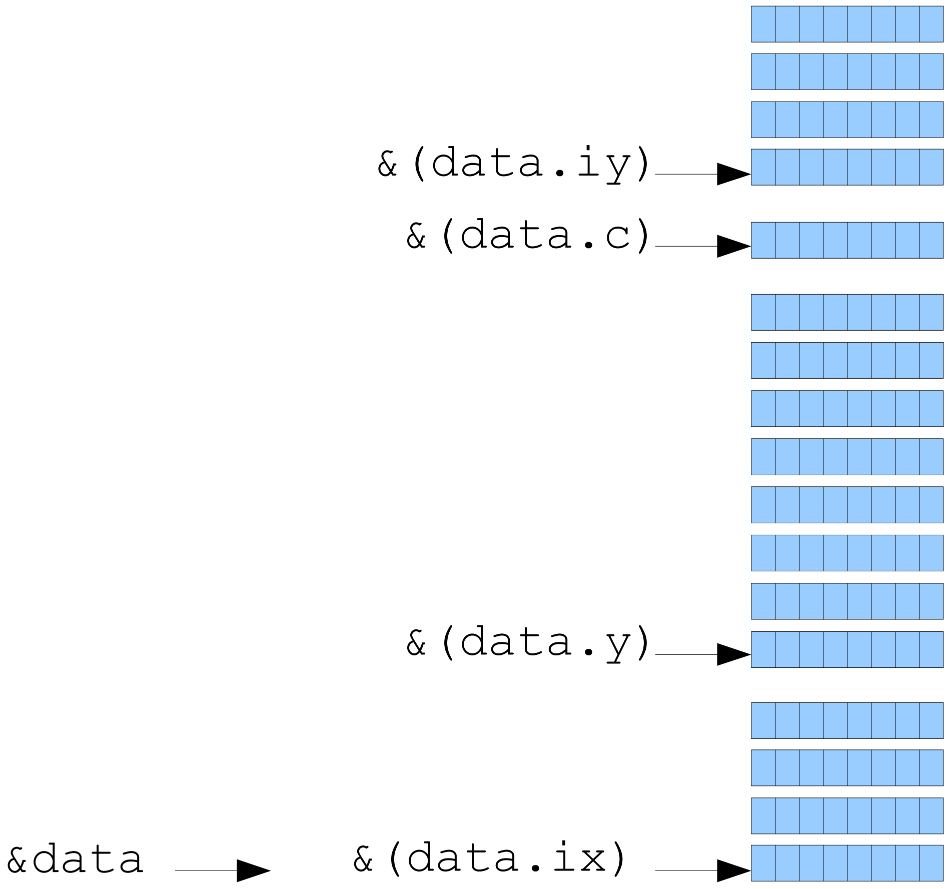
Example Polynomial Interpolation, *n*=12



Allocation of C structures in RAM

```
struct {
  int ix;
  double y;
  char c;
  int iy;
} data;
```

Individual structure elements are referenced as `data.ix`, `data.y`, `data.c`, `data.iy`



Use Structures to Define New Data Types

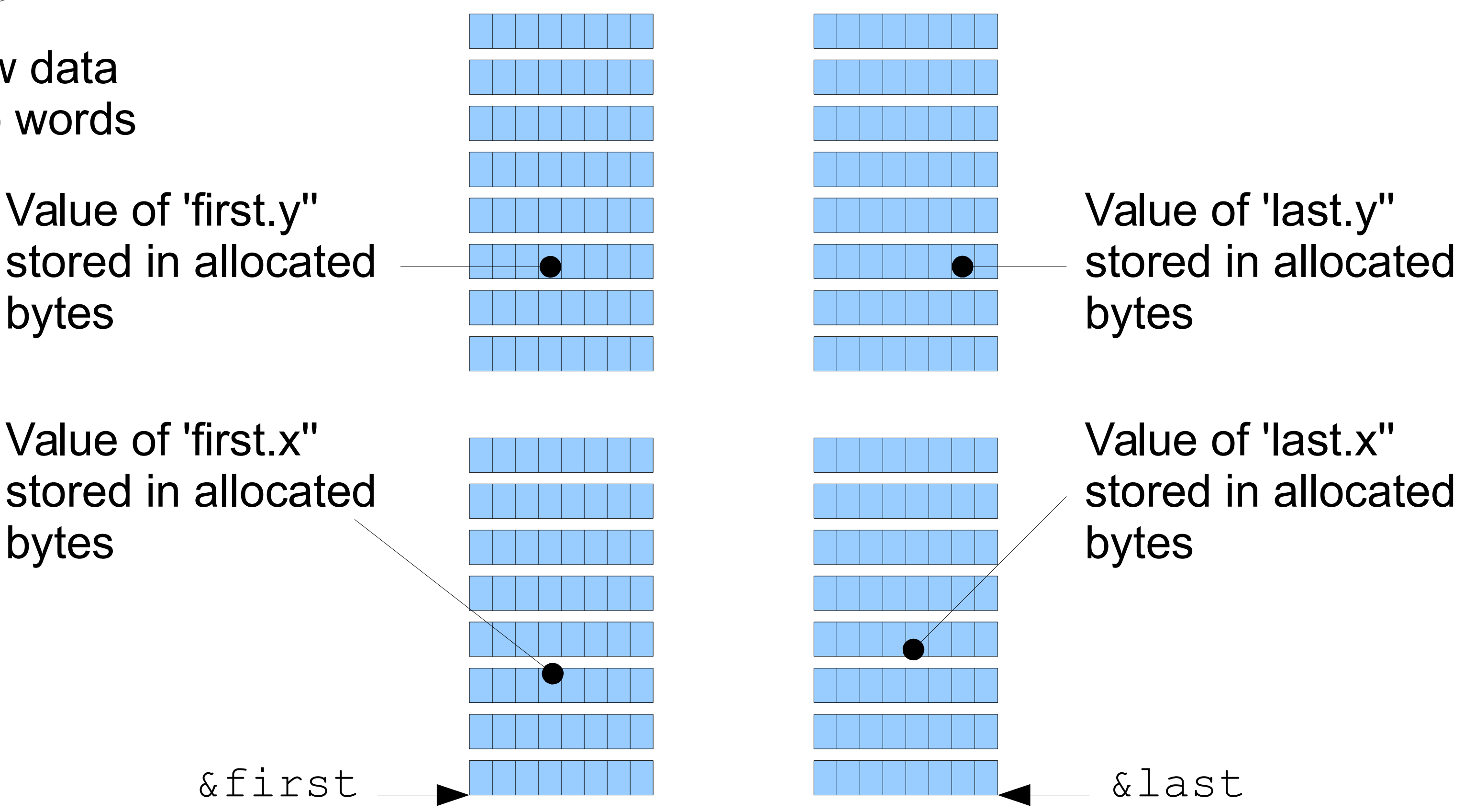
Define structure type by including a structure name, but do not allocate memory here

```
struct point2d {
   double x,y;
};
```

Allocate memory here for two instances of the structure

```
struct point2d first,last;
```

$\underbrace{\texttt{struct point2d}}$ Syntactically like a new data type, described by two words


Value of 'first.y" stored in allocated bytes
Value of 'last.y" stored in allocated bytes
Value of 'first.x" stored in allocated bytes
Value of 'last.x" stored in allocated bytes
&first
&last

Always Use 'sizeof()' to Find Structure Sizes

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

struct test1 {
  double x,y,z;
};

struct test2 {
  int ix;
  char c;
  double y;
  char d;
};

int main() {
  struct test1 data1;
  struct test2 data2;

  printf("Size of test1: %u\n",sizeof(struct test1));
  printf("Size of test2: %u\n",sizeof(struct test2));
  exit(0);
}
```

3 doubles should take 3x8=24 bytes

An int, 2 chars and a double should take 4+1+8+1=14 bytes?

Output on a 32-bit machine:

```
Size of test1: 24
Size of test2: 20
```

OK, makes sense

???

Always Use 'sizeof()' to Find Structure Sizes

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

struct test1 {
  double x,y,z;
};

struct test2 {
  int ix;
  char c,d;
  double y;
};

int main() {
  struct test1 data1;
  struct test2 data2;

  printf("Size of test1: %u\n",sizeof(struct test1));
  printf("Size of test2: %u\n",sizeof(struct test2));
  exit(0);
}
```

3 doubles should take 3x8=24 bytes

Reordered slightly, still an int, 2 chars and a double should take 4+1+8+1=14 bytes?

Output on a 32-bit machine:

```
Size of test1: 24
Size of test2: 16
```

OK, makes sense

???

Structures Can Have Pointers To Them

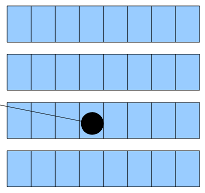
Define structure type by including a structure name, but do not allocate memory here

```
struct point2d {
  double x,y;
};
```

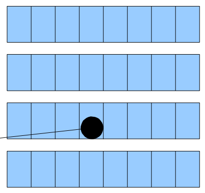
Allocate memory here for two instances of the structure and two pointers to the structure

```
struct point2d first,last,*pf,*pl;

pf = &first;
pl = &last;
```

Value of 'pl', the address of 'last', stored in allocated bytes



Value of 'pf', the address of 'first'' stored in allocated bytes



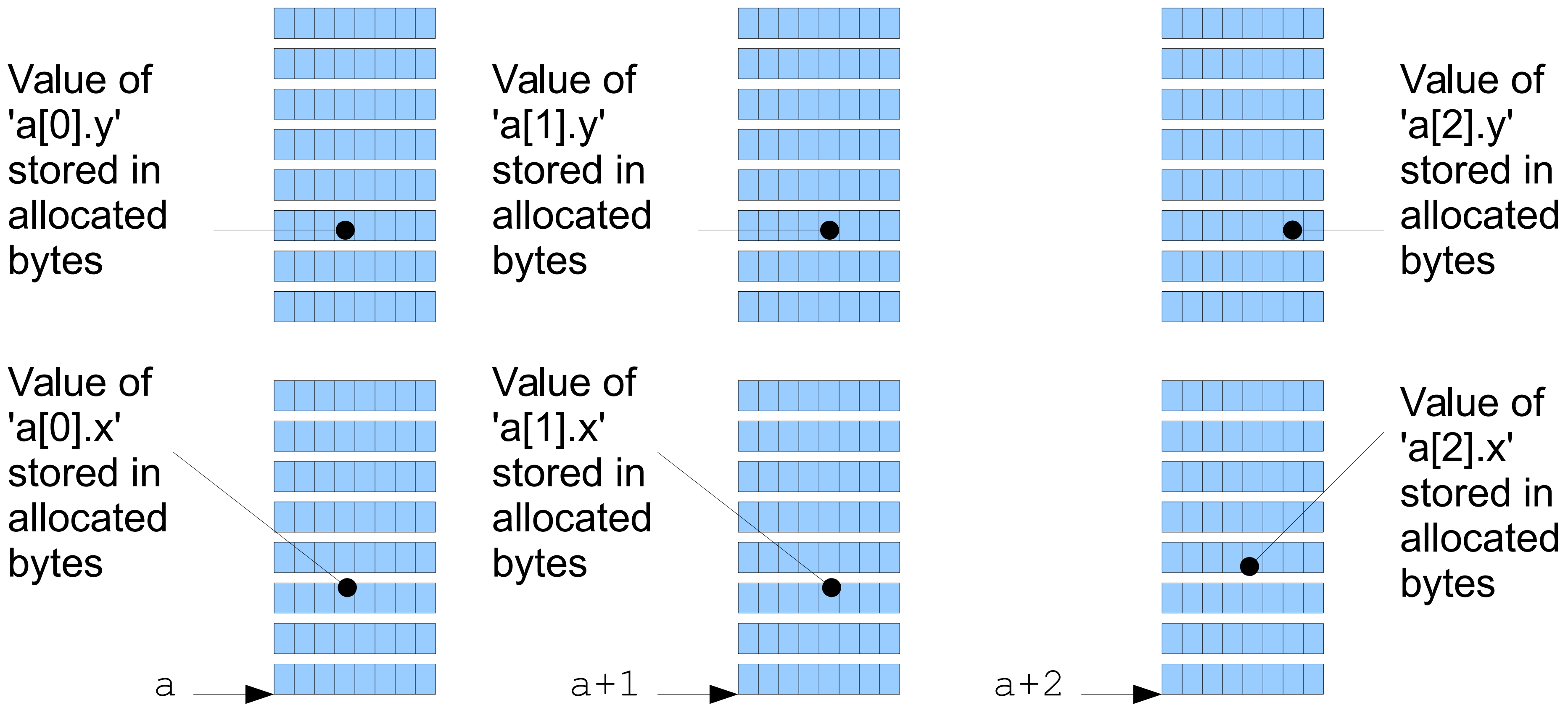
Individual structure elements can referenced as `(*pf).x, (*pf).y, (*pl).x, (*pf).y` or as `pf->x, pf->y, pl->x, pl->y`

Arrays of Structures

Define structure type by including a structure name, but do not allocate memory here

