Mathematical Software Programming (02635)

Lecture 3 — September 20, 2018

Instructor: Martin S. Andersen

Fall 2018

DTU

Practical information

Assignment 1

- due on Wednesday Oct. 24
- 10% of final grade

Assignment 2

- will be posted no later than Oct. 26
- due on Wednesday Nov. 21
- 10% of final grade

Checklist — what you should know by now

- How to write a simple program in C (`int main(void) {}`)
- Basic data types (`int`, `long`, `float`, `double`, ...)
- Basic input/output (`printf`, `scanf`)
- Implicit/explicit typecasting
- How to compile and run a program from terminal / command prompt
- Control structures and loops
- Limitations of integer and floating-point arithmetic

This week

Topics

- Arrays
- Pointers
- Multidimensional arrays
- Memory
- Error analysis and conditioning

Learning objectives

- Evaluate discrete and continuous mathematical expressions
- Describe and use data structures such as **arrays**, linked lists, stacks, and queues
- Choose appropriate data types and data structures for a given problem

Automatic array allocation/deallocation

Compile-time array allocation

```
double data[5] = {-1.0,2.0,4.0,1e3,0.1};
```

Run-time array allocation

```
size_t n = 0;
scanf("%zu",&n);  // Windows: format specifier %Iu
double data[n];
```

- also known as *variable-length arrays* (VLA)
- defined in C99, but optional in C11
- we will talk about variable scope and memory allocation next week
- Windows/MSYS2: add `CPPFLAGS=-D__USE_MINGW_ANSI_STDIO=1` to Makefile to use standard format specifiers

Pointers

```
int val = 1;          // val has type int
int * pval;           // pval has type (int *)

pval = &val;          // store address of val in pval (pval points to val)
*pval = 2;            // set val = 2 (pval is unchanged)
```

- a pointer stores an address in memory (it “points” to something)
- **dereferencing** operator: `*`
  - declaring a pointer: `<type> * <name>`
  - `*pval` dereferences a pointer `pval` (content of memory pointed to by `pval`)
- **address of** operator: `&`
  - `&val` yields address of variable `val`
  - `&val` is the location in memory where `val` is stored
- use format specifier %p to print pointer/address using `printf`

Example: pointers and arrays

```
/* Declare double array and double pointer */
double data[4] = {1.0}; // double array of length 4
double * pdata;         // pointer to double

/* Initialize pdata with address of 2nd element of array */
pdata = &data[1];       // same as pdata = data+1;

/* Update values of array via pointer */
pdata[0] = 2.0;         // sets data[1] = 2.0
pdata++;                // increments pointer
*pdata = 3.0;           // sets data[2] = 3.0
*(++pdata) = 4.0;       // sets data[3] = 4.0
*(pdata-3) = 0.5;       // sets data[0] = 0.5
```

Why use pointers? Is this code easy to read/understand?

Multidimensional arrays

A two-dimensional example

```
double mat[3][4];    // uninitialized array of size 3-by-4

// Set all elements of mat to 1.0
for (size_t i=0;i<3;i++) {
    for (size_t j=0;j<4;j++) {
        mat[i][j] = 1.0;
    }
}
```

- a two-dimensional array can be thought of as “an array of arrays”
- an array “behaves” like a pointer in many ways
- `mat[i]` is an array — corresponds to ith row of `mat`
- `&mat[i][0]` (and `mat[i]`) represents address of first element of ith row

Example 1: two-dimensional array

```c
double mat[3][4];     // uninitialized array of size 3-by-4
double *pi;

// Initialize all elements of mat
for (size_t i=0;i<3;i++) {
    pi = mat[i];                // pointer to i'th array
    for (size_t j=0;j<4;j++) {
        pi[j] = 4*i+j;          // same as mat[i][j] = 4*i+j
    }
}
```

- `pi[0]` is first element of ith array
- `pi[3]` is fourth element of ith array
- What happens if we try to access `pi[4]` or `pi[-1]`?

