Applied Programming

Reading and Parsing

with

`fgets()` and `strtok()`

Reading from text files

- When processing information from text files we often need to ***parse*** the text file ***line by line***
- For this task the **`scanf`** family is not very useful since it does not allow us to easily detect the end-of-lines
- Instead it is common to use **`fgets()`** from the **`stdio.h`** standard library

Note:

`fgets()` is a well known “not secure” function

fgets()

```
char *fgets(char *line, int n, FILE *fp);
```

- line - pointer to the read buffer
- n - size of the read buffer
- fp - pointer to an input stream
- Returns- pointer to “line” or NULL for empty
  - Uses stdio.h

- Notes:
  - **`fgets()`** will read up to **`n-1`** characters from fp
  - As soon as a **`newline`** or and **`end-of-file`** is encountered no additional characters are read and a **`NULL character`** is written to the end of the array.

`fgets()`: Example

- Using **`fgets()`**

```
#include <stdio.h>  /* for fgets() */
#include <string.h> /* for strlen */

#define BUFSIZE 4096
FILE *ifp;       /* pointer to input file stream */
char *buffer;    /* buffer to hold line read */
unsigned buflen; /* length of line read */
. . .
buffer = (char*) malloc(BUFSIZE); /* allocate buffer */
ifp = fopen (“myfyle”, "r");      /* open file */
/* read file line by line */
while ( fgets(buffer, BUFSIZE, ifp) != NULL ) {
  buflen = strlen(buffer);
  if ( 0 == buflen || '\n' != buffer[buflen-1]){
    printf ("ERROR: Wrong buffer !!\n");
    return 1;
  }
  /* do something with the line just read */
  printf("%s", buffer);
```

strtok()

```
char *strtok(char * s1, const char *s2);
```

- parses a string like scanf(), must be called iteratively
- s1 - ***string to be searched,*** ***then NULL,*** ***s1 is destroyed***
- s2 - argument is string of ***token separators***
- Returns – NULL when finished
  - Uses string.h

How it works:

- searches `s1` using characters in `s2`
- If `s1` contains one or more tokens, the char following the token is ***overwritten with a null character.***
  - the ***remainder of*** `s1` is ***stored*** elsewhere
- A ***pointer to the 1st character in the token is returned***.
- Subsequent calls with `s1` equal to `NULL` ***return*** a pointer to the ***next token***, etc. If no additional token are available `NULL` is returned

`strtok()`: Example

- Using **`strtok()`**

```c
#include <stdio.h>  /* for fgets() */
#include <string.h> /* for strlen */

unsigned count;   /* counter for tokens found */
char sep[] = " "; /* token separator is blank */
char *p,*buffer;  /* buffer and pointer to scan buffer */

count=0;
/* begin parsing */
p = strtok(buffer,sep);
while ( p != NULL ) { /* true if no tokens left */
  ++count;            /* count the number of tokens */
  /* do something with token */
  x = atof(p);        /* e.g., Convert string to double */
  printf("\"% 12.6g ", p); /* print it */
  p = strtok(NULL, sep); /* move on to next token*/
}
```

`Note: Full source in the appendix`

Clever use of strtok()

- Sometimes you have to allocate data arrays but you don't know how much data you have until you parse the entire input line.
  - You could dynamically allocate each element as you go along (a pain)
- OR
  - strcpy() the input data to make a 2nd copy
  - strtok() the 2nd copy to count the entries
  - allocate the data space
  - strtok() the original input data

Why do we need two copies?

https://yonocomprobasura.files.wordpress.com/2014/05/img_beneficios_del_huevo_25433_200.jpg

Example

rdnparse.c - Reads data in, parses it and converts to floating point

```
./rdnparse  parseData.txt
l 1: 1 2 3     4.5
   1.0000    2.0000    3.0000    4.5000    0.0000
Number of entries: 5
l 2: 7
   7.0000
Number of entries: 1
l 3:     08.3
   8.3000
Number of entries: 1
l 4: -9.7 - 10.1
  -9.7000    0.0000   10.1000
Number of entries: 3
l 5: bad bad bad
   0.0000    0.0000    0.0000
```

Data contains leading, trailing, embedded spaces AND invalid data (asci)

Note: The trailing space after 4.5 was converted to 0.0!

parseData.txt

```
1 2 3     4.5
7
   08.3
-9.7 - 10.1
bad bad bad
```

strtok() and trailing spaces

- strtok() will parse a trailing space as a zero
  - How do we fix this?
- Get rid of trailing spaces.
  - fgets() a data line
  - write “C” code to remove trailing space
- then use strtok()

Truncation code fragment

```
i = strlen(data)-1;                /* C is origin 0 */

/* Make sure we don't go negative */
while ((i >= 0) && (data [i] == ' ')) {
    data [i] = 0;
    i--;
  }
printf("'%s'\n", data);    /* Now data has no trailing spaces */
```

Note: This style of code is useful to remove other “bad” things like tab characters:

```
        if (data [i] == 0x09) {data [i] = ‘ ‘;}    /* 09 HEX is TAB */
```

More Examples

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) {
  FILE *handle;
  char *rc_p;
  char *string_p = (char *)malloc(256*sizeof(char)); /* no check *./
  handle = fopen("data.txt", "r");                    /* no check */

  /* Read a single line */
  while (fgets(string_p, 256, handle)){
     /* Parse the single line, string is DESTROYED*/
    rc_p = strtok(string_p, " ");       /*parse on the spaces */
     while (rc_p) {
        printf("Value %f ", atof(rc_p));
        rc_p = strtok(NULL, " ");
     }
    printf("\n");
  }
  return 0;  /* Should close and free */
  }
```

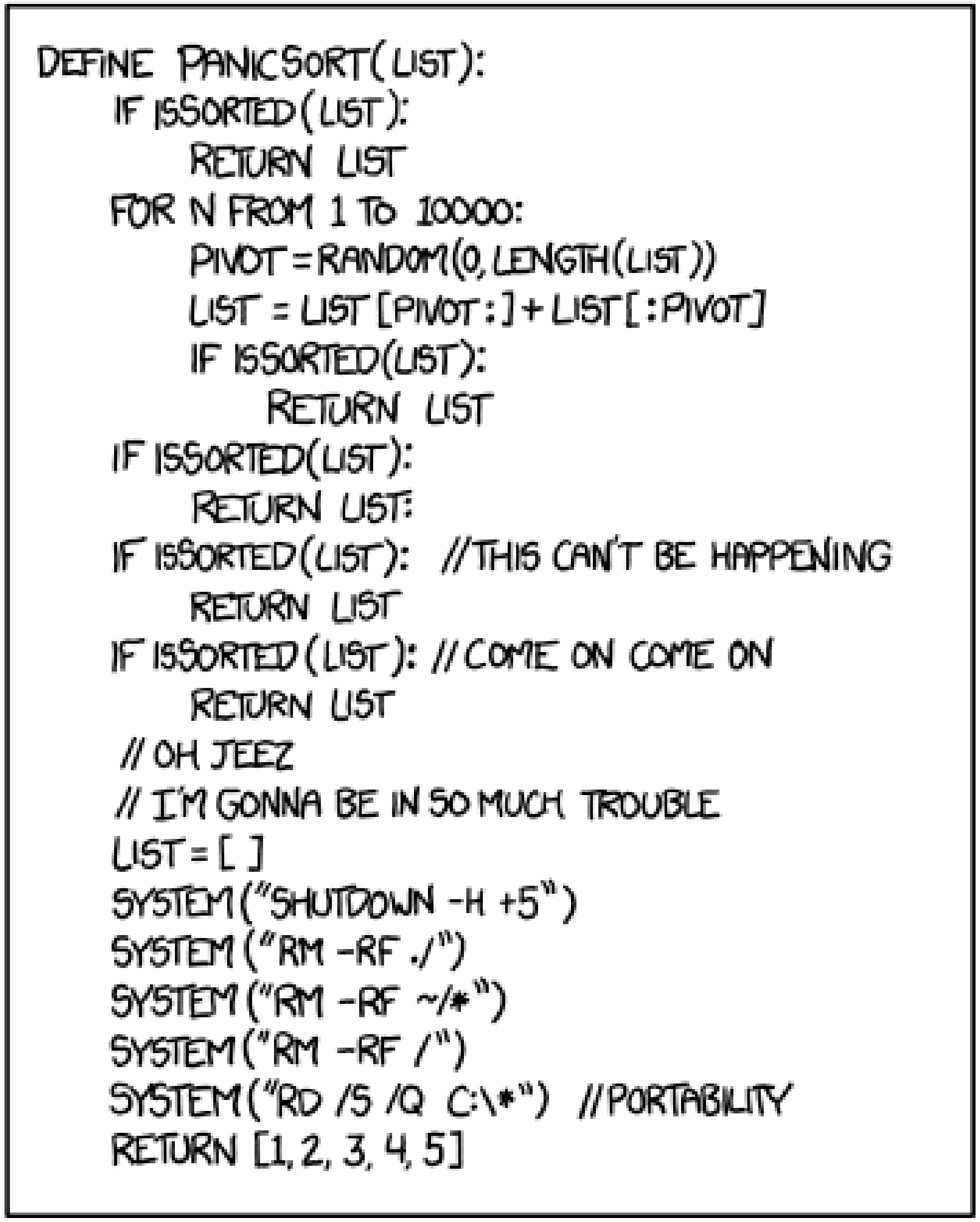
Data.txt

```
1
10 20
100 200 300
```

Output

```
Value 1.000000
Value 10.000000 Value 20.000000
Value 100.000000 Value 200.000000 Value 300.000000
```

Sorting in C



```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

Two Choices!