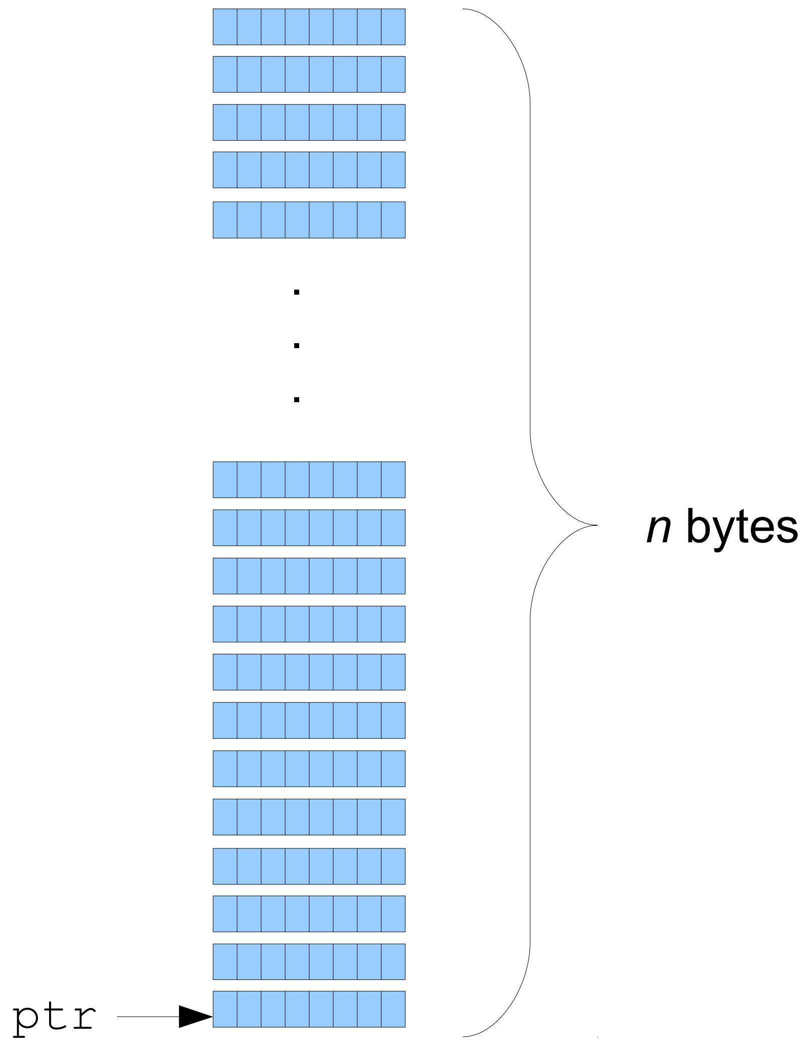
```
struct point2d {
   double x,y;
};

struct point2d a[3];
```

Allocate memory here for a three-element array of the structure

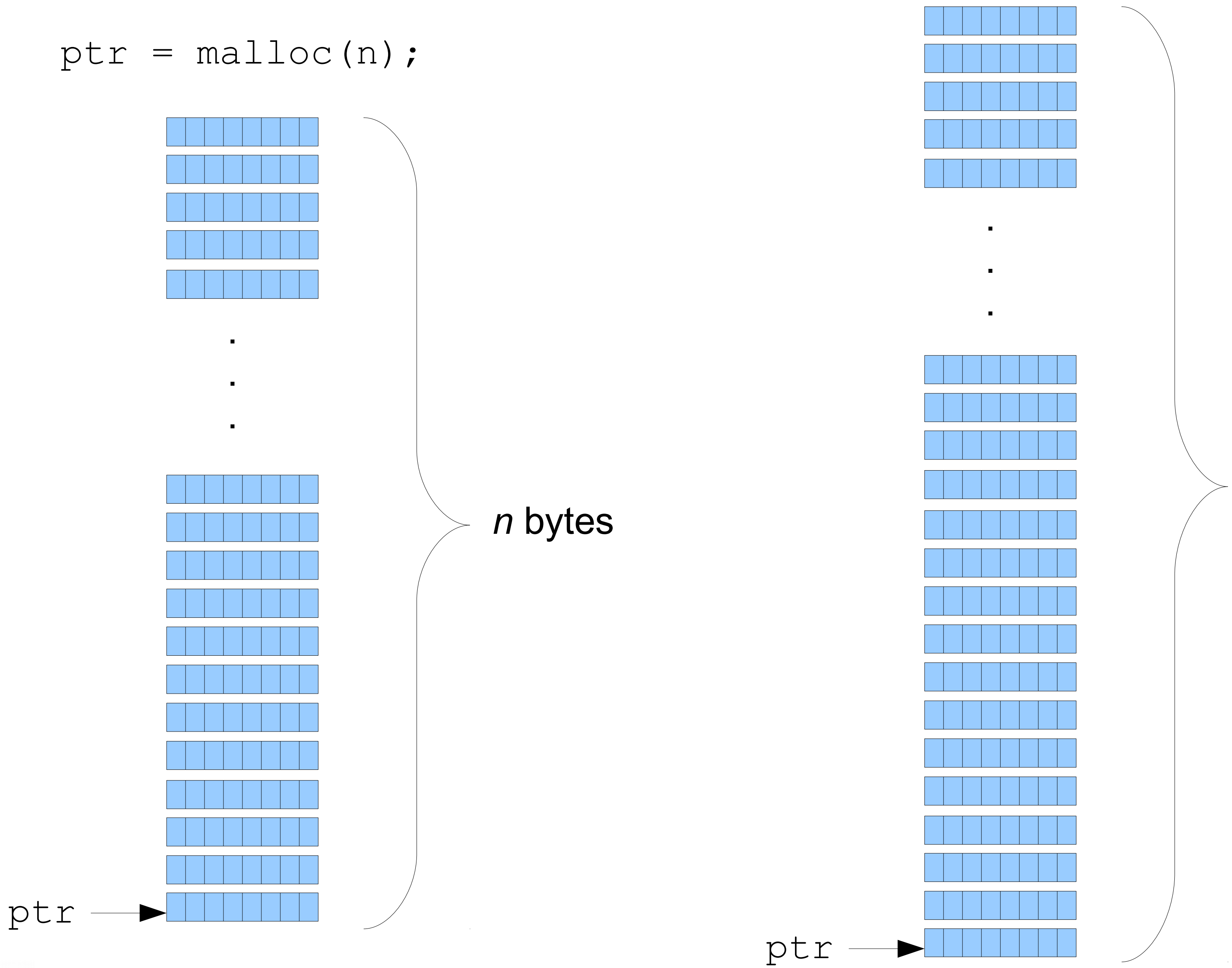

Value of 'a[0].y' stored in allocated bytes
Value of 'a[1].y' stored in allocated bytes
Value of 'a[2].y' stored in allocated bytes
Value of 'a[0].x' stored in allocated bytes
Value of 'a[1].x' stored in allocated bytes
Value of 'a[2].x' stored in allocated bytes
a
a+1
a+2

Allocation of Programmable Lengths of RAM

```
ptr = malloc(n);
```



Reallocation of Expanded Lengths of RAM



Reading a Data File of Arbitrary Length

Header file data_file.h:

Declare structure to hold coordinates of each data point

```
struct point2d {
  double x;
  double y;
};
int read_data_file(char *file_name,struct point2d **data_ptr);
```

Reading a Data File of Arbitrary Length

Source code file data_file.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "data_file.h"

int read_data_file(char *file_name,struct point2d **data_ptr) {
  int n_points,n_alloc;
  double x,y;
  FILE *stream;

