Mathematical Software Programming (02635)

Lecture 4 — September 25, 2025

Instructor: Martin S. Andersen

Fall 2025

This week

Topics

- Condition numbers
- Numerical stability
- Error analysis for sequential summation
- Pointers

Learning objectives

- Describe and use data structures such as **arrays**, linked lists, **stacks**, and queues.
- Choose appropriate data types and data structures for a given problem.
- Design, implement, and document a program that solves a mathematical problem.

Conditioning

How *sensitive* is $f(x)$ to perturbations of x (in a relative sense)?

$$\frac{\text{Relative output error}}{\text{Relative input error}} = \frac{\|f(x+\Delta x) - f(x)\|/\|f(x)\|}{\|\Delta x\|/\|x\|}$$

Relative condition number

$$\operatorname{cond}(f,x) = \lim_{\epsilon \to 0^+} \sup_{\|\Delta x\| \le \epsilon} \frac{\|f(x+\Delta x) - f(x)\|/\|f(x)\|}{\|\Delta x\|/\|x\|}$$

The problem of evaluating f at x is

- well-conditioned if $\operatorname{cond}(f,x)$ is *small*
- ill-conditioned if $\operatorname{cond}(f,x)$ is *large*
- ill-posed if $\operatorname{cond}(f,x)$ is *infinite*

Conditioning: univariate functions

Suppose $f\colon \mathbb{R} \to \mathbb{R}$ 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 Taylor's theorem with Lagrange form remainder)

Implies that when $|\Delta x|$ is sufficiently small,

$$\frac{|\hat{y} - y|}{|y|} \approx \frac{|xf'(x)|}{|f(x)|}\frac{|\hat{x} - x|}{|x|}, \qquad y \neq 0,\ x \neq 0,$$

rel. output error $\approx$ condition number $\times$ rel. input error

Absolute and relative condition number of f at x

$$\mathrm{cond}^{\mathrm{abs}}(f, x) = |f'(x)| \qquad \mathrm{cond}^{\mathrm{rel}}(f, x) = \frac{|xf'(x)|}{|f(x)|}$$

Conditioning: multivariate functions

Suppose $f\colon \mathbb{R}^n \to \mathbb{R}^m$ is twice continuously differentiable and let $\Delta x = \hat{x} - x$

$$\hat{y} - y = f(\hat{x}) - f(x) = J_f(x)\Delta x + o(\|\Delta x\|)$$

Jacobian matrix

$$J_f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Conditioning: multivariate functions (continued)

Absolute condition number

$$\mathrm{cond}^{\mathrm{abs}}(f, x) = \|J_f(x)\|$$

Relative condition number

$$\mathrm{cond}(f, x) = \frac{\|J_f(x)\|\|x\|}{\|f(x)\|}$$

- depends on the choice of norms
- implies that when $\|\Delta x\|$ is sufficiently small,

$$\frac{\|f(x + \Delta x) - f(x)\|}{\|f(x)\|} \lessapprox \mathrm{cond}(f, x) \frac{\|\Delta x\|}{\|x\|}$$

Examples

Univariate functions

$f(x)$	$\mathrm{cond}(f, x)$
$1 - x$	$\lvert x\rvert / \lvert 1 - x\rvert$
$\log(x)$	$1 / \lvert \log(x)\rvert$
$\exp(x)$	$\lvert x\rvert$
x^a	$\lvert a\rvert$

Multivariate functions

$f(A)$	$\mathrm{cond}_2(f, A)$
Ab	$\lVert A\rVert_2 \lVert b\rVert_2 / \lVert Ab\rVert_2$
A^{-1}	$\lVert A\rVert_2 \lVert A^{-1}\rVert_2$

Example: sequential summation

Let $f\colon \mathbb{R}^n \to \mathbb{R}$ be defined as

$$f(x) = x_1 + x_2 + \cdots + x_n$$

with Jacobian matrix $J_f(x) = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}$

Relative 1-norm condition number of summation

$$\operatorname{cond}_1(f, x) = \frac{\|x\|_1 \|J_f(x)\|_1}{|f(x)|} = \frac{\sum_{i=1}^n |x_i|}{|\sum_{i=1}^n x_i|}$$