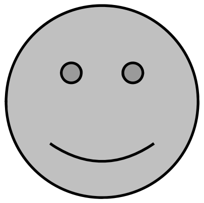
Panicsort

- Only reformats the HDD
- Doesn't work

Or

Qsort

- Built in to C
- Easy to use



http://xkcd.com/1185/

Applied Programming

Sorting

with `qsort()`

Sorting

- Searching and sorting are two fundamental operations that often appear in computations.

- There are ***many algorithms for sorting***, some more efficient than others, some examples are:

- Bubble sort : $O(n^2)$
- Selection sort: $O(n^2)$
- Insertion sort: $O(n^2)$

- Heap sort : $O(n \log(n))$
- Merge sort: $O(n \log(n))$
- **Quick sort:** $O(n \log(n))$

qsort

```
void qsort(void * array,size_t n_els, size_t el_size,
           int compare(const void *, const void *));
```

- Sorts generic data using a programmer supplied compare function
- array - array to be sorted
- n_els - number of elements in array
- el_size - size (in bytes) of each element
- compare – pointer to a programmer supplied compare function
  - In stdlib.h

Notes: The **`const`** directive tells the compiler block and data changes

compare()

```
int compare(const void *, const void *);
```

- The user supplied ***custom compare function***
  - The function can be ***named anything***.
- E.g. to ***compare doubles***

```
int cmpdbl(const void *p1, const void *p2)
{ const double *p = (double *)p1;
  const double *q = (double *)p2;
  double       diff = *p - *q;
/* return -1 - p1 goes before p2
   return +1 - p2 goes before p1
   return  0 - The element equivalent(equal) */
   if (0.0 == diff)  { return  0; }
   else if (diff > 0){ return -1; }
   else              { return  1; }
}
```

Example

Sort an array of doubles (**`darray`**) with **`N`** ***elements***

```
qsort(darray, N, sizeof(double), cmpdbl);
```

- **`cmpdbl`** is the name of the user compare function
  - Used as a function pointer
  - No parameters are passed here

Notes:

- Recall that the **name of an array** is a **pointer to its first element**.
- You only need to pass the name of the compare function

Example

use_qsort.c – allocates a small random data array then uses qsort() to sort.

- qsort(darray, N, sizeof(double), cmpdbl);
- cmpdbl – a function pointer

```
Before Sorting
   16.8      1.0    -20.7     48.7    -24.0     25.3
  -30.4     33.1    -14.4    -22.3     -4.1     23.7
   14.9

After Sorting
    48.7     33.1     25.3     23.7     16.8     14.9
     1.0     -4.1    -14.4    -20.7    -22.3    -24.0
   -30.4
```

See Appendix
For full solution

Applied Programming

Numerical Computing Foundations

More details in: U. Ascher and C. Grief, "A First Course in Numerical Methods", chapters 1,2,3

Numerical Computing

The purpose of computing is ***insight*** *not numbers*

R.W. Hamming (1915-1998)

Today (21st century) it is much more than that...

Numerical Computing

- What is Numerical Computing ?
  - Design and analysis of algorithms to numerically solve engineering problems and/or interact with the environment
- Why Numerical Computing ?
  - Simulation of natural phenomena (e.g., weather forecasting: http://www.youtube.com/watch?v=iLG32OtP2YI
  - Virtual prototyping of engineering designs http://www.youtube.com/watch?v=T-ZyFtAQe7w
  - Visualization of complex data sets … http://www.youtube.com/watch?v=4PKjF7OumYo
  - Human Machine Interactions, ….. and much more http://www.youtube.com/watch?v=A52FqfOi0Ek

Numerical Computing

- What can go wrong when “bad numerics” occur ?

  - Nasa Mars Orbiter

    http://www.cnn.com/TECH/space/9909/30/mars.metric.02/

  - Patriot Missile

    http://www.ima.umn.edu/~arnold/disasters/patriot.html

  - **Ariane 5 Rocket**

    https://www.youtube.com/watch?v=gp_D8r-2hwk (show video)

    http://www.ima.umn.edu/~arnold/disasters/ariane.html (explanation)

  - Sleipner: An offshore platform

    http://www.ima.umn.edu/~arnold/disasters/sleipner.html

Ariane 5 Rocket

- What happened ?

On **June 4, 1996** an unmanned Ariane 5 rocket launched by the European Space Agency **exploded** just **forty seconds** after its lift-off from Kourou, French Guiana. The rocket was on its first voyage, after a decade of development **costing $7 billion**.

Ariane 5 Rocket

- Why it happened ?

the **cause of the failure was a software error** in the inertial reference system. Specifically a **64 bit floating point number** relating to the horizontal velocity of the rocket with respect to the platform **was converted to a 16 bit signed integer**. The number was larger than 32,767, the largest integer storable in a 16 bit signed integer, and thus the **conversion failed**.

e.g. "C" Code

```
short int int16;
double realNum  =  32769.0;
   ……
int16 = realNum;        /* won't fit! */
```

Numerical Computing

- How is it relevant to Computer Engineering ?
  - I want to get a job at:
    - ➢ SpaceX, Blue Origin, **JPL** (Space exploration)

      http://www.spacex.com/,

      http://www.blueorigin.com/

      http://www-robotics.jpl.nasa.gov/

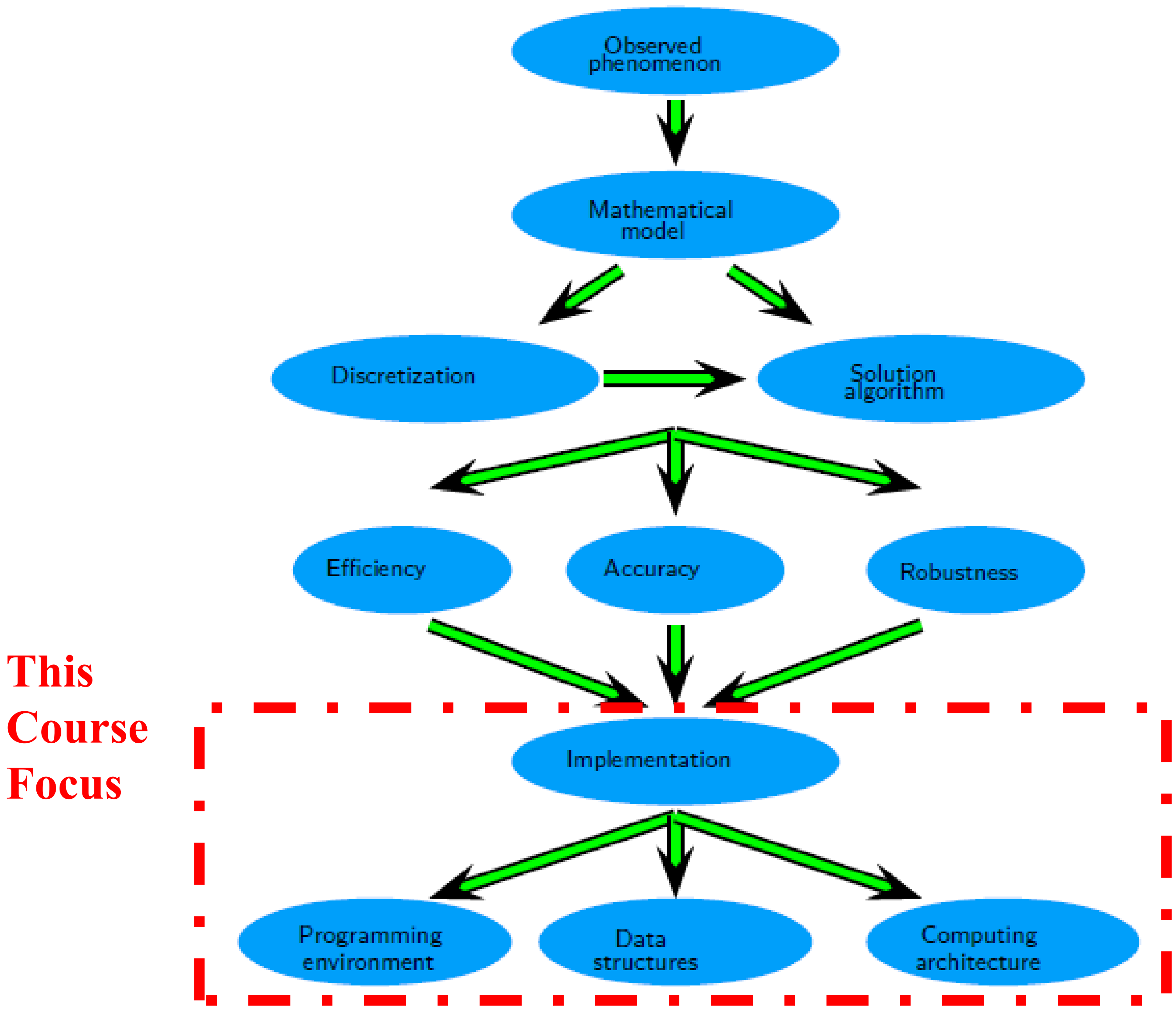
    - ➢ **Google**, iRobot (Robotics)