  stream = fopen(file_name,"r");
  if (stream == NULL) {
    fprintf(stderr,"Can't open file '%s' for reading\n",file_name);
    return(-1);
  }
  n_points = 0;
  n_alloc = 0;
  *data_ptr = NULL;
```

Use stdio library routines for reading disk files
This function opens a file for reading as a stream of characters

Reading a Data File of Arbitrary Length

Source code file data_file.c, continued:

fscanf function in stdio library scans file characters and assigns values to variables

As more array elements are needed, reallocate larger memory blocks

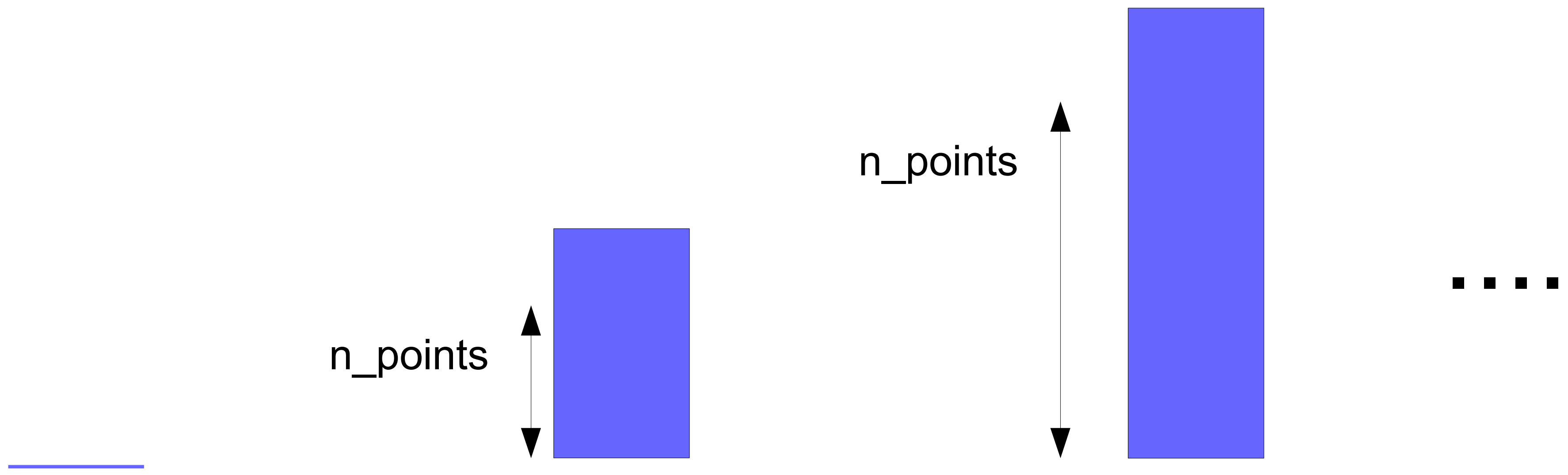
```
  while (fscanf(stream,"%lf %lf",&x,&y) == 2) {
    if (n_points >= n_alloc) {
      n_alloc += 10;
      *data_ptr = realloc(*data_ptr,n_alloc * sizeof(struct point2d));
    }
    (*data_ptr)[n_points].x = x;
    (*data_ptr)[n_points].y = y;
    n_points++;
  }
  fclose(stream);
  return(n_points);
}
```

Note: sizeof() operator works for structures as well as for simple data types

Don't forget to close any file you open!

Store each pair of coordinates read from file in next array element and increment count of elements

Memory Allocation Process



Initially no memory allocated, n_alloc=0, n_points=0

After first through tenth point read, n_alloc=10

After eleventh through twentieth point read, n_alloc=20

Interpolation Methods

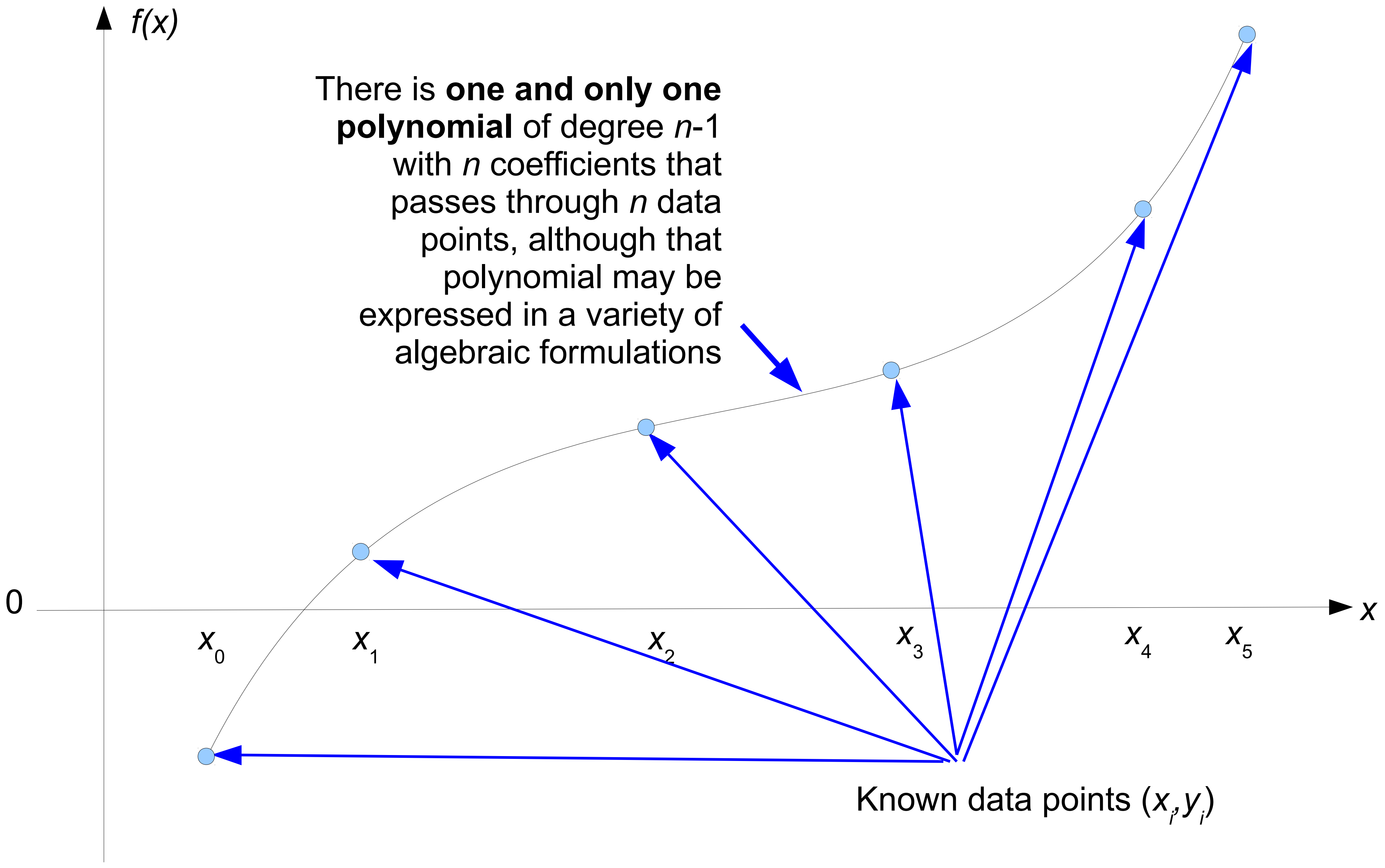
“Brute force”
Only applicable for small orders; solving for coefficients is numerically unstable for large orders
Calculate coefficients with system of linear equations; very simple polynomial evaluation (just nested Horner's algorithm)

Newton polynomial
Moderate computation needed to calculate coefficients; moderately simple polynomial evaluation

Lagrange polynomial
No computation needed to calculate coefficients; more expensive to evaluate polynomial

Choice of method determined by how many evaluations of polynomial are expected to be needed

Fitting Polynomial is Unique



There is **one and only one polynomial** of degree *n*-1 with *n* coefficients that passes through *n* data points, although that polynomial may be expressed in a variety of algebraic formulations

Brute Force Interpolation Polynomials

For *n* data points

$$(x_0, y_0) \text{ through } (x_{n-1}, y_{n-1})$$

Form a polynomial of order *n-1* with a canonical polynomial formulation:

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

Find *n* coefficients a_i by solving an *n* by *n* system of linear equations:

$$a_0 + x_0 a_1 + x_0^2 a_2 + x_0^3 a_3 + \cdots + x_0^{n-1} a_{n-1} = y_0$$

$$a_0 + x_1 a_1 + x_1^2 a_2 + x_1^3 a_3 + \cdots + x_1^{n-1} a_{n-1} = y_1$$

$$a_0 + x_2 a_1 + x_2^2 a_2 + x_2^3 a_3 + \cdots + x_2^{n-1} a_{n-1} = y_2$$

$$\vdots$$

$$a_0 + x_{n-1} a_1 + x_{n-1}^2 a_2 + x_{n-1}^3 a_3 + \cdots + x_{n-1}^{n-1} a_{n-1} = y_{n-1}$$

This is a particular case of the Vandermonde matrix, known to be badly conditioned, and troublesome to solve for large *n*

Evaluate polynomial efficiently with Horner's nested algorithm:

$$f(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x(a_n))\cdots))$$

Newton Interpolation Polynomial Examples

For 2 data points form a 1st order polynomial:

$$f(x)=a_0+\underbrace{a_1(x-x_0)}_{P_1(x)}$$

Note: These are not the same a_i values as in the canonical polynomial formulation!

For 3 data points form a 2nd order polynomial:

$$f(x)=a_0+\underbrace{a_1(x-x_0)}_{P_1(x)}+\underbrace{a_2(x-x_0)(x-x_1)}_{P_2(x)}$$

For 4 data points form a 3rd order polynomial:

$$f(x)=a_0+\underbrace{a_1(x-x_0)}_{P_1(x)}+\underbrace{a_2(x-x_0)(x-x_1)}_{P_2(x)}+\underbrace{a_3(x-x_0)(x-x_1)(x-x_2)}_{P_3(x)}$$

General Newton Interpolation Polynomials

For *n* data points form a polynomial of order *n-1*:

$$f(x) = a_0 + \sum_{i=1}^{n-1} a_i \cdot P_i(x)$$

$$\text{where } P_i(x) = \prod_{j=0}^{i-1} (x - x_j)$$

Solving for Newton Polynomial Coefficients

$$f(x_0)=y_0=a_0$$

a_i coefficients may be solved for recursively with simple algorithm

$$f(x_1)=y_1=a_0+a_1(x_1-x_0)$$

$$\rightarrow \quad a_1=\frac{y_1-a_0}{(x_1-x_0)}$$

$$f(x_2)=y_2=a_0+a_1(x_2-x_0)+a_2(x_2-x_0)(x_2-x_1)$$

$$\rightarrow \quad a_2=\frac{y_2-a_0-a_1(x_2-x_0)}{(x_2-x_0)(x_2-x_1)}$$

$$f(x_3)=y_3=a_0+a_1(x_3-x_0)+a_2(x_3-x_0)(x_3-x_1)+a_3(x_3-x_0)(x_3-x_1)(x_3-x_2)$$

$$\rightarrow \quad a_3=\frac{y_3-a_0-a_1(x_3-x_0)-a_2(x_3-x_0)(x_3-x_1)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}$$

Notation for Newton Polynomial Coefficients

$$
\begin{aligned}
&a_0 \text{ depends on } f(x_0)=y_0\\
&a_1 \text{ depends on } f(x_1)=y_1 \text{ and } f(x_0)=y_0\\
&a_2 \text{ depends on } f(x_2)=y_2 \text{ and } f(x_1)=y_1 \text{ and } f(x_0)=y_0\\
&\qquad\vdots\\
&a_k \text{ depends on } f(x_k)=y_k \text{ and } \cdots \text{ and } f(x_2)=y_2 \text{ and } f(x_1)=y_1 \text{ and } f(x_0)=y_0
\end{aligned}
$$

So introduce the notation

$$a_k = f[x_{0,} x_{1,} x_{2,} \cdots, x_k]$$

See text for proof of the recursive property of divided differences:

$$f[x_{0,} x_{1,} x_{2,} x_{3,} \cdots, x_k] = \frac{f[x_{1,} x_{2,} x_{3,} \cdots, x_k] - f[x_{0,} x_{1,} x_{2,} \cdots, x_{k-1}]}{x_k - x_0}$$

This recursive property can be exploited to construct a simpler table algorithm for solving for the a_i coefficients

Construct "Divided Difference" Table

x_0	$f[x_0]$	$f[x_0, x_1]$	$f[x_0, x_1, x_2]$		$f[x_0, x_1, \cdots, x_n]$
x_1	$f[x_1]$	$f[x_1, x_2]$	$f[x_1, x_2, x_3]$		
x_2	$f[x_2]$	$f[x_2, x_3]$	$\vdots$	....	
$\vdots$	$\vdots$	$\vdots$	$f[x_{n-2}, x_{n-1}, x_n]$		
x_{n-1}	$f[x_{n-1}]$	$f[x_{n-1}, x_n]$			
x_n	$f[x_n]$				

Construct "Divided Difference" Table

$$\begin{tikzcd}
x_0 & f[x_0] & f[x_0,x_1] \arrow[r] & f[x_0,x_1,x_2] & & f[x_0,x_1,\cdots,x_n] \\
x_1 & f[x_1] & f[x_1,x_2] \arrow[ur] \arrow[r] & f[x_1,x_2,x_3] & \cdots\cdot & \\
x_2 & f[x_2] & f[x_2,x_3] \arrow[ur] & \vdots & & \\
\vdots & \vdots & \vdots & f[x_{n-2},x_{n-1},x_n] & & \\
x_{n-1} & f[x_{n-1}] & f[x_{n-1},x_n] & & & \\
x_n & f[x_n] & & & &
\end{tikzcd}$$

Numerator terms come from previous column

Construct “Divided Difference” Table

$$\begin{tikzcd}
x_0 \arrow[rrr] & f[x_0] & f[x_0,x_1] & f[x_0,x_1,x_2] & & f[x_0,x_1,\cdots,x_n] \\
x_1 \arrow[rrr] & f[x_1] & f[x_1,x_2] & f[x_1,x_2,x_3] & & \\
x_2 \arrow[uurrr] & f[x_2] & f[x_2,x_3] & \vdots & \cdots\cdots & \\
\vdots \arrow[uurrr] & \vdots & \vdots & f[x_{n-2},x_{n-1},x_n] & & \\
x_{n-1} & f[x_{n-1}] & f[x_{n-1},x_n] & & & \\
x_n & f[x_n] & & & &
\end{tikzcd}$$

Denominator terms come from first *x* column

Newton Coefficients End Up on Top Row

$$
\begin{array}{c|ccccc|}
 & a_0 & a_1 & a_2 & \cdots & a_n \\
 & \downarrow & \downarrow & \downarrow & & \downarrow \\
\cline{2-6}
x_0 & f[x_0] & f[x_0, x_1] & f[x_0, x_1, x_2] & & f[x_0, x_1, \cdots, x_n] \\
\cline{2-6}
x_1 & f[x_1] & f[x_1, x_2] & f[x_1, x_2, x_3] & \cdots & \\
x_2 & f[x_2] & f[x_2, x_3] & \vdots & & \\
\vdots & \vdots & \vdots & f[x_{n-2}, x_{n-1}, x_n] & & \\
x_{n-1} & f[x_{n-1}] & f[x_{n-1}, x_n] & & & \\
x_n & f[x_n] & & & &
\end{array}
$$

Relationship between Divided Differences and Derivatives

For $f^{(n)}$ continuous over $[a,b]$ and points $x_{0,} x_{1,} x_{2,} \cdots, x_n$ in $[a,b]$

$$f[x_{0,} x_{1,} x_{2,} \cdots, x_n] = \frac{1}{n!} f^{(n)}(\xi)$$

for some ξ in (a,b)

So extending an interpolation to higher orders involves higher order derivatives of the original function relationship

Lagrangian Interpolation Polynomial Examples

For 2 data points form a 1st order polynomial:

$$f(x) = \underbrace{\frac{x-x_1}{x_0-x_1}}_{L_0(x)} \cdot a_0 + \underbrace{\frac{x-x_0}{x_1-x_0}}_{L_1(x)} \cdot a_1$$

Note: These are not the same a_i values as in the canonical or Newtonian polynomial formulation!

Lagrangian Interpolation Polynomial Examples

For 3 data points form a 2nd order polynomial:

$$f(x)=\underbrace{\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}}_{L_0(x)}\cdot a_0+\underbrace{\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}}_{L_1(x)}\cdot a_1+\underbrace{\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}}_{L_2(x)}\cdot a_2$$

General Lagrangian Interpolation Polynomials

For *n* data points form a polynomial of order *n-1*:

$$f(x) = \sum_{i=0}^{n-1} L_i(x) \cdot a_i$$

$$\text{where } L_i(x) = \prod_{j=0,\, j\neq i}^{n-1} \frac{x - x_j}{x_i - x_j}$$

Note: $L_i(x)$ is an $(n-1)$th order polynomial

$L_i(x) = 1$ at $x = x_i$ and $L_i(x) = 0$ at $x = x_j \forall J \neq i$

“Solving” for Lagrange Polynomial Coefficients

$$f(x_0) = y_0 = a_0$$

$$f(x_1) = y_1 = a_1$$

$$f(x_2) = y_2 = a_2$$

$$f(x_3) = y_3 = a_3$$

$$\vdots$$

So simply identify $a_i = y_i$, with no computation necessary

General Lagrangian Interpolation Polynomials

So general formula for *n* data points is simply:

$$f(x)=\sum_{i=0}^{n-1} L_i(x)\cdot y_i$$

$$\text{where } L_i(x)=\prod_{j=0,\, j\neq i}^{n-1} \frac{x-x_j}{x_i-x_j}$$

Conditional Compilation

Example program conditional.c :

```
#include <stdio.h>
#include <stdlib.h>
#define COND0 0
#define COND1 1
#define SYM1 123

int main() {
#if 1
  printf("Condition 1\n");
#endif
#if 0
  printf("Condition 2\n");
#endif
#if COND0
  printf("Condition 3\n");
#endif
#if COND1
  printf("Condition 4\n");
#endif
#ifdef SYM1
  printf("SYM1 defined and = %d\n",SYM1);
#endif
#ifdef SYM2
  printf("SYM2 defined and = %d\n",SYM2);
#endif
#ifdef SYM3
  printf("SYM3 defined and = %d\n",SYM3);
#endif
#ifdef SYM4
  printf("SYM4 defined and = %f\n",SYM4);
#endif
  exit(0);
}
```

Compilation Time Options

```
$ gcc -Wall -o conditional conditional.c
$ ./conditional
Condition 1
Condition 4
SYM1 defined and = 123
$ gcc -Wall -DSYM3 -o conditional conditional.c
$ ./conditional
Condition 1
Condition 4
SYM1 defined and = 123
SYM3 defined and = 1
$ gcc -Wall -DSYM4=456.789 -o conditional conditional.c
$ ./conditional
Condition 1
Condition 4
SYM1 defined and = 123
SYM4 defined and = 456.789000
$
```

interpolate.c Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "data_file.h"
#include "interpolate.h"

/* pointer to divided difference table, div_diff[i][j] = row i, column j*/
double **div_diff = NULL;

double lagrange(int n,struct point2d *sample,double x) {
  double sum,product;
  int i,j;

  sum = 0.0;
  i = 0;
  while (i < n) {
    product = sample[i].y;
    j = 0;
    while (j < n) {
      if (j != i) {
        product *= (x - sample[j].x) / (sample[i].x - sample[j].x);
      }
      j++;
    }
    sum += product;
    i++;
  }
  return(sum);
}
```

interpolate.c Code (continued)

```
double **alloc_tri(int n) {
  int i;
  double **tri;

  tri = (double **) malloc(n * sizeof(double *));
  if (tri == NULL) return(NULL);
  i = 0;
  while (i < n) {
    tri[i] = (double *) malloc((n-i) * sizeof(double));
    i++;
  }
  return(tri);
}

void free_tri(int n,double **tri) {
  int i;
  i = 0;
  while (i < n) {
    if (tri[i]) free(tri[i]);
    i++;
  }
  free(tri);
  return;
}
```