Example 2: row-wise storage in one-dimensional array

```
double matr[3*4];  // uninitialized array of length 12
double * pd;       // pointer to double
// Treat mat as a 3-by-4 matrix with row-wise storage
for (size_t i=0;i<3;i++) {      // loop over rows
    pd = matr+i*4;
    for (size_t j=0;j<4;j++) {    // loop over cols.
        pd[j] = i*4.0 + j;
    }
}
```

Alternatively, loop can be expressed as:

```
for (size_t i=0;i<3;i++) {      // loop over rows
    for (size_t j=0;j<4;j++) {    // loop over cols.
        matr[i*4+j] = i*4.0 + j;
    }
}
```

Example 3: column-wise storage in one-dimensional array

```
double matc[3*4];  // uninitialized array of length 12
double * pd;       // pointer to double
// Treat mat as a 3-by-4 matrix with col.-wise storage
for (size_t j=0;j<4;j++) {      // loop over cols.
    pd = matc+j*3;
    for (size_t i=0;i<3;i++) {    // loop over rows
        pd[i] = j*3.0 + i;
    }
}
```

Alternatively, loop can be expressed as:

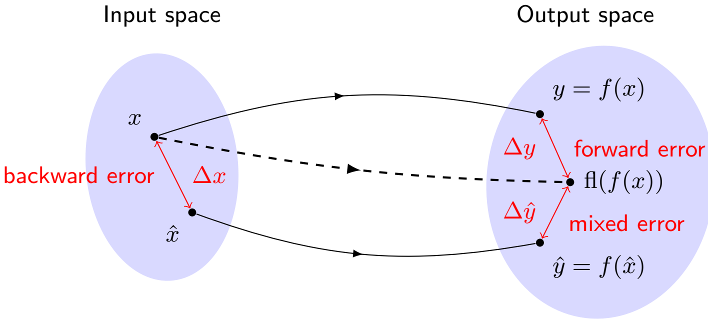
```
for (size_t j=0;j<4;j++) {      // loop over cols.
    for (size_t i=0;i<3;i++) {  // loop over rows
        matc[i+j*3] = j*3.0 + i;
    }
}
```

Forward and backward error

Recall floating-point model (assuming no overflow/underflow)

$$\mathrm{fl}(f(x)) = (1+\delta_1)f((1+\delta_2)x)$$

where δ_1 and δ_2 represent errors



Conditioning

Suppose f is twice continuously differentiable and let $\hat{x} = x + \Delta x$

$$\hat{y} - y = f(\hat{x}) - f(x) = f'(x)\Delta x + \frac{f''(x + \theta\Delta x)}{2!}(\Delta x)^2 \qquad \text{for some } \theta \in (0,1)$$

follows from the *Remainder Theorem*

Implies that for small Δx (and $y \neq 0$)

$$\frac{\hat{y} - y}{y} = \frac{xf'(x)}{f(x)}\frac{\hat{x} - x}{x} + O((\Delta x)^2)$$

rel. forward error $\approx$ condition number $\times$ rel. backward error

Relative condition number of f

$$c(x) = \left|\frac{xf'(x)}{f(x)}\right|$$

Error analysis of sequential summation

- Compute sum sequentially as follows

$$s_1 = x_1, \quad s_k = s_{k-1} + x_k, \quad k = 2, \dots, n$$

- Using our floating-point model

$$\hat{s}_k = \mathrm{fl}(\hat{s}_{k-1} + x_k) = (\hat{s}_{k-1} + x_k)(1 + \delta_k), \quad |\delta_k| \le u, \quad k = 2, \dots, n$$

we arrive at

$$\begin{aligned}
\hat{s}_n &= (x_1 + x_2) \prod_{k=2}^{n} (1 + \delta_k) + \sum_{i=3}^{n} x_i \prod_{k=i}^{n} (1 + \delta_k) \\
&= (x_1 + x_2)(1 + \theta_2) + \sum_{i=3}^{n} x_i (1 + \theta_i), \qquad 1 + \theta_i = \prod_{k=i}^{n} (1 + \delta_k) \\
&= s_n + \theta_2 x_1 + \sum_{i=2}^{n} \theta_i x_i
\end{aligned}$$

Error analysis of sequential summation (cont.)