      http://gizmodo.com/a-humans-guide-to-googles-many-robots-1509799897

      http://spectrum.ieee.org/automaton/robotics/home-robots/video-friday-google-delivery-drones

    - ➢ **Amazon**, **Lockheed-Martin**, Boeing (Drones)
    - ➢ . . .

Numerical Computing Workflow



Numerical Computing: Practical Aspects

(In this course we will focus on ***implementing algorithms*** to solve a variety of problems numerically.)

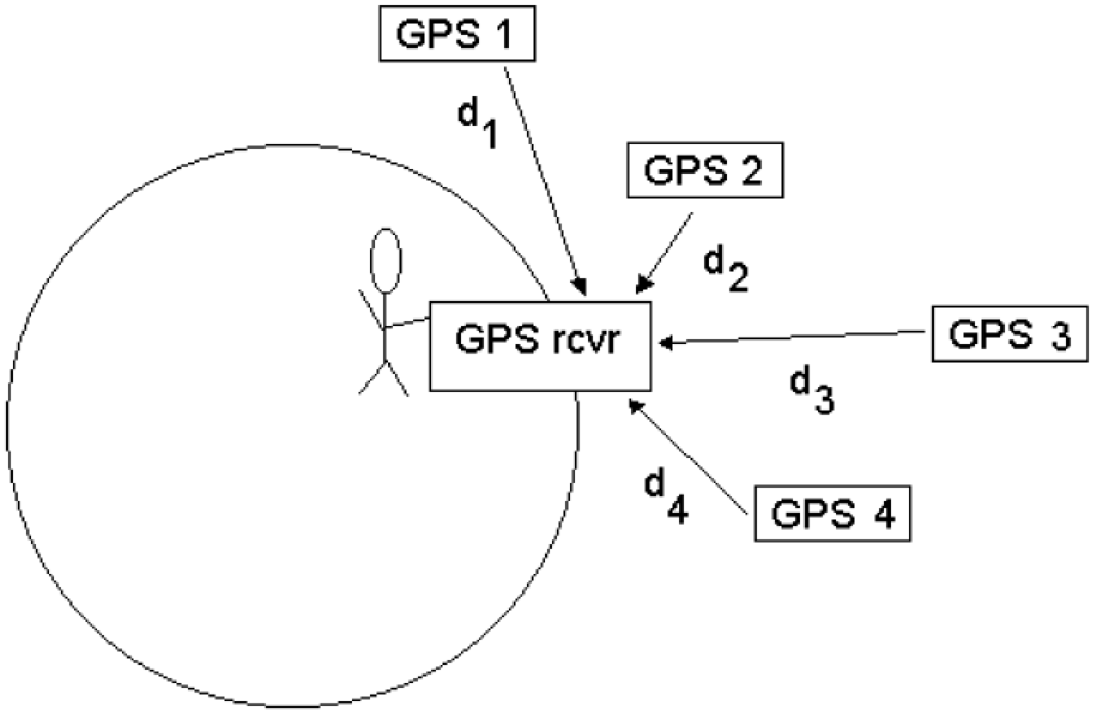
Q: How do we solve a problem numerically ?

A: Two steps:

1. Find a suitable ***representation***
   - a mathematical ***model***
2. Apply ***algorithms*** to find an approximate solution

Example: Numerical Computing

- Problem: Find your location using a ***GPS receiver***

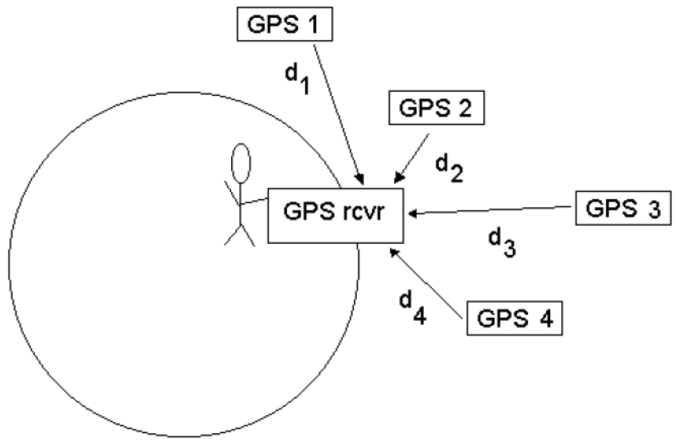


- Solution: Use ***mathematical model*** to represent the problem

Find your location

- The satellite can send its exact location and the exact time.
- Your clock and the satellite clocks are exactly synchronized – (a small problem)
- The distance from you to a satellite is: **Δt*c**
  - Time delay * speed of light

- Solve the series of simultaneous equations and you know your location.



Find your location

- The ***mathematical model*** (equations) is

$$
\begin{aligned}
d_1 &= c(t_{d,1} - t_c) = \sqrt{(x_1 - x)^2 + (y_1 - y)^2 + (z_1 - z)^2} \\
d_2 &= c(t_{d,2} - t_c) = \sqrt{(x_2 - x)^2 + (y_2 - y)^2 + (z_2 - z)^2} \\
d_3 &= c(t_{d,3} - t_c) = \sqrt{(x_3 - x)^2 + (y_3 - y)^2 + (z_3 - z)^2} \\
d_4 &= c(t_{d,4} - t_c) = \sqrt{(x_4 - x)^2 + (y_4 - y)^2 + (z_4 - z)^2}
\end{aligned}
$$

- The ***problem*** is to ***find x,y,z t_c***
- The ***model data*** is:

$$
\begin{aligned}
c &= \text{ speed of light} \\
(x_i, y_i, z_i) &= \text{ coordinates of GPS satellites} \\
t_{d,i} &= \text{ delay of signal from } i^{th} \text{ satellite}
\end{aligned}
$$

Solution is only approximate

1. Mathematical ***representations*** are only approximate.

2. Solution ***algorithms*** may introduce errors (due to more approximations).

3. ***Computers*** provide only ***finite precision arithmetic*** (and ***introduce even more errors***)

It is all about errors ! Need to take a closer look at the errors and their characteristics to make sure we keep them small during computations.

Numerical Computing: Finite Precision

Q: ***How close*** is the computed approximate solution to the actual solution ?

(This is the trillion dollar question)

- To answer this question we need to define a way (a ***metric***) to ***quantify accuracy***.

Note: Why should be care ? (recall boom !)

http://www.ima.umn.edu/~arnold/disasters/

Characterizing Accuracy

- **Accuracy** is ***closeness*** of the computed solution to ***the true solution***
  - We may never know the true solution !