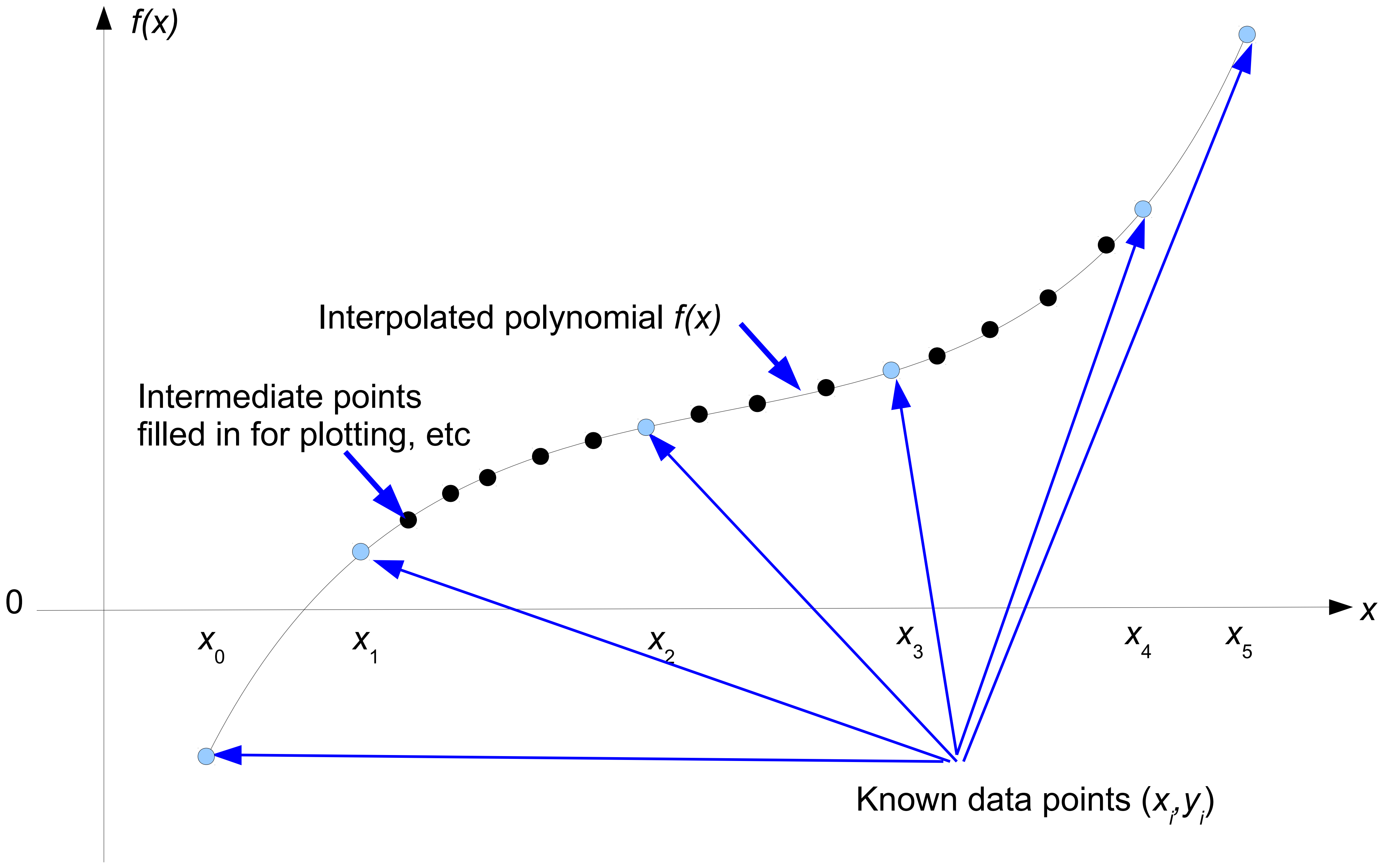
interpolate.c Code (continued)

```
int find_newton_coeffs(int n,struct point2d *sample) {
  int i,j;

  if (n < 2) return(-1);
  if (div_diff != NULL) free_tri(n,div_diff);
  div_diff = alloc_tri(n);
  i = 0;  /*initialize first column to be data sample y values*/
  while (i < n) {
    div_diff[i][0] = sample[i].y;
    i++;
  }
  j = 1;  /*calculate second through nth column*/
  while (j < n) {
    i = 0;
    while (i < (n - j)) {
      div_diff[i][j] = (div_diff[i + 1][j - 1] - div_diff[i][j - 1]) / 
(sample[i + j].x - sample[i].x);
      i++;
    }
    j++;
  }
  dump_tri(n,div_diff);
  return(0);
}
```

interpolate.c Code (continued)

```
double newton(int n,struct point2d *sample,double x) {
  int j;
  double p;

  if (n < 2) return(0.0);
  if (div_diff == NULL) return(0.0);
  p = div_diff[0][n - 1];
  j = n - 2;
  while (j >= 0) {
    p = (p * (x - sample[j].x)) + div_diff[0][j];
    j--;
  }
  return(p);
}
```

Filling In Intermediate Polynomial Points

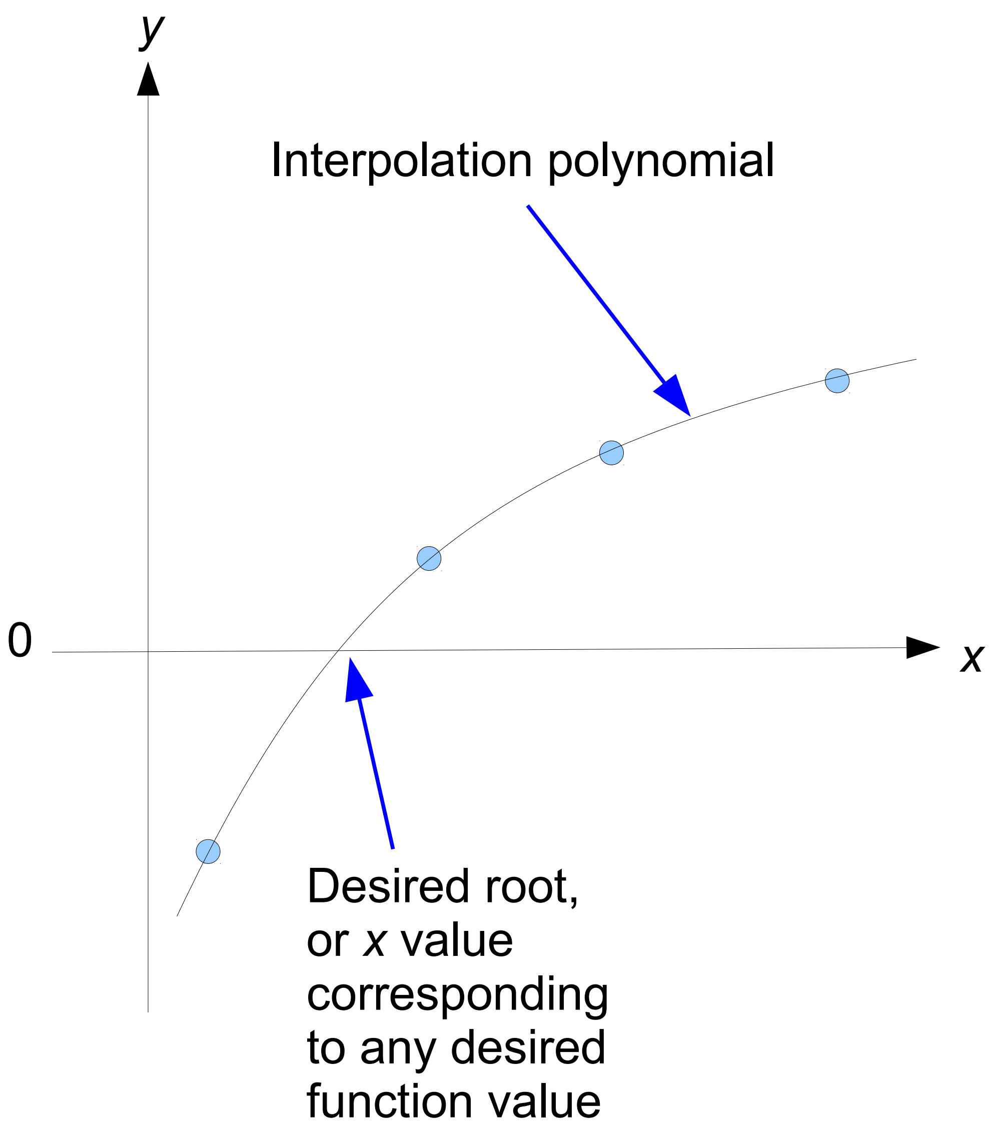


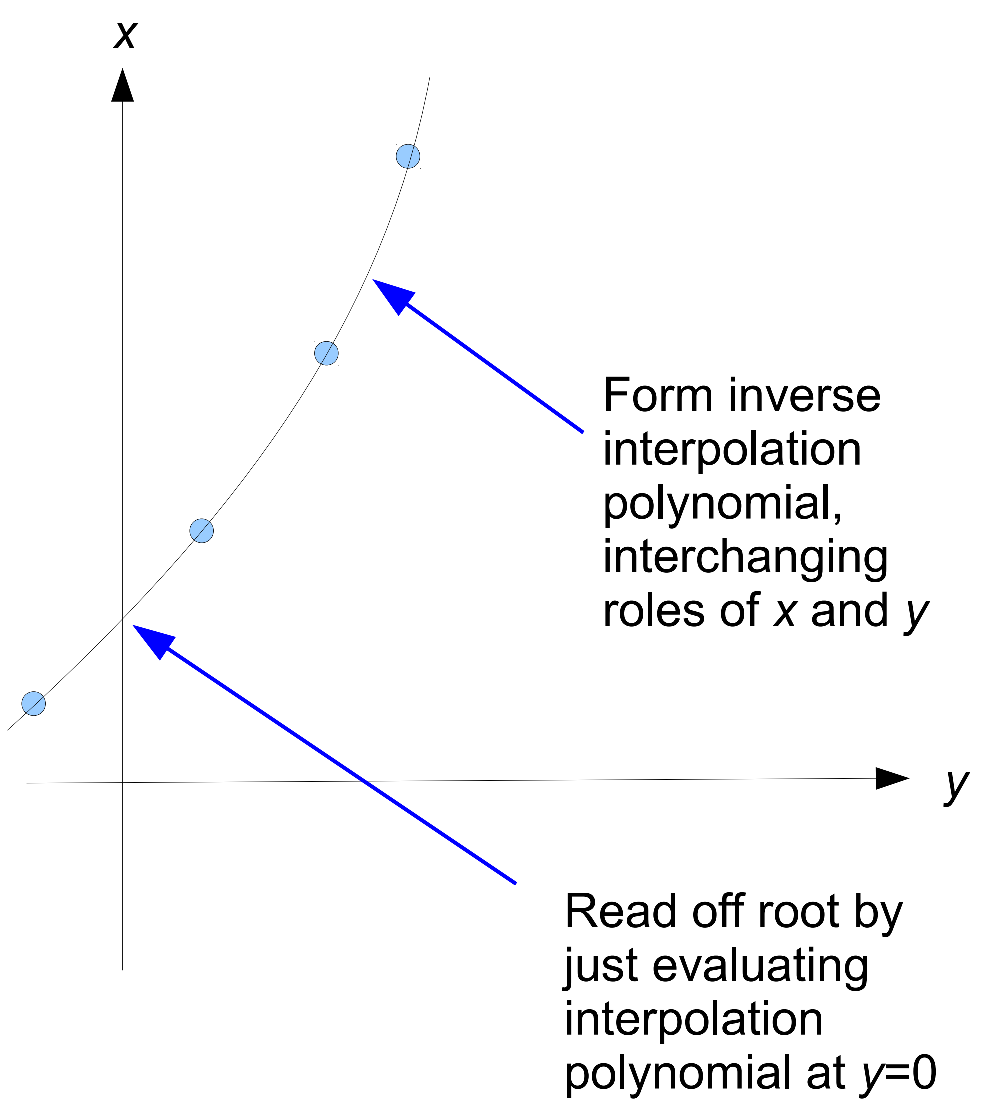
f(x)
Interpolated polynomial f(x)
Intermediate points filled in for plotting, etc
0
x
x_0
x_1
x_2
x_3
x_4
x_5
Known data points (x_i, y_i)

interpolate_sweep.c Code

```
#include <stdio.h>
#include <stdlib.h>
#include "data_file.h"
#include "interpolate.h"

struct point2d *data;

int main(int argc,char *argv[]) {
  double xstart,xstop,xinc,x;
  int n_points;

  if (argc != 5) {
    fprintf(stderr,"%s <input file> <xstart> <xstop> <xinc>\n",argv[0]);
    exit(1);
  }
  n_points = read_data_file(argv[1],&data);
  fprintf(stderr,"Read %d points from data file %s\n",n_points,argv[1]);
  xstart = atof(argv[2]);
  xstop = atof(argv[3]);
  xinc = atof(argv[4]);
#ifdef NEWTON
  find_newton_coeffs(n_points,data);
#endif
  xstop = xstop + (xinc * 0.5);
  x = xstart;
  while (((xinc > 0.0) && (x < xstop)) || ((xinc < 0.0) && (x > xstop))) {
#ifdef LAGRANGE
    printf("%.8g %.8g\n",x,lagrange(n_points,data,x));
#endif
#if NEWTON
    printf("%.8g %.8g\n",x,newton(n_points,data,x));
#endif
    x = x + xinc;
  }
  exit(0);
}
```

Finding Roots with Inverse Interpolation

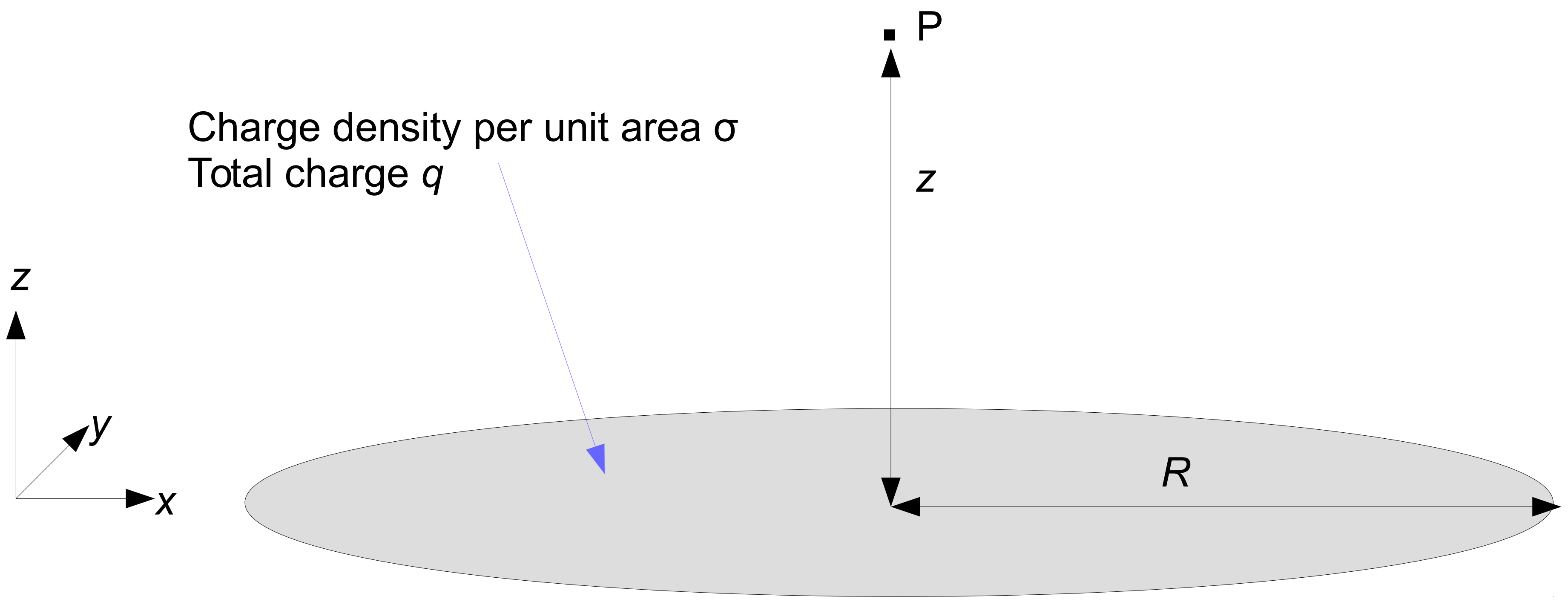

y
Interpolation polynomial
0
x
Desired root, or *x* value corresponding to any desired function value


x
Form inverse interpolation polynomial, interchanging roles of *x* and *y*
y
Read off root by just evaluating interpolation polynomial at *y*=0

interpolate_solve.c Code

```
#include <stdio.h>
#include <stdlib.h>
#include "data_file.h"
#include "interpolate.h"

struct point2d *data;

int main(int argc,char *argv[]) {
  double value;
  int i_start,n_points;

  if (argc != 3) {
    fprintf(stderr,"%s <input file> <value>\n",argv[0]);
    exit(1);
  }
  n_points = read_data_file(argv[1],&data);
  fprintf(stderr,"Read %d points from data file %s\n",n_points,argv[1]);
  value = atof(argv[2]);
  i_start = 0;
  while (i_start < n_points - 3) {
    if ((data[i_start + 1].y - value) * (data[i_start + 2].y - value) < 0.0) {
#ifdef NEWTON
      find_newton_inverse_coeffs(4,data + i_start);
#endif
#ifdef LAGRANGE
      printf("%.8g %.8g\n",lagrange_inverse(4,data + i_start,value),value);
#endif
#ifdef NEWTON
      printf("%.8g %.8g\n",newton_inverse(4,data + i_start,value),value);
#endif
    }
    i_start++;
  }
  exit(0);
}
```