- Lower and upper bounds: use $|\delta_k| \le u$ and inequality $e^x > 1 + x$ for $x \neq 0$

$$e^{-(n-i+1)u} < (1-u)^{n-i+1} \le \underbrace{\prod_{k=i}^{n}(1+\delta_k)}_{1+\theta_i} \le (1+u)^{n-i+1} < e^{(n-i+1)u}$$

- Apply inequality $e^{-x} \ge 1 - x$ to lower and upper bound (assumption: $nu < 1$)

$$1 - nu < 1 + \theta_i < \frac{1}{1-nu} = 1 + \frac{nu}{1-nu} = 1 + \hat{\theta}$$

- It follows that

$$|\hat{s}_n - s_n| \le |\theta_2||x_1| + \sum_{i=2}^{n} |\theta_i||x_i| \le \hat{\theta} \sum_{i=1}^{n} |x_i|$$

which leads to the following upper bound on the relative error

$$\frac{|\hat{s}_n - s_n|}{|s_n|} \le \frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|} \frac{nu}{1-nu}$$

Simplified error bound

It follows from the previous slide that

$$1 + \theta_i < e^{nu} = 1 + nu + \frac{(nu)^2}{2!} + \frac{(nu)^3}{3!} + \cdots$$

and hence

$$\theta_i < e^{nu} - 1 = nu\left(1 + \frac{nu}{2} + \frac{(nu)^2}{3!} + \cdots\right) < nu \underbrace{\left(1 + \frac{nu}{2} + \left(\frac{nu}{2}\right)^2 + \cdots\right)}_{\sum_{k=0}^{\infty}\left(\frac{nu}{2}\right)^k}$$

The right-hand side is the sum of a geometric series (converges if $nu/2 < 1$)

$$\sum_{k=0}^{\infty}\left(\frac{nu}{2}\right)^k = \frac{1}{1 - \frac{nu}{2}}$$

It follows that if $nu < 0.1$, then

$$\frac{|\hat{s}_n - s_n|}{|s_n|} \leq \frac{\sum_{i=1}^{n}|x_i|}{|\sum_{i=1}^{n} x_i|} 1.06nu$$

Simplified error bound (cont.)

Recall that for binary64 (double precision) we have

$$u = 2^{-53}$$

and hence $nu < 0.1$ implies that $n < 0.1 \cdot 2^{53} \approx 9 \cdot 10^{14}$

Given $n = 9 \cdot 10^{14}$ double precision floating-point numbers

- storage would require $8n$ bytes or approximately 7.2 PB (petabytes)
- summation at 300 GFLOPS would take

$$\frac{9 \cdot 10^{14} \text{ flops}}{300 \cdot 10^{9} \text{ flops/s}} = 3{,}000 \text{ s}$$

- retrieving from RAM at 50 GB/s would take around 40 hours
- retrieving from a harddrive at 120 MB/s would take around 1.9 years

Kahan's summation algorithm (compensated summation)

```
/* Compensated summation of array x */
double sum = 0.0, c = 0.0, t, y;
for (size_t i=0;i<n;i++) {
    y = x[i] - c;
    t = sum + y;
    c = (t - sum) - y;
    sum = t;
}
```

- In exact arithmetic, we have $c = (t - \text{sum}) - y = 0$
- Associative property $(a + b) + c = a + (b + c)$ not satisfied for FP arithmetic
- It can be shown that the relative error satisfies

$$\epsilon_{\text{rel}} \leq \frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|}(2u + O(nu^2))$$

- Several other compensated sum algorithms exist

Summary

- Automatic array allocation (compile-time and run-time allocation)
- Pointers (variables that hold a memory address)
- Forward error, backward error, and condition number
- Error analysis of sequential sum

$$\frac{|\hat{s}_n - s_n|}{|s_n|} \leq \frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|} 1.06nu, \quad nu < 0.1$$

- Condition number for sequential summation

$$\frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|}$$

- Compensated summation