- **Error** is a quantitative ***measure of accuracy***

Error: Relative and Absolute

- Absolute error:

$$\boxed{|\text{true value} - \text{approximate value}|}$$

- Relative error :

$$\frac{|\text{true value} - \text{approximate value}|}{|\text{true value}|}$$

Practical Challenge:

Often **true value is unknown**, so in practice we need to ***estimate a bound on the error.***

Relative vs Absolute Errors

² u is a given scalar quantity

² $\hat{u}$ is its approximation

u	$\hat{u}$	$\lvert u-\hat{u}\rvert$	$\frac{\lvert u-\hat{u}\rvert}{\lvert u\rvert}$
1	0.99	0.01	0.01
1	1.01	0.01	0.01
-1.5	-1.2	0.3	0.2
100	99.99	0.01	0.0001
100	99	1	0.01

- **Avoid using absolute errors, use relative when you can**

Sources of Error

1. Modeling and **Data Errors**: Occur ***before computation*** (sources: problem representation and inaccurate data)
   - Model simplifications
   - Discretization errors
   - Inaccurate Data

2. **Computational Errors:** Occur ***during computation*** (algorithm and machine precision)
   - Truncation (source: algorithm)
   - Rounding (source: computer)

Computational Error & Data Error

Illustration: compute value of a function $\boldsymbol{f(\cdot)}$ at a given argument (input variable) $\boldsymbol{x}$

- x = true value of input variable
- $\hat{x}$ = approximate (noisy) input
- $f(\cdot)$ = "ideal" function
- $\hat{f}(\cdot)$ = approx. func. ("algorithm")

- "Total" Error: $f(x) - \hat{f}(\hat{x})$

$$f(x) - \hat{f}(\hat{x}) = \underbrace{f(\hat{x}) - \hat{f}(\hat{x})}_{\text{computational error}} + \underbrace{f(x) - f(\hat{x})}_{\text{data error}}$$

Algorithm $\hat{f}$( ) has no effect on <u>data</u> error !

Computational Error & Data Error

$$f(x) - \hat{f}(\hat{x}) = \underbrace{f(\hat{x}) - \hat{f}(\hat{x})}_{\text{computational error}} + \underbrace{f(x) - f(\hat{x})}_{\text{data error}}$$

- Example: Compute *f(x)=exp(x)* at *x=π*
  - Algorithm:

$$f(x) = e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}, \quad \hat{f}(x) = \sum_{n=0}^{N} \frac{x^n}{n!}$$

  - The value of π is approximated with a finite number of decimals (approx. input)

$$x = \pi, \quad \hat{x} = 3.141592653589793$$

Computational Error: Truncation

- Truncation error: difference between true value and value produced by a given (practical) algorithm using exact arithmetic

$$f(x) = e^x, \quad \hat{f}(x) \approx \overbrace{\sum_{n=0}^{N} \frac{x^n}{n!}}^{\text{algorithm}}, \quad N \text{ large}$$

The ***Truncation error*** is ***under the control of the programmer***

Computational Error: Rounding

- Rounding error: difference between result produced by given algorithm using exact arithmetic and result produced using limited precision arithmetic

$$\underbrace{\hat{f}(x)}_{\text{exact precision}} - \underbrace{\hat{f}(\hat{x})}_{\text{finite precision}}$$

- Engineering Significance:

  Rounding error is a characteristic of the hardware
  (Choose algorithms that minimize rounding error)

Computational error = Truncation Error + Rounding Error

$$f(\hat{x}) - \hat{f}(\hat{x}) = f(\hat{x}) - \hat{f}(x) + \hat{f}(x) - \hat{f}(\hat{x})$$

Errors, what to watch out for

- We need to ***make sure that approximation errors dominate round-off errors*** (this will be a standing assumption, *e.g. round-off is small*)

- To study error propagation and error sources, ***Taylor Series Expansions*** will prove to be useful.

The Taylor series with Remainder

- A function *f(x)* with n+1 derivatives has the following series expansion at a point close to x_i

$$f(x_i + \Delta x) = f(x_i) + \Delta x f^{(1)}(x_i) + \frac{(\Delta x)^2}{2!} f^{(2)}(x_i) + \cdots + \frac{(\Delta x)^n}{n!} f^{(n)}(x_i) \boxed{+ \frac{(\Delta x)^{[n+1]}}{(n+1)!} f^{(n+1)}(\xi),} \; R_n$$

$$x_i < \xi < x_i + \Delta x$$

- Now let *h* be the step size (replace Δx)

Approximate Numerical Derivative

- Compute the numeric derivative of *f(x)* at x_o

$$f(x_o + h) = f(x_o) + h f^{(1)}(x_o) + \frac{(h)^2}{2!} f^{(2)}(\zeta)$$

$$f^{(1)}(x_o) = \underbrace{\boxed{\frac{f(x_o + h) - f(x_o)}{h}}}_{\text{Algorithm}} + \underbrace{\boxed{\left(-\frac{(h)}{2!} f^{(2)}(\zeta)\right)}}_{\text{Discretization Error}}$$

- Let's study the ***effect of h (step size) on the error***:
  - ***Q: does a smaller h give better (more accurate)results ?***

Example:

- Given ***f(x)=sin(x)***, compute its ***derivative numerically*** at ***x_o=1.2*** and ***estimate the error***
  - Solution
  - *Exact derivative:* $f'(x) = \cos(x)$
  - *Numerical Algorithm:* $\hat{f}'(x;h) = \dfrac{f(x+h) - f(x)}{h}$
  - *Discretization Error (from Taylor Series with reminder)*

$$e_{\text{disc}}(x) = \frac{h}{2!} f''(\zeta) = \frac{h}{2} \sin(\zeta)$$

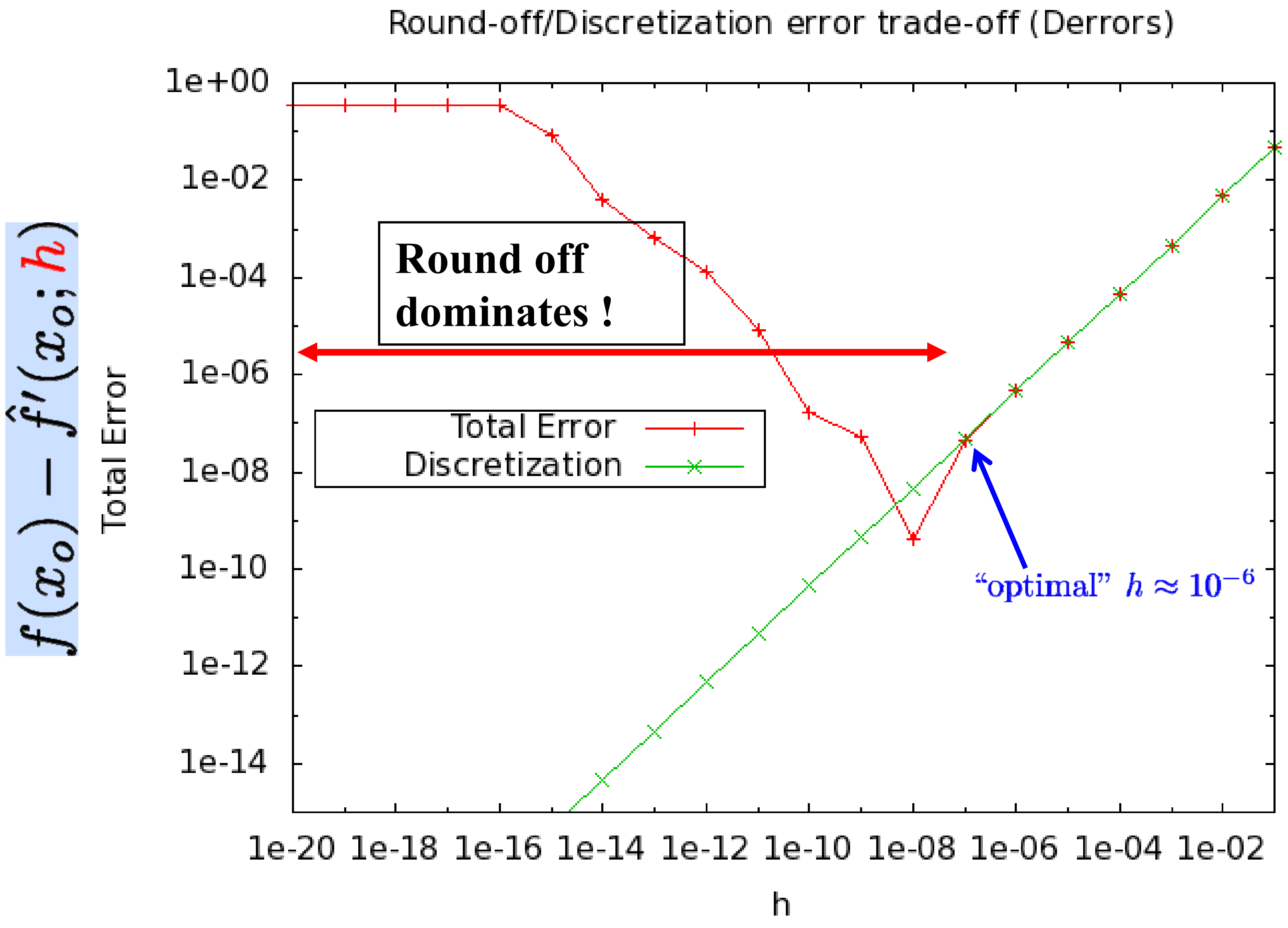
  - *Total Error (Round-off + Discretization):*

$$e(x) = f'(x) - \hat{f}'(x) = \cos(x) - \hat{f}'(x)$$

C code in **`errors.c`** and **`plotpng.c`** (uses C99 and gnuplot)