Electrical Potential Above a Disc of Charge

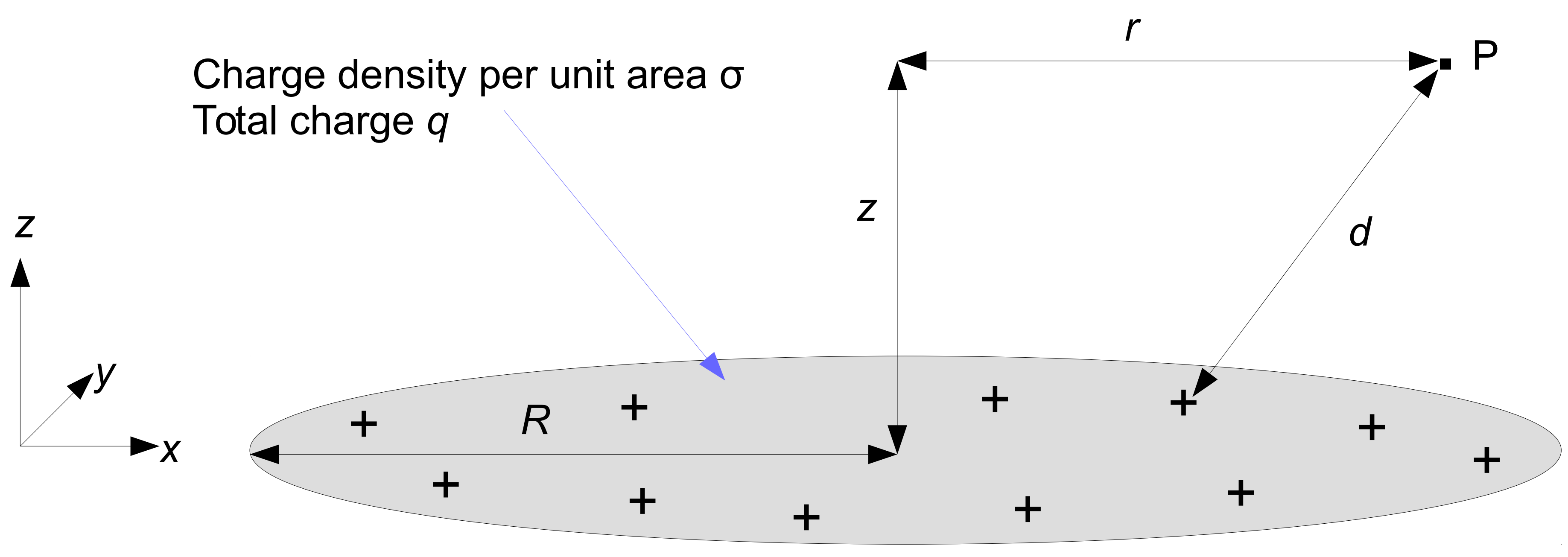


Assume the potential at an infinite distance is 0. Then from basic electrostatics, integrating concentric rings of charge with Coulomb's law, the electrical potential at a point P restricted to the axis of the disc is

$$V = \frac{\sigma}{2\epsilon_0}\left(\sqrt{z^2+R^2}-z\right) = \frac{q}{2\pi R^2\epsilon_0}\left(\sqrt{z^2+R^2}-z\right)$$

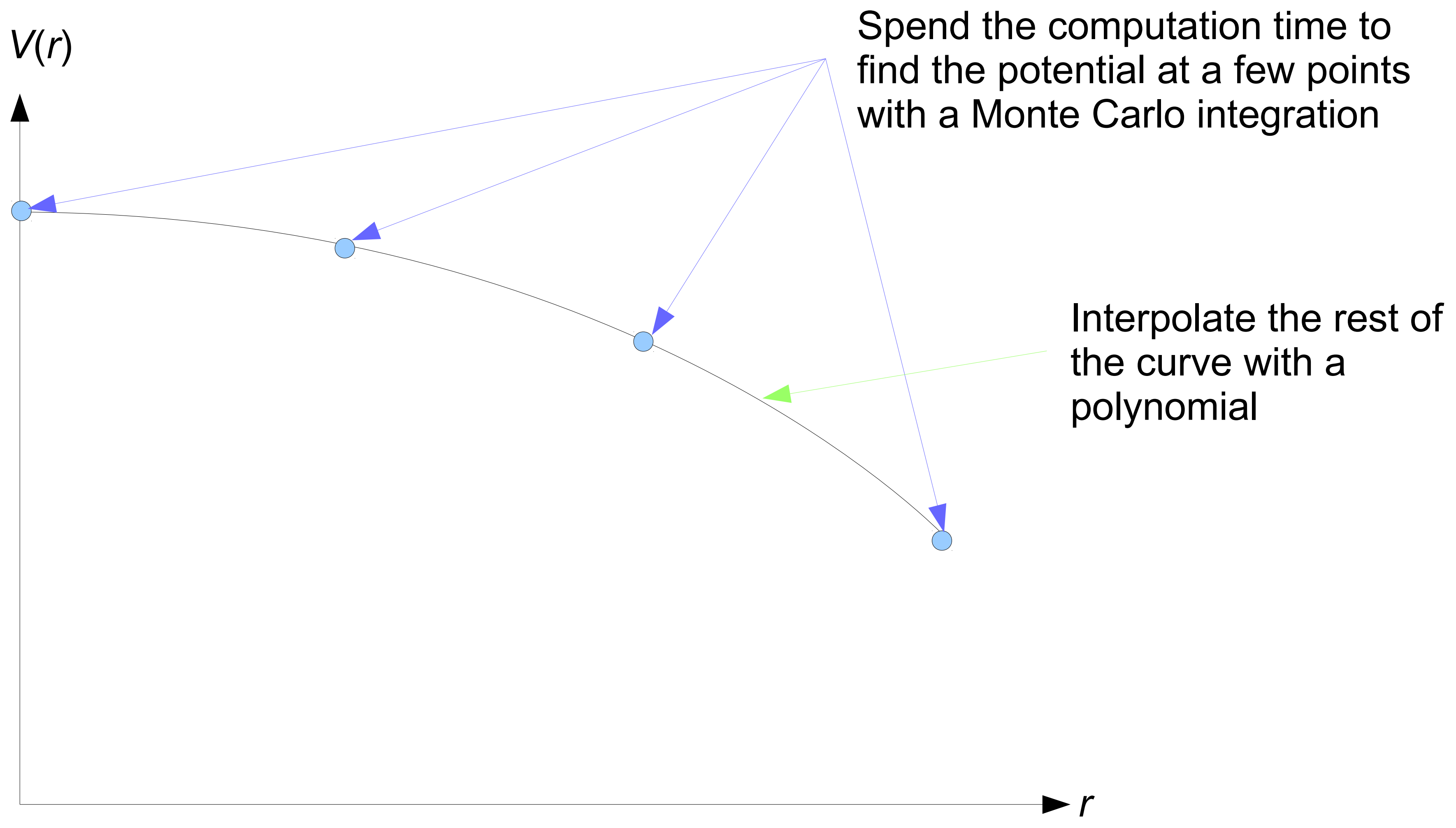
But what if the potential at a point off the axis is needed? Use a Monte Carlo approximation to the continuous integral

Electrical Potential Above a Disc of Charge

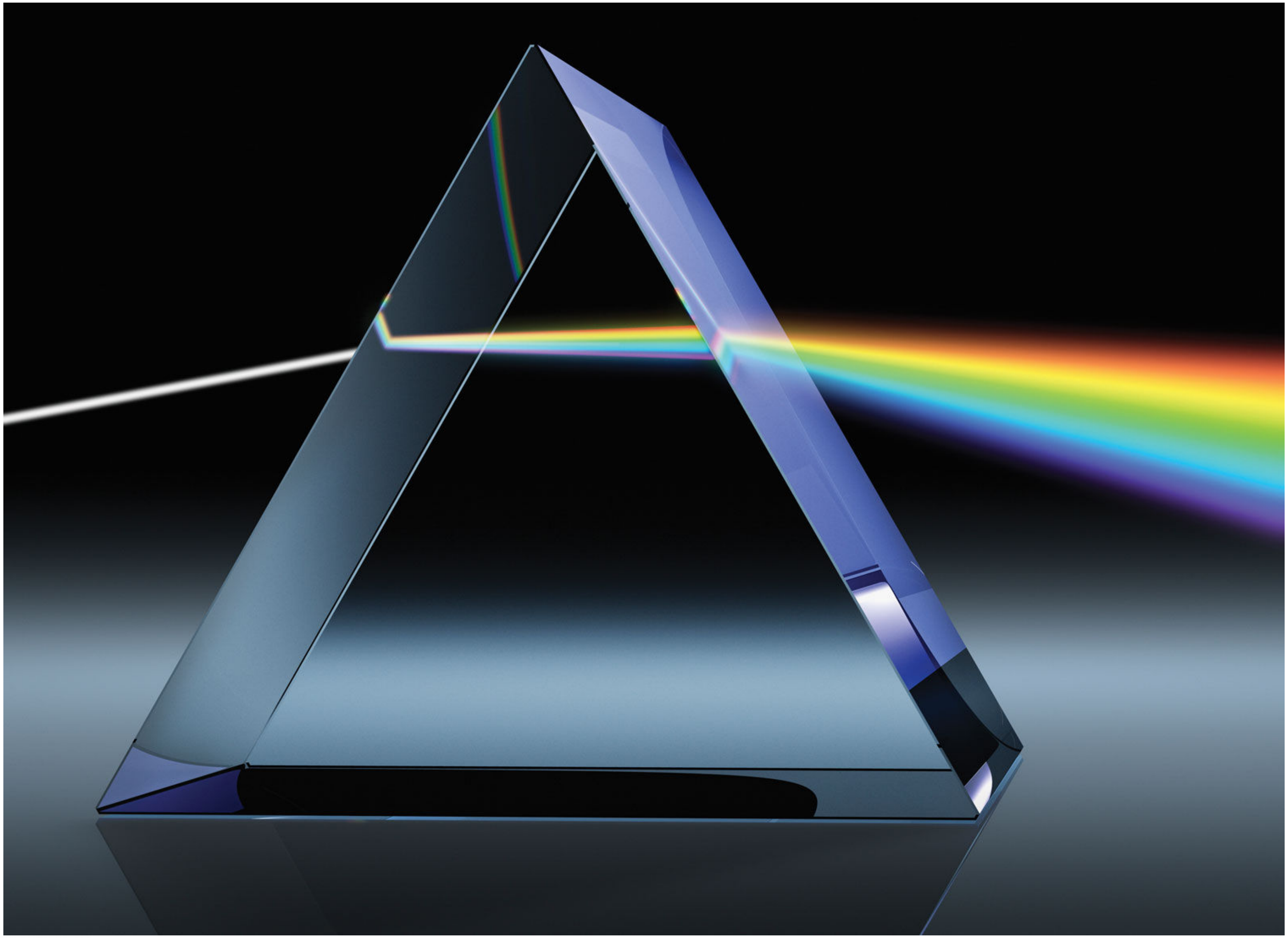


Place discrete portions of point charge randomly around disc with uniform distribution at some coordinates (*x*,*y*). Compute the contribution to the potential at point P, a distance d away. Sum up all the contributions from the portions of charge to make a discrete approximation to the continuous potential integral. An accurate calculation of the potential requires many random charges, and can take some significant computation time.

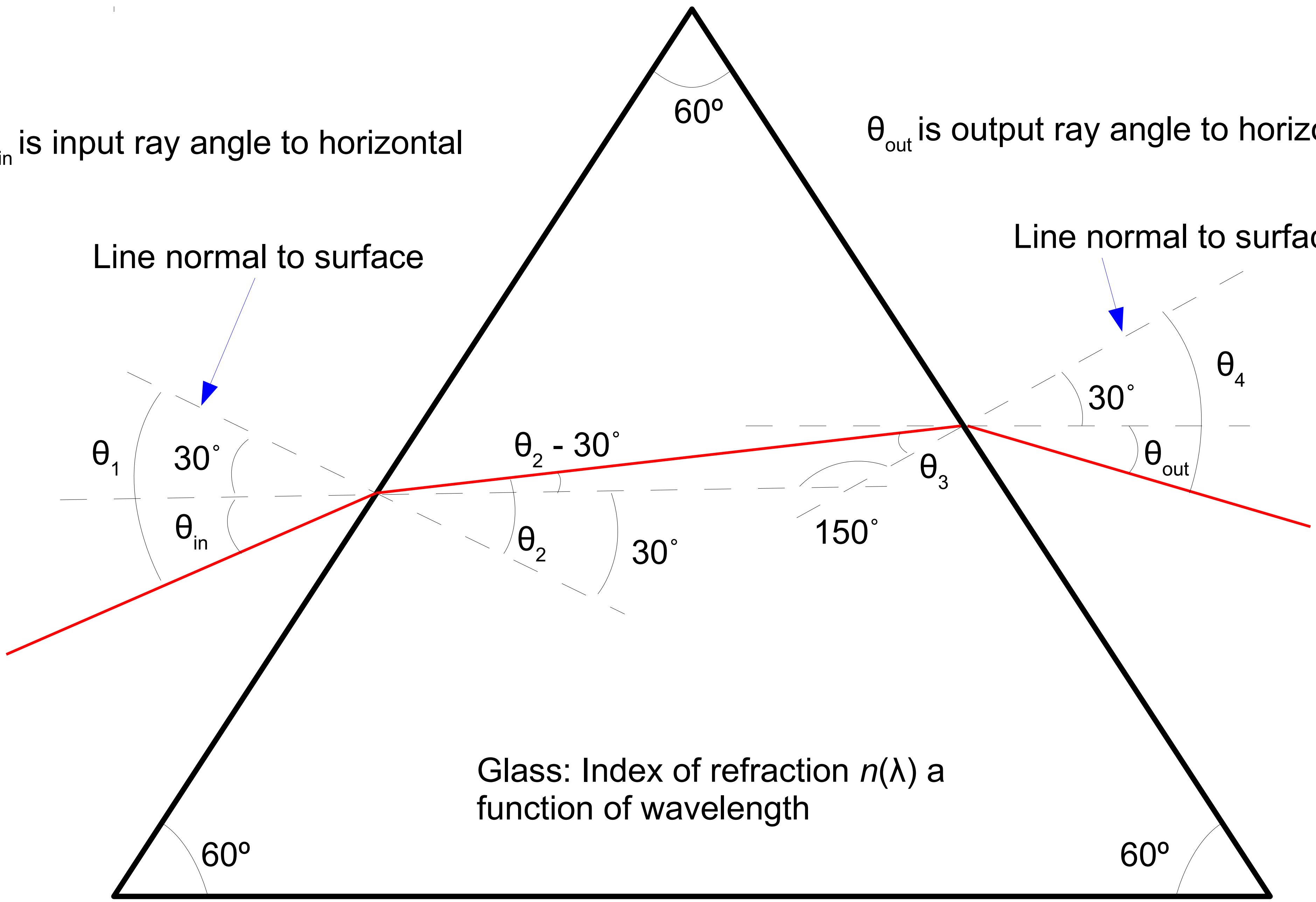
Interpolate the Continuous Potential Profile


$V(r)$
Spend the computation time to find the potential at a few points with a Monte Carlo integration
Interpolate the rest of the curve with a polynomial
r

Calculating Refraction through a Prism



Calculating Refraction through a Prism



60º
θ_{in} is input ray angle to horizontal
θ_{out} is output ray angle to horizontal
Line normal to surface
Line normal to surface
θ_4
30˚
θ_1
30˚
θ_2 - 30˚
θ_3
θ_{out}
θ_{in}
θ_2
30˚
150˚
Glass: Index of refraction $n(\lambda)$ a function of wavelength
60º
60º

Calculating Refraction through a Prism

$$\theta_1 = \theta_{in} + 30^\circ$$

From Snell's law $\sin(\theta_1) = n\sin(\theta_2)$ so $\theta_2 = \arcsin\left(\frac{1}{n}\sin(\theta_1)\right)$

From triangle law $\theta_3 = 180^\circ - (150^\circ + \theta_2 - 30^\circ) = 60^\circ - \theta_2$

From Snell's law $n\sin(\theta_3) = \sin(\theta_4)$ so $\theta_4 = \arcsin(n\sin(\theta_3))$

$$\theta_{out} = \theta_4 - 30^\circ$$

This algorithm will predict the output ray angle θ_{out} given an input ray angle θ_{in} and the index of refraction $n(\lambda)$. Assume we know $n(\lambda)$ only at a few measured light wavelenths.

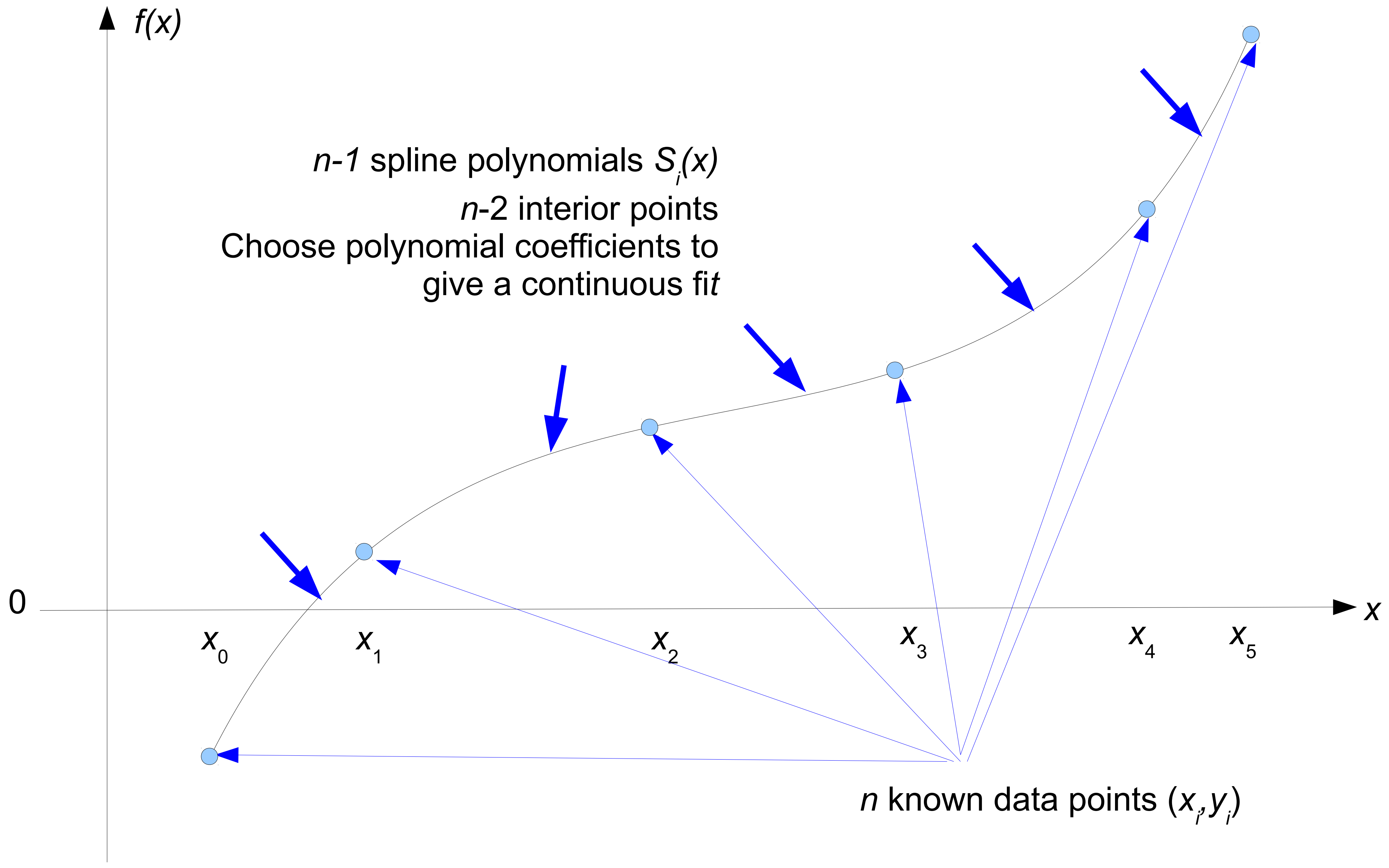
C Program Outline

```
struct point2d *data;
double theta_i;
int n_points;

double prism(double lambda) {
    ...
  n = newton(n_points,data,lambda);
    ...
  return(theta_o * RADTODEG);
}

int main(int argc,char *argv[]) {
    ...
  n_points = read_data_file(argv[1],&data);
    ...
  find_newton_coeffs(n_points,data);
  sweep(prism,lstart,lstop,lstep);
  exit(0);
}
```

Function to use an interpolated value of the index of refraction to calculate output angle

Main program reads measured *n* versus λ and sweeps wavelength to produce output file of output angle versus wavelength for gnuplot

Spline Fit to Known Data Points



$f(x)$
n-1 spline polynomials $S_i(x)$
n-2 interior points
Choose polynomial coefficients to
give a continuous fit
0
x
x_0
x_1
x_2
x_3
x_4
x_5
n known data points (x_i, y_i)

Total Coefficient Count for Cubic Splines

Each spline polynomial is assumed to be a cubic:

$$S_i = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

Each polynomial has four coefficients to choose, so we must have 4(*n*-1) constraints to solve for

The first condition is that each of the *n*-1 polynomials must pass through its two end points, giving 2(*n*-1) constraints

The second condition is that the first derivative of the spline must be continuous at each of the *n*-2 interior points, giving (*n*-2) constraints

The third condition is that the second derivative of the spline must be continuous at each of the *n*-2 interior points, giving another (*n*-2) constraints

Define the second derivative at the two end points to be zero for a “natural” cubic spline fit, giving two more constraints

Total constraints = 2(*n*-1) + (*n*-2) + (*n*-2) + 2 = 4(*n*-1)

Solving for Natural Cubic Spline Fit

Define $z_i = S''(x_i)$ for $0 \leqslant i \leqslant n-1$ to impose continuity in S''
By assumption $z_0 = z_{n-1} = 0$
Also define $h_i = x_{i+1} - x_i$
Spline fit will be defined if all z_i found

For a cubic polynomial, the second derivative is a linear function, so

$$S_i''(x) = \frac{z_{i+1}}{h_i}(x - x_i) + \frac{z_i}{h_i}(x_{i+1} - x)$$

Integrate twice, accumulate two constants of integration:

$$S_i(x) = \frac{z_{i+1}}{6h_i}(x - x_i)^3 + \frac{z_i}{6h_i}(x_{i+1} - x)^3 + C_i(x - x_i) + D_i(x_{i+1} - x)$$

Solving for Natural Cubic Spline Fit

Impose conditions $S_i(x_i)=y_i$ and $S_i(x_{i+1})=y_{i+1}$ to identify C_i and D_i :

$$S_i(x)=\frac{z_{i+1}}{6h_i}(x-x_i)^3 + \frac{z_i}{6h_i}(x_{i+1}-x)^3$$

$$+ \left(\frac{y_{i+1}}{h_i}-\frac{h_i z_{i+1}}{6}\right)(x-x_i) + \left(\frac{y_i}{h_i}-\frac{h_i z_i}{6}\right)(x_{i+1}-x)$$

Then impose condition $S'_{i-1}(x_i)=S'_i(x_i)$:

$h_{i-1}z_{i-1} + 2(h_{i-1}+h_i)z_i + h_i z_{i+1} = 6(b_i-b_{i-1})$ for $1\leqslant i\leqslant n-2$

where $b_i=\dfrac{y_{i+1}-y_i}{h_i}$

Also define $u_i=2h_{i-1}-h_i$ and $v_i=6(b_i-b_{i-1})$

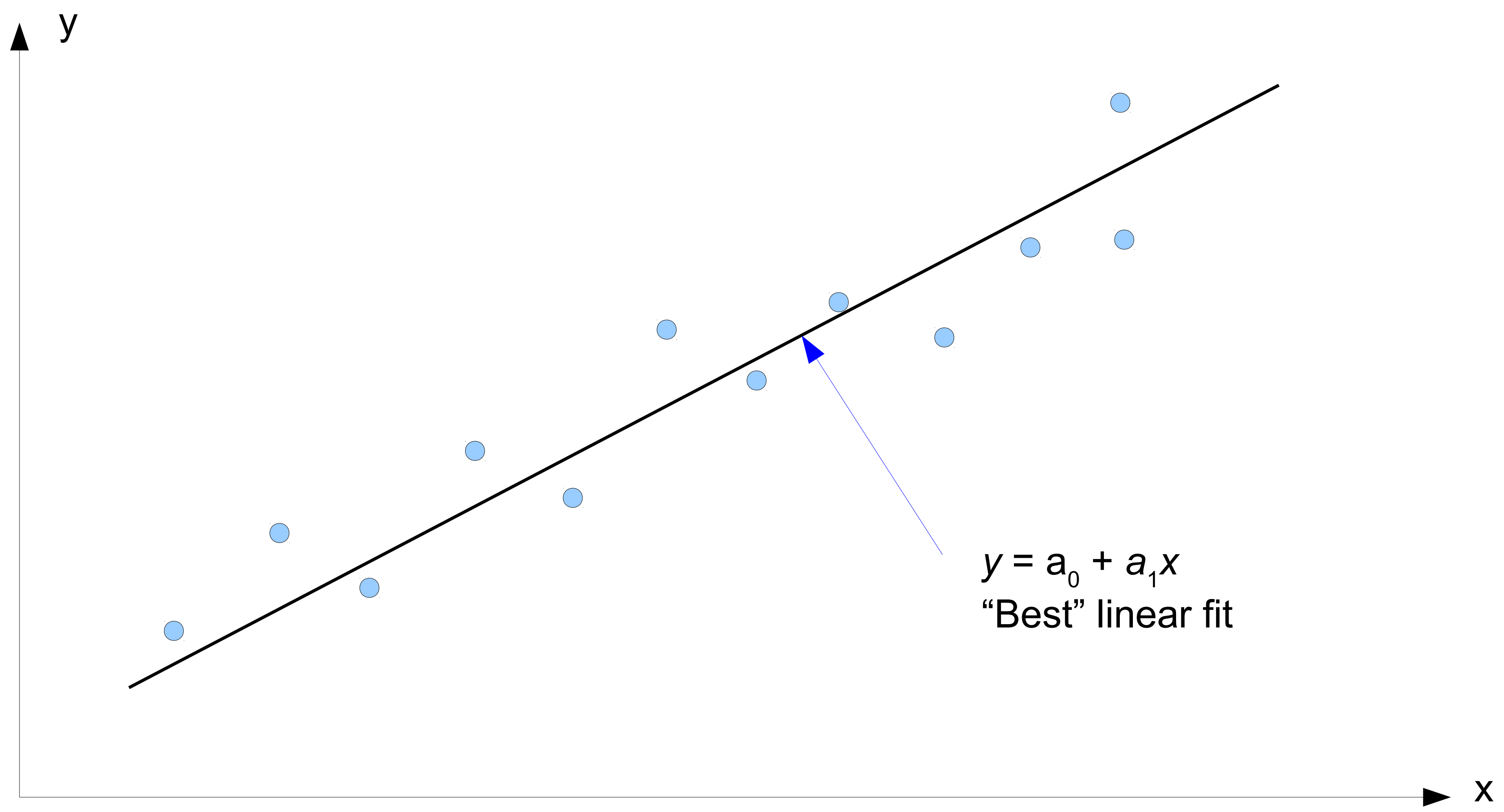
Solving for Natural Cubic Spline Fit

So resulting tridiagonal system of linear equations for z_i is

$$z_0 = 0$$
$$\vdots$$
$$h_{i-1} z_{i-1} + u_i z_i + h_i z_{i+1} = v_i \text{ for } 1 \leqslant i \leqslant n-2$$
$$\vdots$$
$$z_{n-1} = 0$$