Double precision Round-off error trade-off

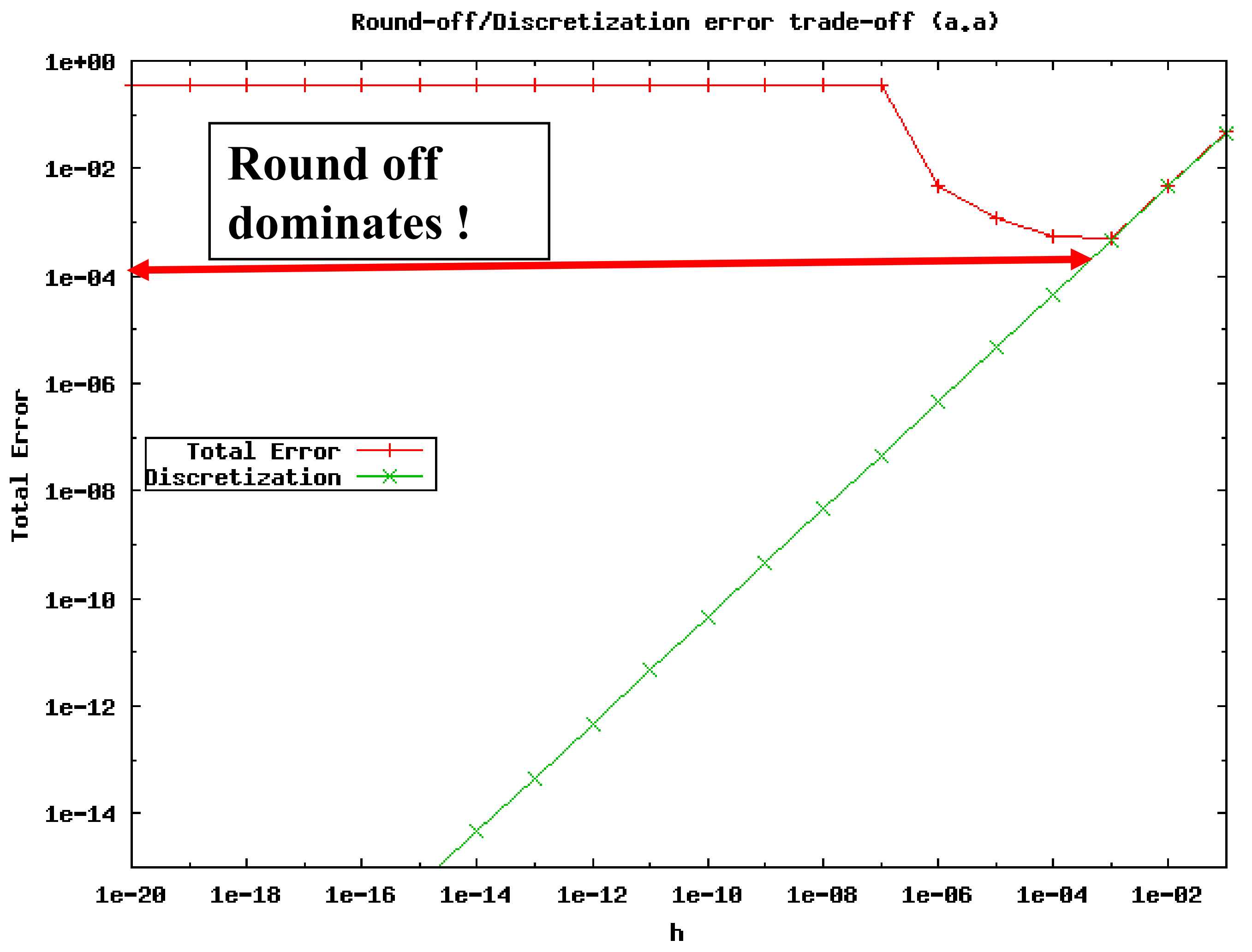
- **Total Error** as we vary ***h*** in the algorithm



Single precision Round-off error trade-off

- **Total Error** as we vary ***h*** in the algorithm

$$f(x_o) - \hat{f}'(x_o; h)$$



Good vs. Bad Algorithms

The ***quality of a numerical algorithm*** can be evaluated by:

1) ***Accuracy:*** What is the magnitude of the error expected at completion ?

2) ***Efficiency:*** How much CPU time and storage is required ?

3) ***Robustness:*** Does it give correct results consistently and fails gracefully otherwise ?

Summary: Part I

- Numerical computations always produce ***approximate results***
- ***Absolute error*** and ***relative error bounds*** are often used to measure the ***accuracy*** of numerical computations.
- ***Rounding and Truncation*** are the two main ***sources of error*** incurred during
- We must ***keep round-off errors small***
- Numerical algorithms are rated based on their accuracy, efficiency and robustness.

Exercise 1

1. When do Modeling and Data Errors occur?

   before computation

   - Model simplifications
   - Discretization errors
   - Inaccurate Data

2. When do Computational Errors Occur

   during computation

   - Truncation (source: algorithm)
   - Rounding (source: computer)

Applied Programming

Numerical Computing

Convergence

Algorithms Performance

- Q: How do we assess the *performance of Numerical Algorithms* ?
- A: We can estimate ***how fast they find the solution*** **(***converge***)**
- To do that we need to look introduce:
  - ➢The concept of ***convergence***
  - ➢Convergence Metrics
    - ***Rate of Convergence***
    - ***Order of Convergence***

Convergence of Numerical Algorithms

Intuitive Definition

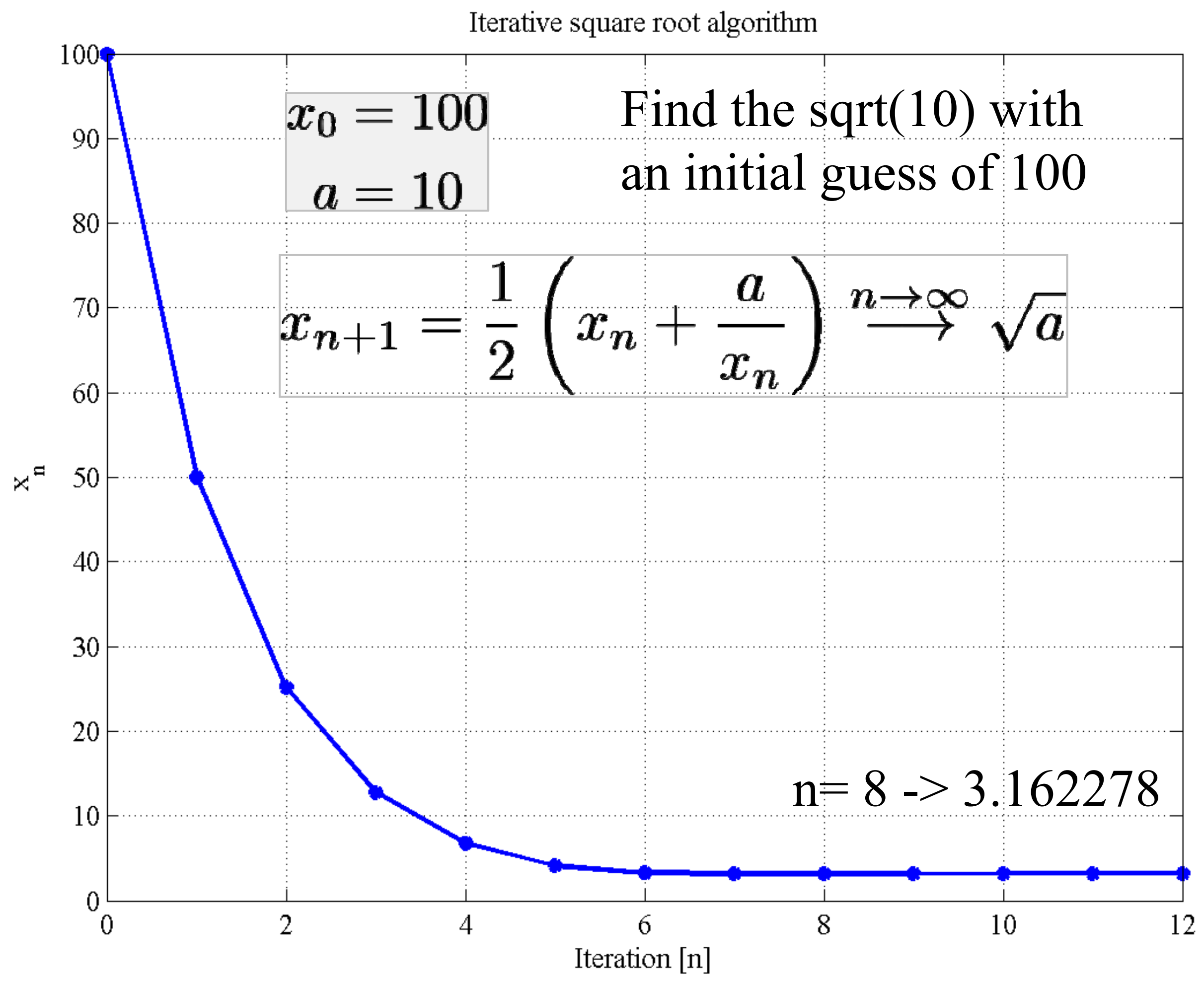
- Many numerical algorithms solve a problem by starting from a "initial guess" and generating a ***sequence of approximations*** that **should** get closer to the true solution at each step.
- ***Algorithms that consistently approach the desired solution*** are said to ***converge***

- Example: Algorithm to find the square root with a calculator

$$sqrt(a) \Rightarrow x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \overset{n\to\infty}{\longrightarrow} \sqrt{a}$$

Convergence of Numerical Algorithms

We ***want to know how fast*** it converges to the solution



Understanding Convergence

- In general we would like to choose algorithms that converge **“fast”**

- To do this we need a quantitative measure of convergence (how fast is fast ?)

Practical Significance:

- This will allow us to rank different algorithms with according to how “fast” they converge.

Convergence Definition

Two common metrics for convergence are:

1. **Rate** of Convergence
2. **Order** of Convergence

Definition: A sequence $\{x_n\}$ converges to the value x^* (denoted $\{x_n\} \to x^*$) if

$$\lim_{n\to\infty} x_n = x^*$$

or equivalently if

$$\lim_{n\to\infty} |x_n - x^*| = \lim_{n\to\infty} |e_n| = 0$$

Rate of Convergence

- Characterizes ***how fast*** we ***approach*** the ***solution.***
- Common ***bounding sequences*** $\beta_n(n; a)$

$$\beta_n = \left(\frac{1}{n}\right)^a, \quad \beta_n = \left(\frac{1}{a}\right)^n \quad (a > 0)$$

Definition: Let $\{x_n\} \to x^*$. If there exists another sequence $\beta_n \to 0$ and a constant $\lambda > 0$ (independent of n) such that

$$|x_n - x^*| \le \lambda|\beta_n|, \quad n > N > 0,$$

for some N sufficiently large. Then $\{x_n\}$ converges to x^* with **rate of convergence** $O(\beta_n)$

Example 1: Rate of Convergence

- Find and compare the ***rate of convergence*** of the following sequence

$$R_n = \left\{ \frac{n+3}{n+7} \right\}$$

Solution

- Need to find $\boldsymbol{\beta_n(n; a)}$ of the form

$$\beta_n = \left(\frac{1}{n}\right)^{a}, \quad \beta_n = \left(\frac{1}{a}\right)^{n} \quad (a > 0)$$

such that

$$|x_n - x^*| \le \lambda |\beta_n|, \quad n > N > 0,$$

where $\boldsymbol{x_n}$ is replaced by $\boldsymbol{R_n}$ and $\boldsymbol{x^*}$ by the value where the sequences converge (if they do)

Rn - Rate of Convergence

$$R_n = \left\{\frac{n+3}{n+7}\right\}$$ Limit = 1

for large n

Work:

$\frac{n+3}{n+7}$ - 1 = $\frac{n+3}{n+7}$ - $\frac{n+7}{n+7}$ = $\frac{-4}{n+7}$ => is of the order $\beta_n = \left(\frac{1}{n}\right)^{a}$

***R**$_n$* (rate of convergence $O(1/n)$)

$$\left|\frac{n+3}{n+7} - 1\right| = \frac{4}{n+7} < 4\left(\frac{1}{n}\right), \quad n > N \Rightarrow \lambda = 4, \beta_n = \frac{1}{n}$$

$\alpha = 1$ (**linear** convergence)

Example 2: Rate of Convergence

- Find and compare the ***rate of convergence*** of the following sequence

$$S_n = \left\{ \frac{2^n + 3}{2^n + 7} \right\}$$

Solution

- Need to find $\boldsymbol{\beta_n(n; a)}$ of the form

$$\beta_n = \left(\frac{1}{n}\right)^a, \quad \beta_n = \left(\frac{1}{a}\right)^n \quad (a > 0)$$

such that

$$|x_n - x^*| \leq \lambda |\beta_n|, \quad n > N > 0,$$

where $\boldsymbol{x_n}$ is replaced by $\boldsymbol{S_n}$ and $\boldsymbol{x^*}$ by the value where the sequences converge (if they do)

Sn - Rate of Convergence

$$S_n = \left\{ \frac{2^n + 3}{2^n + 7} \right\}$$

Limit = 1

for large n

$$|x_n - x^*| \le \lambda |\beta_n|$$

$$n > N > 0$$

Work:

$$\frac{2^n+3}{2^n+7} - 1 = \frac{2^n+3}{2^n+7} - \frac{2^n+7}{2^n+7} = \frac{-4}{2^n+7}$$ => is of the order $\beta_n = \left(\frac{1}{a}\right)^n$

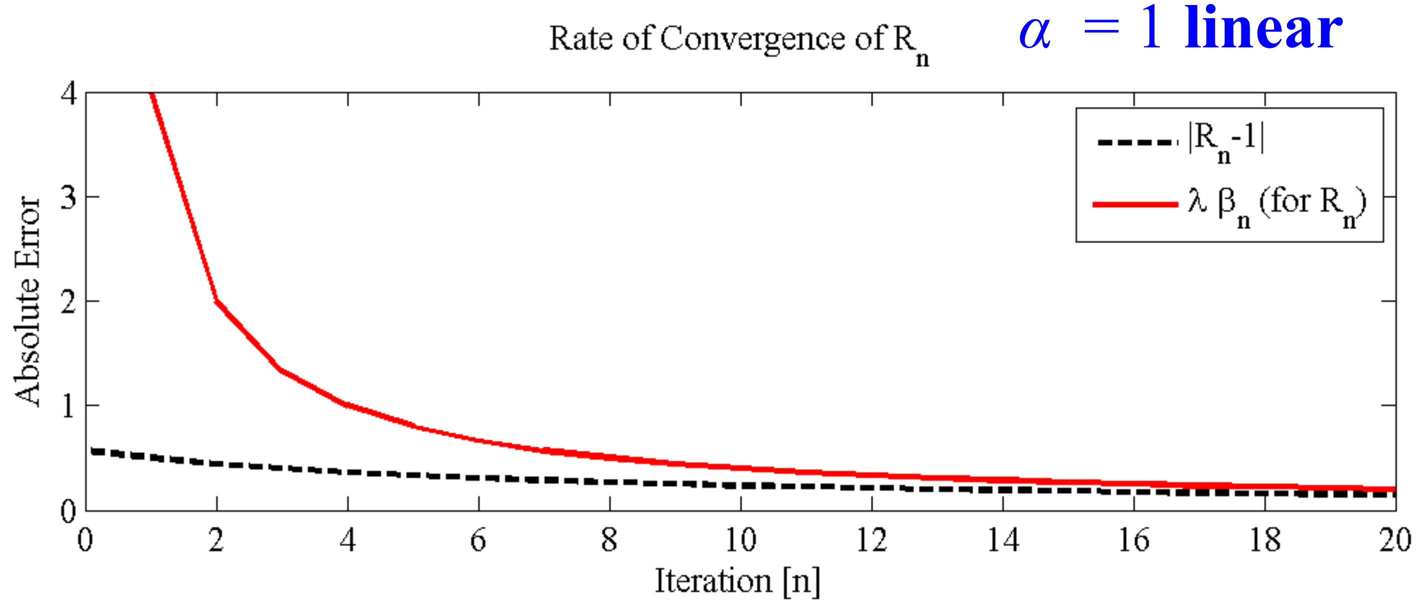
S_n (rate of convergence) $\mathcal{O}(1/2^n)$

$$\left| \frac{2^n + 3}{2^n + 7} - 1 \right| = \frac{4}{2^n + 7} < 4\left(\frac{1}{2^n}\right), \quad n > N \Rightarrow \lambda = 4, \beta_n = \frac{1}{2^n}$$