Solve this system for z_i and plug back into expression for $S_i(x)$ above

Least Squares Fitting to a Straight Line



Solve for Linear Model Coefficients

Define error between the fit line and each data point as

$$e_i = \left| a_0 + a_1 x_i - y_i \right|$$

Define total fit error as the sum of the squared errors at each point:

$$E = \sum_{i=0}^{n-1} e_i^2 = \sum_{i=0}^{n-1} \left(a_0 + a_1 x_i - y_i \right)^2$$

Finding the 'best' fit is defined to be finding a_0 and a_1 that minimizes E

E is a function of a_0 and a_1 so the minimum E will be where

$$\frac{\partial E}{\partial a_0} = 0 \text{ and } \frac{\partial E}{\partial a_1} = 0$$

Solve for Linear Model Coefficients

First look at $\dfrac{\partial E}{\partial a_0} = 0$

$$\frac{\partial \sum_i (a_0 + a_1 x_i - y_i)^2}{\partial a_0} = 0$$

$$2 \sum_i^{n-1} (a_0 + a_1 x_i - y_i) = 0$$

$$n a_0 + a_1 \sum_i x_i - \sum_i y_i = 0$$

$$a_0 = \frac{\sum_i y_i - a_1 \sum_i x_i}{n}$$

Solve for Linear Model Coefficients

Then look at $\frac{\partial E}{\partial a_1}=0$

$$\frac{\partial \sum_i (a_0+a_1 x_i-y_i)^2}{\partial a_1}=0$$

$$2\sum_i^{n-1}(a_0+a_1 x_i-y_i)x_i=0$$

$$a_0\sum_i x_i \; + \; a_1\sum_i x_i^2 \; - \; \sum_i x_i y_i \; = \; 0$$

Substitute a_0 from previous page:

$$\frac{(\sum_i x_i)(\sum_i y_i)-a_1(\sum_i x_i)^2}{n} \; + \; a_1\sum_i x_i^2 \; - \; \sum_i x_i y_i \; = \; 0$$

Solve for Linear Model Coefficients

Solve for a_1 :

$$a_1 = \frac{n\sum_i (x_i y_i) - (\sum_i x_i)(\sum_i y_i)}{n\sum_i (x_i^2) - (\sum_i x_i)^2}$$

Then solve for a_0 :

$$a_0 = \frac{(\sum_i x_i^2)(\sum_i y_i) - (\sum_i x_i)(\sum_i x_i y_i)}{n\sum_i (x_i^2) - (\sum_i x_i)^2}$$

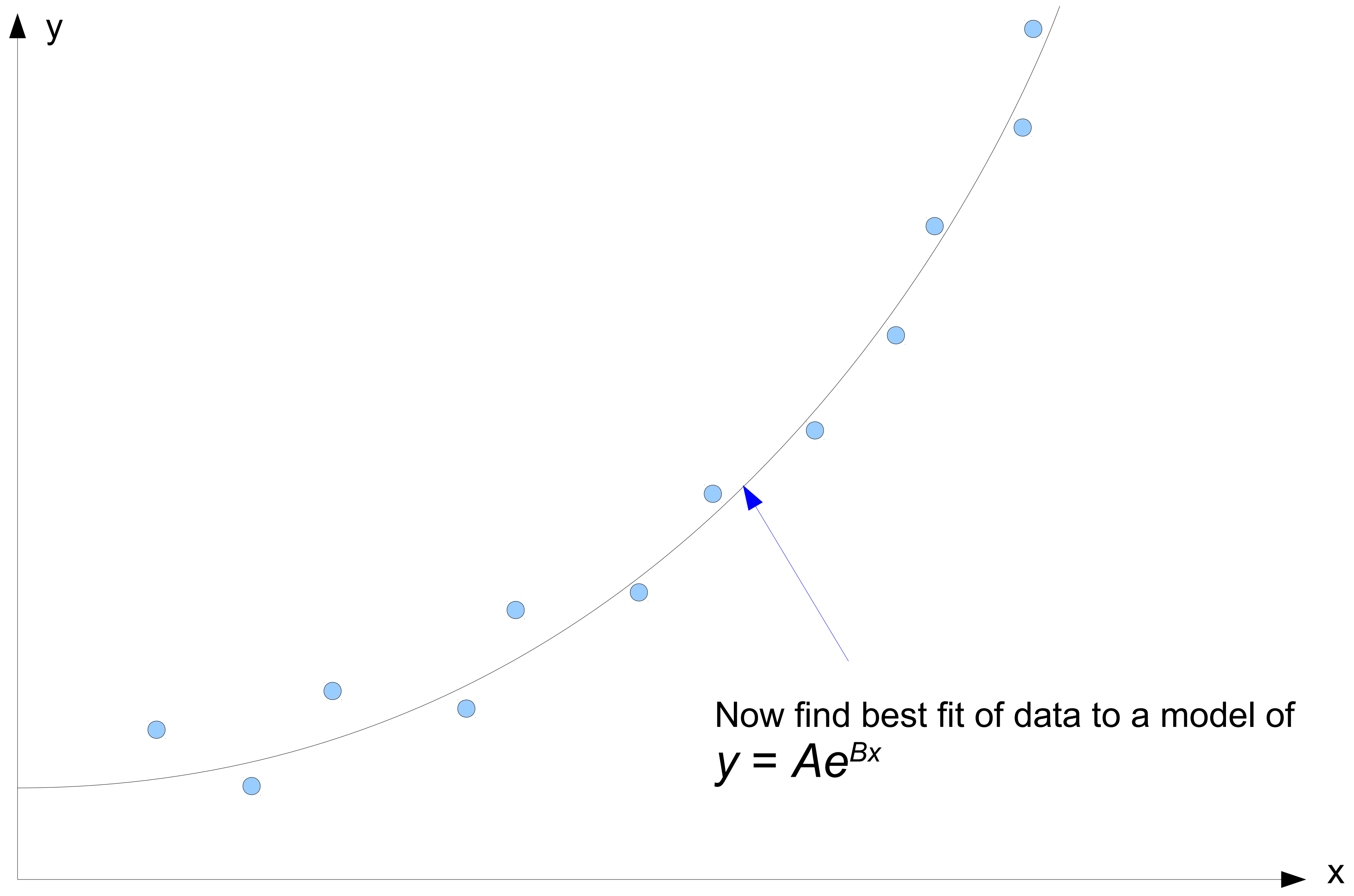
Code for Linear Least Squares Fitting

```
#include <stdio.h>
#include <stdlib.h>
#include "data_file.h"

int fitlinear(int n,double *a0ptr,double *a1ptr,struct point2d *sample){
  int i;
  double sumx,sumy,sumxy,sumxsq,denom,nd;
  sumx = 0.0;
  sumy = 0.0;
  sumxy = 0.0;
  sumxsq = 0.0;
  i = 0;
  while (i < n) {
    sumx += sample[i].x;
    sumy += sample[i].y;
    sumxy += sample[i].x * sample[i].y;
    sumxsq += sample[i].x * sample[i].x;
    i++;
  }
  nd = n;
  denom = (nd * sumxsq) - (sumx * sumx);
  *a0ptr = ((sumxsq * sumy) - (sumx * sumxy)) / denom;
  *a1ptr = ((nd * sumxy) - (sumx * sumy)) / denom;
  return(0);
}
```

Code for Linear Least Squares Fitting

```
#include <stdio.h>
#include <stdlib.h>
#include "data_file.h"
#include "fitlinear.h"

struct point2d *data;
int n_points;
double a0,a1;

int main(int argc,char *argv[]) {
  double xstart,xstop,xinc,x;

  if (argc != 5) {
    fprintf(stderr,"fitlinear_sweep <input file> <xstart> <xstop> <xinc>\n");
    exit(1);
  }
  n_points = read_data_file(argv[1],&data);
  fprintf(stderr,"Read %d points from data file %s\n",n_points,argv[1]);
  xstart = atof(argv[2]);
  xstop = atof(argv[3]);
  xinc = atof(argv[4]);
  fitlinear(n_points,&a0,&a1,data);
  fprintf(stderr,"Fit: a0=%.8g a1=%.8g\n",a0,a1);
  xstop = xstop + (xinc * 0.5);
  x = xstart;
  while (((xinc > 0.0) && (x < xstop)) || ((xinc < 0.0) && (x > xstop))) {
    printf("%.8g %.8g\n",x,a0 + (a1 * x));
    x = x + xinc;
  }
  exit(0);
}
```

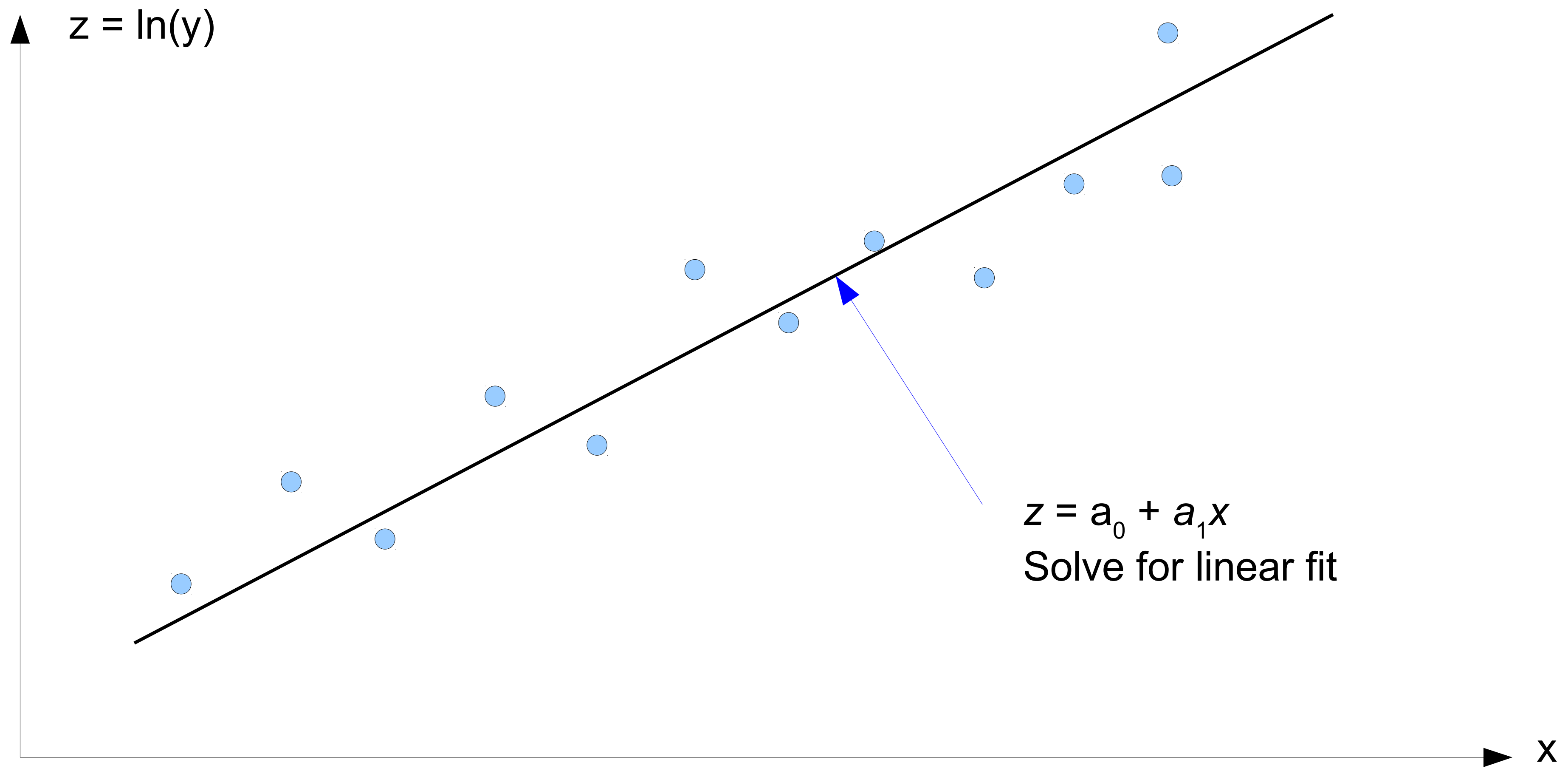
Applying Linear Fitting to a Nonlinear Models



Now find best fit of data to a model of

$y = Ae^{Bx}$

Change Variables and Fit to a Straight Line



Since $z = \ln(y) = \ln(A) + Bx$, assign $A = \exp(a_0)$ and $B = a_1$

Radioactive Decay

Start with a sample of material with N_0 radioactive nuclei at time $t = 0$.
At a given time in the future, the population of radioactive nuclei is $N(t)$.