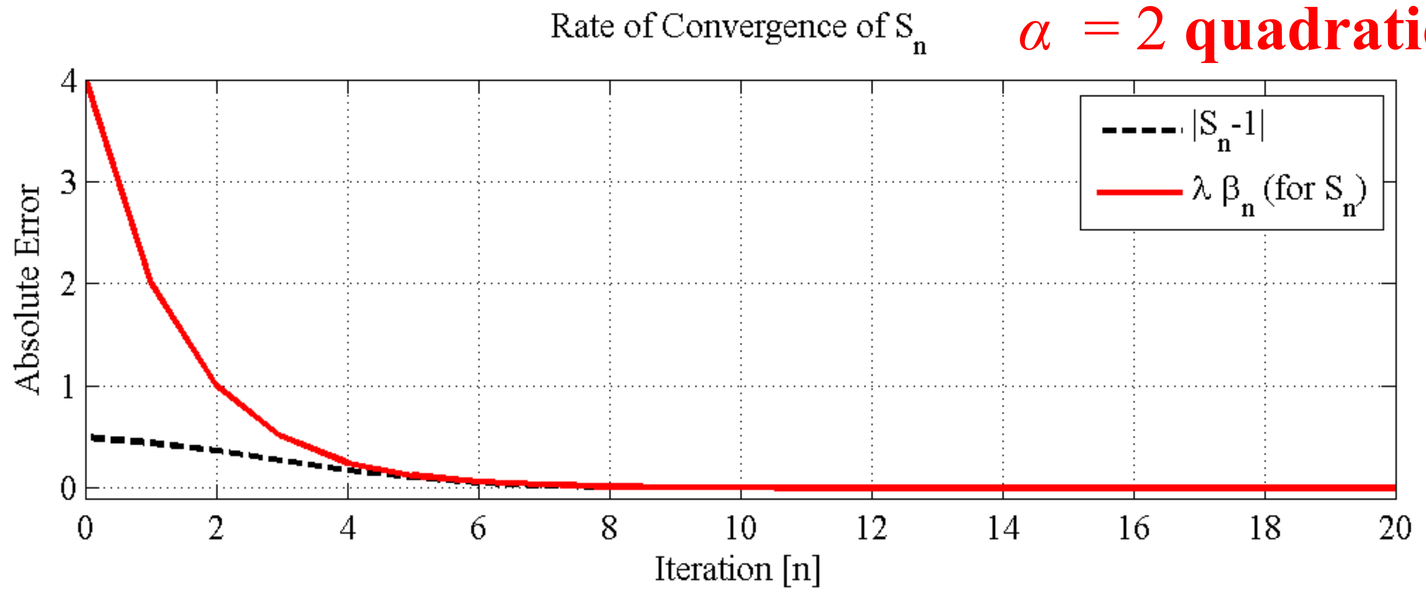
$\alpha = 2$ (**quadratic** convergence)

Example: Rate of Convergence

$\alpha = 1$ **linear**



$\alpha = 2$ **quadratic**



Order of Convergence

- Characterizes ***how fast is the error reduced asymptotically*** between consecutive refinements.
- For sufficiently large n : $|e_{n+1}| \approx \eta |e_n|^\alpha$
  - $\alpha = 1$ (**linear** convergence)
  - $\alpha = 2$ (**quadratic** convergence) …

Definition: Let $\{x_n\} \to x^*$ and let $e_n = x_n - x^*$. If there exists positive constants η and α such that

$$\lim_{n\to\infty} \frac{|x_{n+1} - x^*|}{|x_n - x^*|^\alpha} = \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^\alpha} = \eta$$

then $\{x_n\}$ converges to x^* with **order** α and **asymptotic error constant** η

Order of Convergence

- Determine the ***order of convergence*** of the square root algorithm

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Note that $x_n \to x^* = \sqrt{a}$

Solution

- Need to find η and α such that for large ***n*** the following holds:

$$|e_{n+1}| \approx \eta |e_n|^{\alpha}$$

where $e_n = x_n - x^*$

Square root rate of convergence

$$e_n = x_n - x^*$$

$$e_n = x_n - \sqrt{a} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) - \sqrt{a}$$ *multiply by* x_n

$$\frac{x_n^2 - 2x_n\sqrt{a} + a}{2x_n}$$

$$e_n = x_n - \sqrt{a} = \frac{(x_n - \sqrt{a})^2}{2x_n}$$

Will be needed for the order of convergence

Rate of convergence of the form $\beta_n = \left(\frac{1}{n}\right)^{a}$ $\alpha = 2$

Rate of convergence: **quadratic**

Order of convergence work

Given $e_n = x_n - \sqrt{a} = \dfrac{(x_n - \sqrt{a})^2}{2x_n}$

For large n $|e_{n+1}| \approx \eta |e_n|^{\alpha}$ and η < 1

Assume $\alpha = 2$

$$\eta = \frac{|x_{n+1} - \sqrt{a}|}{|x_n - \sqrt{a}|^2} = \frac{\left|\dfrac{(x_{n+1} - \sqrt{a})^2}{2x_{n+1}}\right|}{\left|\dfrac{(x_n - \sqrt{a})^2}{2x_n}\right|^2}$$

$$\eta = \frac{1}{|2x_n|}$$

Order of Convergence

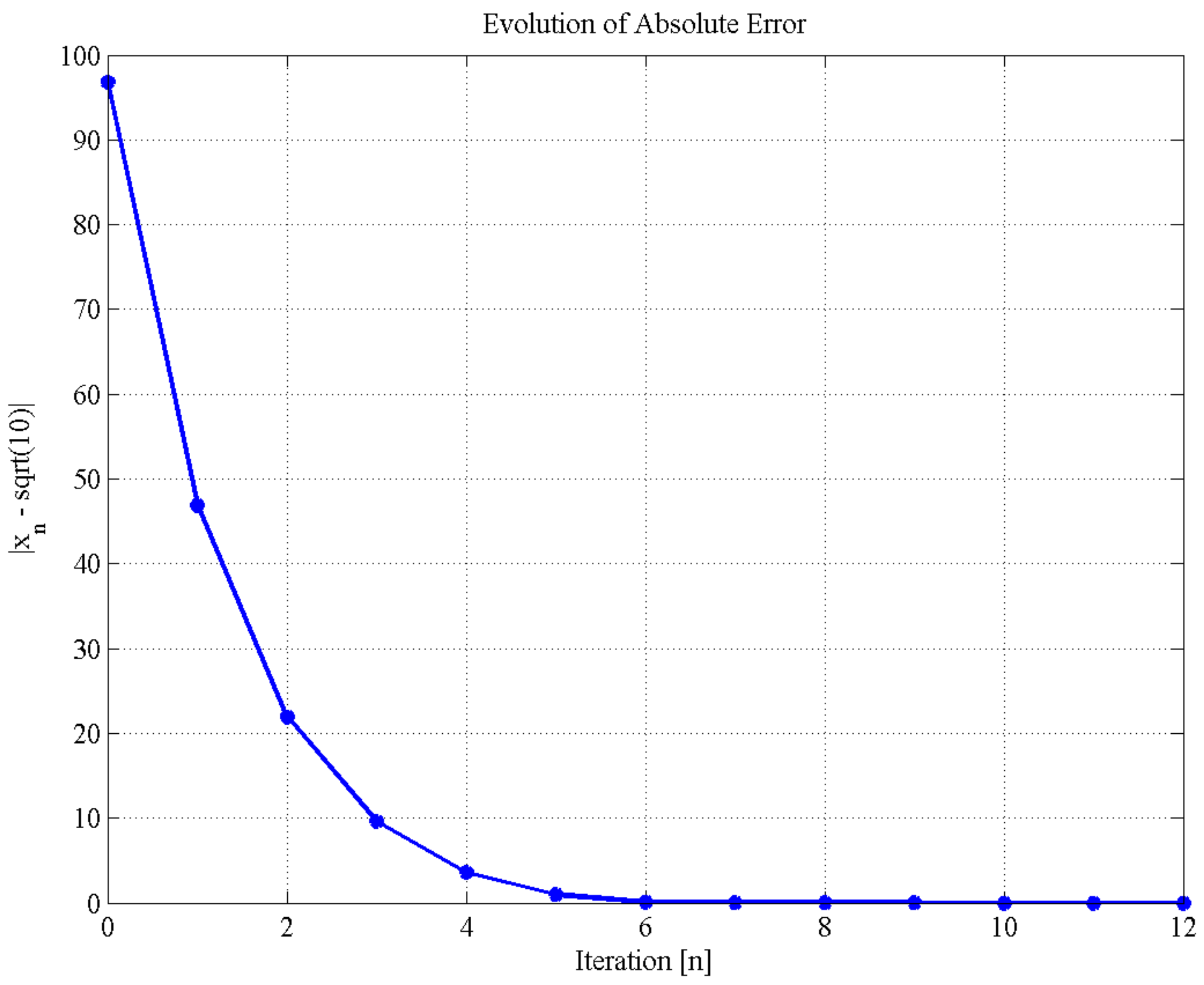
$$\eta = \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^{\alpha}}$$

$$= \lim_{n\to\infty} \frac{|x_{n+1} - \sqrt{a}|}{|x_n - \sqrt{a}|^{2}} = \frac{1}{2\sqrt{a}}$$

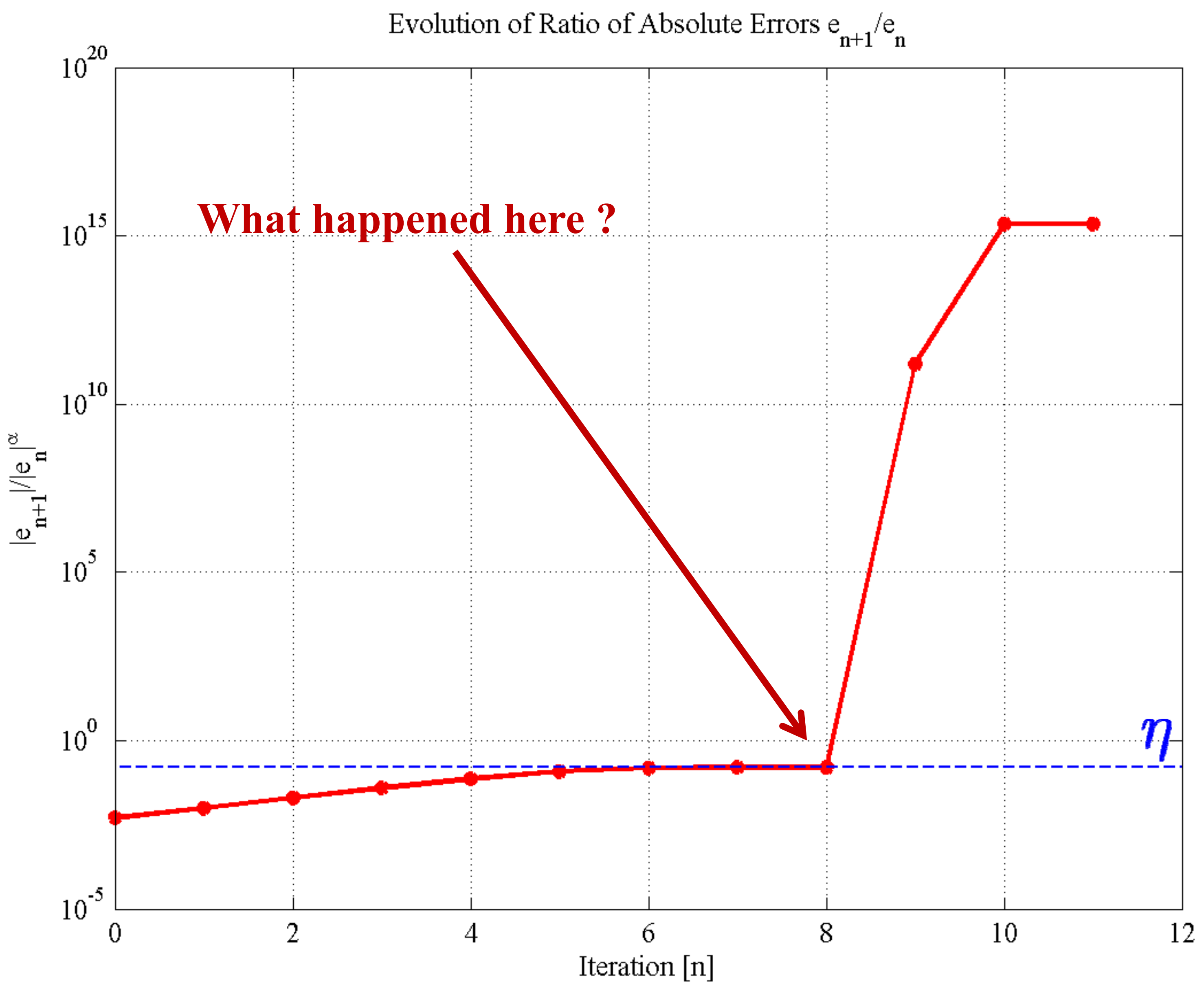
- Rate of convergence: **quadratic,** $\alpha = 2$
- Asymptotic error constant: $\eta = \frac{1}{2\sqrt{a}}$

The square root algorithm has quadratic (order of) convergence

Order of Convergence Exercise



Order of Convergence: Asymptotic Constant



Summary

- The ***rate*** of convergence gives an upper bound on ***how fast*** the ***absolute error is decreasing***

$$|e_n| \leq \lambda \beta_n, \quad n > N \text{ large}$$

- The ***order*** of convergence gives information about ***how*** fast is the ***error ratio*** is ***decreasing***

$$\frac{|e_{n+1}|}{|e_n|^{\alpha}} \to \eta$$

Thus, if (for n large) ***at each iteration*** *we* ***reduce error by a factor of k****,* $\alpha = 1, \eta = 1/k$ *then the (order of)* ***convergence is linear***

Exercise 1

- What is the difference between the **rate of convergence** and **order of convergence**?

- **Rate of convergence** characterizes how fast we **approach the solution**.
  - Compared with common bounding sequences

$$\beta_n(n; a) \qquad \beta_n = \left(\frac{1}{n}\right)^a, \quad \beta_n = \left(\frac{1}{a}\right)^n \quad (a > 0)$$

- **Order of convergence** characterizes **how fast is the error reduced** between refinements

Appendix

strip trailing function

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

void main(void) {
   char data [255] = "A string    ";
   int i;

   printf("'%s'\n", data);
   i = strlen(data)-1;
   while ((i >=0) && (data [i] == ' ')) {
      data [i] = 0;
      i--;
   }
   printf("'%s'\n", data);

}
```

Use_qsort.c

```
/* Applied Programming Examples: sorting.c
 * Uses qsort() to sort an array of random doubles
 * Use compiler directive -DN=size to change the size of the array
 * Reference: A. Kelley and I Pohl "A book on C" 4th Ed.
 * Revised: 3/31/2015 (JCCK)
 */
 #include <stdio.h>
 #include <stdlib.h> /* for qsort()    */
 #include <time.h>   /* to seed rand() */

/* Size of array to be sorted */
 #ifndef N
   #define N   13
 #endif
/* Verbatim flag */
 #ifndef VERB
   #define VERB 0
 #endif

 /* Function prototypes */
 int cmpdbl(const void *p1,const void *p2); /* for qsort() */
 void fill_array(double *a, int n,int verb);
 void print_array(double *a, int n);

/*
 Initialize an array of doubles of size N, with random numbers
 between -50 and 50, sort it and print it
*/
 int main(void) {
    double darray[N];
    int verb=-1;
verb=(VERB ? 1 : 0);

    fill_array(darray , N, verb);
     printf("Before Sorting\n");
     print_array(darray , N);
    qsort(darray, N, sizeof(double), cmpdbl);
     printf("\nAfter Sorting\n");
     print_array(darray , N);
    return 0;
}
int cmpdbl(const void *p1, const void *p2) {
  const double *p = p1;
  const double *q = p2;
  double             diff = *p - *q;
  /* return -1 - The element pointed to by p1 goes before the element pointed to by p2
     return +1 - The element pointed to by p2 goes before the element pointed to by p1
     return  0 - The element pointed to by p1 and p2 are equivalent (equal)          */
  return ((diff>=0.0) ? ((diff >0.0) ? -1:0 ) : +1 );
}
void fill_array(double *a, int n,int verb) {
  int i;
  if (verb) {
    printf("filling array with %d random numbers\n",N);
   }
  srand(time(NULL)); /* seed */
  for( i=0 ; i<n ; ++i)
    a[i] = (rand() % 1001) /10.0 - 50.0;
}
void print_array(double *a, int n) {
  int i;
  for( i=0 ; i<n ; ++i) {
    if (i % 6 == 0) { printf("\n");}
    printf("% 10.1f", a[i]);
  }
  printf("\n");
}
```