Assume the probability of radioactive decay is such that the decay rate is

$$-\frac{dN}{dt} = \lambda N$$

$$\frac{dN}{N} = -\lambda\, dt$$

$$\int_{N_0}^{N} \frac{dN}{N} = -\lambda \int_{0}^{t} dt$$

$$\log(N) - \log(N_0) = -\lambda t$$

$$N(t) = N_0 e^{-\lambda t}$$

Radioactive Decay

But the measured quantity in a lab, for example with a Geiger counter, is the decay rate *R(t)*.

$$R(t) = -\frac{dN(t)}{dt} = \lambda N_0 e^{-\lambda t}$$

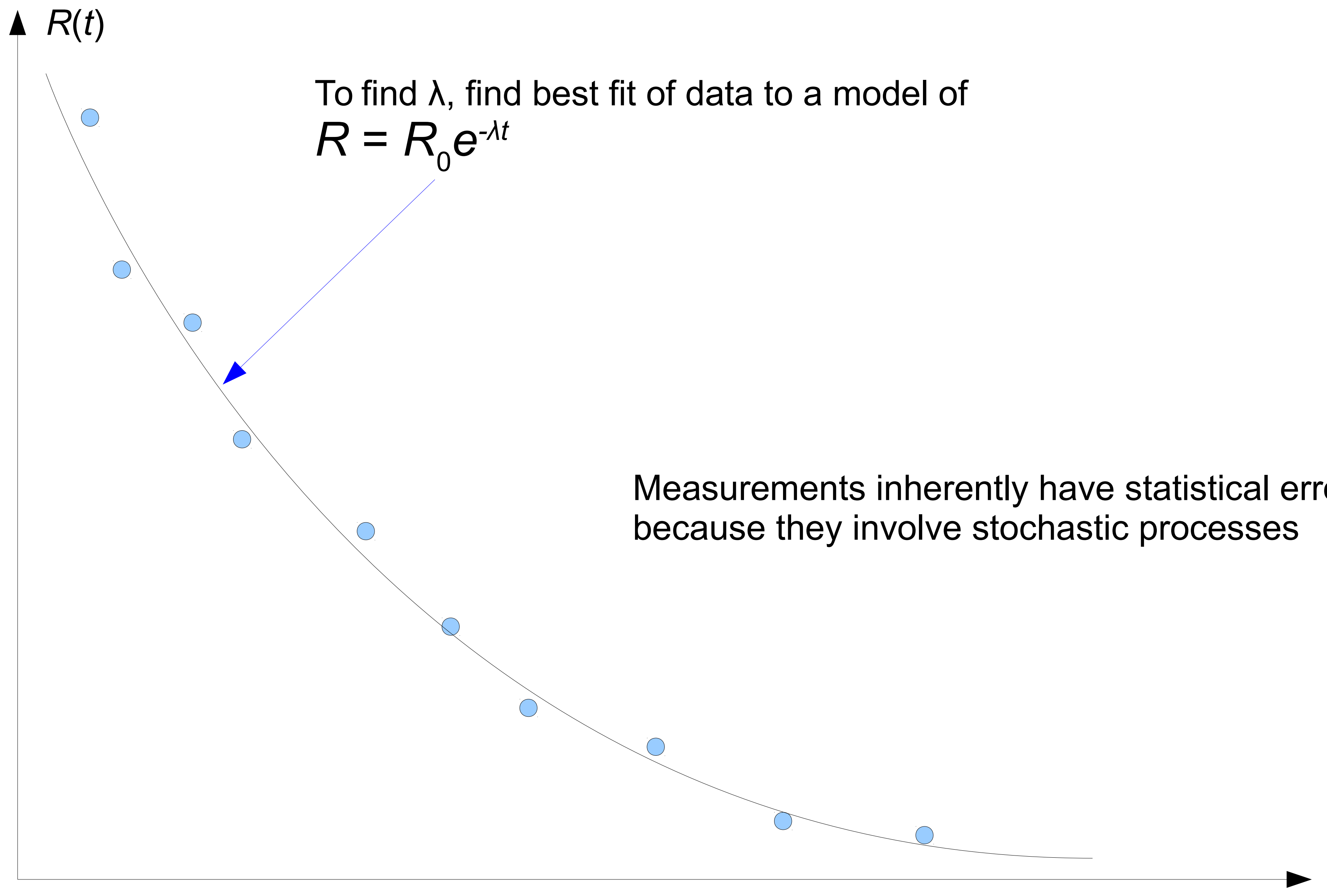
$$R(t) = R_0 e^{-\lambda t}$$

where R_0 is the measured decay rate at time t=0.

Usually a decay rate is specified as the “half-life” of a species, that is, the time for half the population to decay.

$$T_{1/2} = \frac{\ln(2)}{\lambda}$$

Radioactive Decay Model

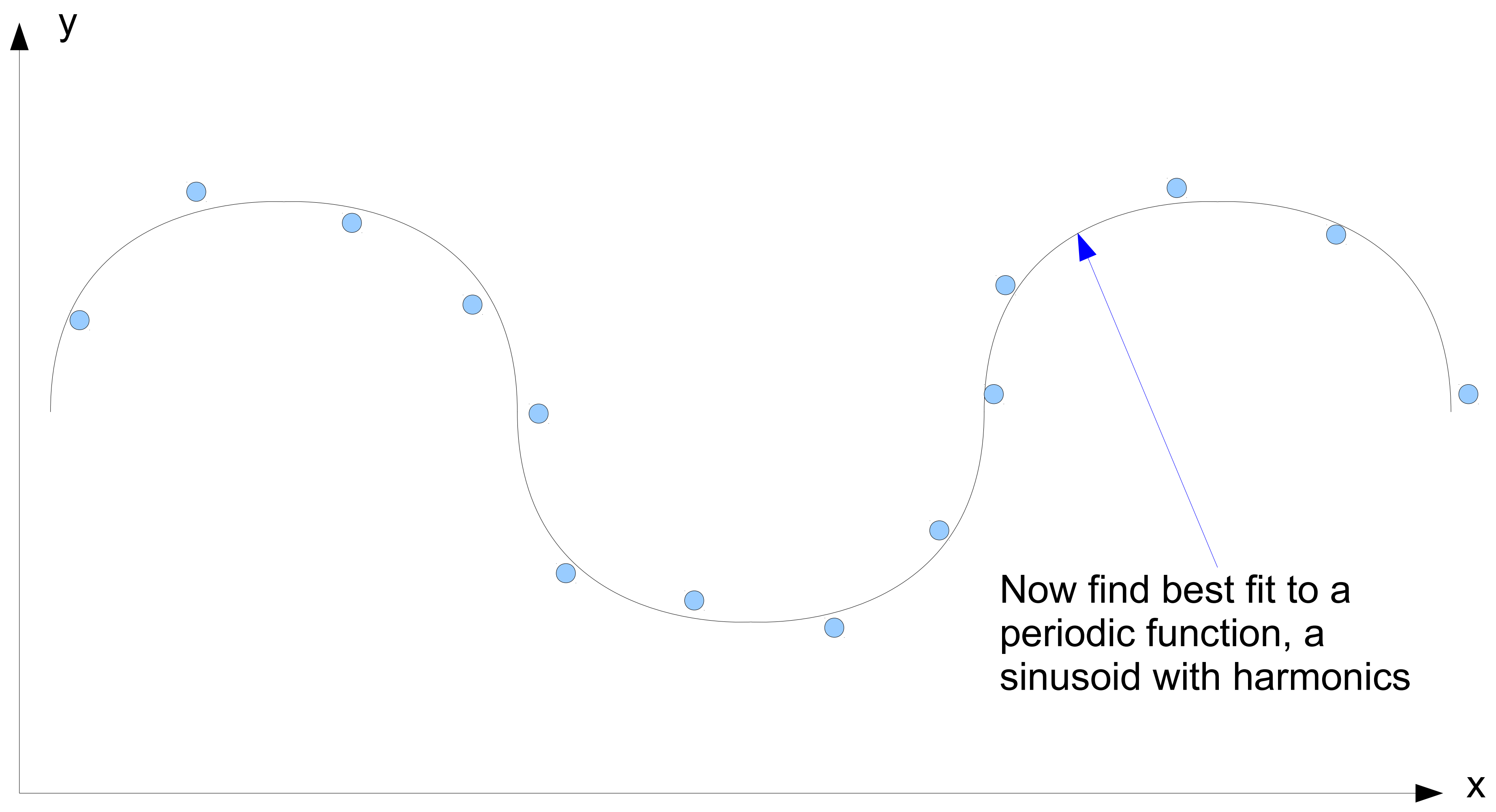


To find λ, find best fit of data to a model of

$R = R_0 e^{-\lambda t}$

Measurements inherently have statistical error because they involve stochastic processes

Fitting to Periodic Model



Now find best fit to a periodic function, a sinusoid with harmonics

Solve for Periodic Model Coefficients

Fit data to a general sinusoidal function with harmonics:

$f(x)=a_0+a_1\sin(2\pi f x)+a_2\cos(2\pi f x)+a_3\sin(4\pi f x)+a_4\cos(4\pi f x)\cdots$

or in general $f(x)=a_0\phi_0(x)+a_1\phi_1(x)+a_2\phi_2(x)+\cdots$

where the functions $\phi_i(x)$ are the basis functions for the fit

Define error between the fit line and each data point as

$$e_i=|f(x_i)-y_i|$$

Define total fit error as the sum of the squared errors at each point:

$$E = \sum_{i=0}^{n-1} e_i^2 = \sum_{i=0}^{n-1} (f(x_i)-y_i)^2$$

Finding the 'best' fit is defined to be finding a_i that minimizes E

E is a function of all a_i so the minimum E will be where

$$\frac{\partial E}{\partial a_0}=0 \quad \frac{\partial E}{\partial a_1}=0 \quad \frac{\partial E}{\partial a_2}=0\cdots$$

Solve for Periodic Model Coefficients

Look at $\frac{\partial E}{\partial a_j} = 0$

$$\frac{\partial \sum_i (f(x_i) - y_i)^2}{\partial a_j} = 0$$

$$2\sum_i^{n-1} (f(x_i) - y_i)\frac{\partial f(x_i)}{\partial a_j} = 0$$

But $\frac{\partial f(x)}{\partial a_j} = \phi_j(x)$

$$\sum_i f(x_i)\phi_j(x_i) - \sum_i y_i \phi_j(x_i) = 0$$

$$a_0 \sum_i \phi_0(x_i)\phi_j(x_i) + a_1 \sum_i \phi_1(x_i)\phi_j(x_i) + \cdots = \sum_i y_i \phi_j(x_i)$$

Matrix Formulation for Coefficients

$$\begin{bmatrix} \sum_i \phi_0(x_i)\phi_0(x_i) & \sum_i \phi_1(x_i)\phi_0(x_i) & \sum_i \phi_2(x_i)\phi_0(x_i) & \cdots \\ \sum_i \phi_0(x_i)\phi_1(x_i) & \sum_i \phi_1(x_i)\phi_1(x_i) & \sum_i \phi_2(x_i)\phi_2(x_i) & \cdots \\ \sum_i \phi_0(x_i)\phi_2(x_i) & \sum_i \phi_1(x_i)\phi_2(x_i) & \sum_i \phi_2(x_i)\phi_2(x_i) & \cdots \\ & & \vdots & \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} \sum_i y_i \phi_0(x_i) \\ \sum_i y_i \phi_1(x_i) \\ \sum_i y_i \phi_2(x_i) \\ \vdots \end{bmatrix}$$

The size of the linear system to be solved is the number of basis functions being combined to make the fitting function

Note: System matrix is symmetrical about its diagonal, so only half the entries need to be calculated

Code for Sinusoidal Least Squares Fitting

```c
#include <stdio.h>
#include <stdlib.h>
#include "data_file.h"
#include "lineq.h"

double phi(int,double);

int fitsine(int n,double *a,struct point2d *sample) {
  int i,j,k;
  double **coeff,rhs[7];

  coeff = alloc_matrix(7,7);
  for (j = 0; j < 7; j++) {
    for (k = 0; k < 7; k++) {
      coeff[j][k] = 0.0;
      for (i = 0; i < n; i++) {
        coeff[j][k] += phi(k,sample[i].x) * phi(j,sample[i].x);
      }
    }
    rhs[j] = 0.0;
    for (i = 0; i < n; i++) {
      rhs[j] += sample[i].y * phi(j,sample[i].x);
    }
  }
  if (gauss(7,coeff,rhs,a,1.0e-18)) {
    fprintf(stderr,"Singular coefficient matrix\n");
    free_matrix(7,coeff);
    exit(1);
  }
  free_matrix(7,coeff);
  return(0);
}
```

Code for Sinusoidal Least Squares Fitting

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "data_file.h"
#include "fitlinear.h"
#include "constants.h"

struct point2d *data;
int n_points;
double a[7],frequency;

double phi(int i,double x) {
  if (i == 0) return(1.0);
  if (i == 1) return(sin(2.0 * PI * frequency * x));
  if (i == 2) return(cos(2.0 * PI * frequency * x));
  if (i == 3) return(sin(4.0 * PI * frequency * x));
  if (i == 4) return(cos(4.0 * PI * frequency * x));
  if (i == 5) return(sin(6.0 * PI * frequency * x));
  return(cos(6.0 * PI * frequency * x));
}
```

Code for Sinusoidal Least Squares Fitting

```c
int main(int argc,char *argv[]) {
  double xstart,xstop,xinc,x,f;
  int i;

  if (argc != 6) {
    fprintf(stderr,"fitsine_sweep <input file> <frequency> <xstart> <xstop> <xinc>\n");
    exit(1);
  }
  n_points = read_data_file(argv[1],&data);
  fprintf(stderr,"Read %d points from data file %s\n",n_points,argv[1]);
  frequency = atof(argv[2]);
  xstart = atof(argv[3]);
  xstop = atof(argv[4]);
  xinc = atof(argv[5]);
  fitsine(n_points,a,data);
  for (i = 0; i < 7; i++) {
    fprintf(stderr,"a[%d]=%.8g ",i,a[i]);
  }
  putc('\n',stderr);
  xstop = xstop + (xinc * 0.5);
  x = xstart;
  while (((xinc > 0.0) && (x < xstop)) || ((xinc < 0.0) && (x > xstop))) {
    f = 0.0;
    for (i = 0; i < 7; i++) {
      f += a[i] * phi(i,x);
    }
    printf("%.8g %.8g\n",x,f);
    x = x + xinc;
  }
  exit(0);
}
```