- $\operatorname{cond}_1(f, x) = 1$ if $x_1, \ldots, x_n$ are all positive or negative
- ill-conditioned if $\|x\|_1 \gg |f(x)|$
- ill-posed when $f(x) = 0$

Example: sample variance

A vector $x \in \mathbb{R}^n$ is *centered* by subtracting $\mu = \frac{1}{n}\sum_{i=1}^n x_i$ componentwise:

$$\bar{x}_i \leftarrow x_i - \mu, \quad i \in \{1, \ldots, n\}$$

Equivalently, letting $\mathbf{1}_n$ denote the n-dimensional vector of ones:

$$\bar{x} \leftarrow C_n x \qquad \text{where} \quad C_n = I - (1/n)\mathbf{1}_n\mathbf{1}_n^T \quad (C_n^2 = C_n)$$

The sample variance of x is

$$f(x) = \frac{1}{n-1}\sum_{i=1}^n (x_i - \mu)^2 = \frac{1}{n-1}\|C_n x\|_2^2$$

and

$$\mathrm{cond}_2(f, x) = \frac{\|J_f(x)\|_2 \|x\|_2}{|f(x)|} = \frac{2\|x\|_2}{\|C_n x\|_2}$$

Numerical stability

What is the behavior of an algorithm when small errors are introduced during the computations?

An algorithm is said to be

- *stable* if, for every input, such errors have an insignificant impact on the result,
- *unstable* if the impact is significant for one or more inputs.

Forward and backward error

Suppose $y = f(x)$ but our algorithm outputs an approximation $\hat{y}$.

Relative forward error

$$\frac{\|\hat{y} - y\|}{\|y\|}$$

Relative backward error

Smallest perturbation of x (in some norm) such that $\hat{y} = f(x + \Delta x)$

$$\inf_{\Delta x} \left\{ \frac{\|\Delta x\|}{\|x\|} : \hat{y} = f(x + \Delta x) \right\}$$

(Δx such that $\hat{y} = f(x + \Delta x)$ may or may not exist)

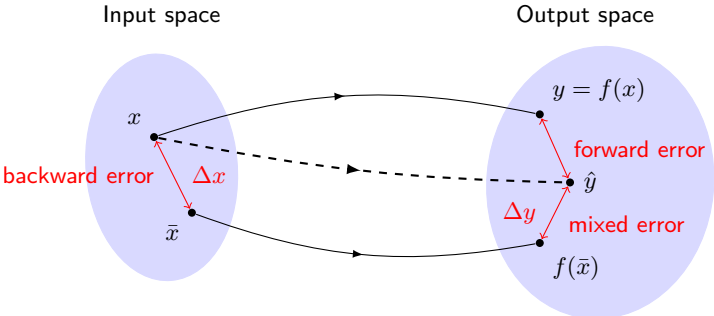
Stability analysis and error bounds

An algorithm is

- *forward stable* if $\|\hat{y} - y\| \leq c \cdot \mathrm{cond}(f, x) \cdot \|y\|$ is always satisfied for some small constant c,
- *backward stable* if the backward error is always small (implies forward stability), and
- *mixed forward-backward stable* if

$$\hat{y} + \Delta y = f(x + \Delta x), \qquad \|\Delta y\| \leq \delta_1 \|y\|, \ \|\Delta x\| \leq \delta_2 \|x\|,$$

for some small constants δ_1 and δ_2.



Example

Suppose we want to evaluate $f(x) = \exp(x)$ using the series expansion

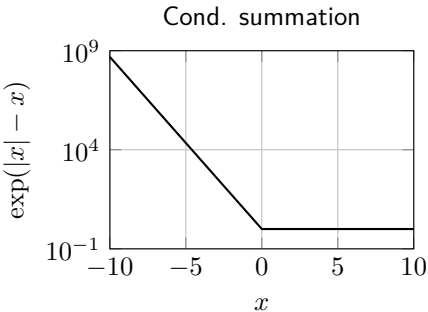
$$f(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \cdots$$

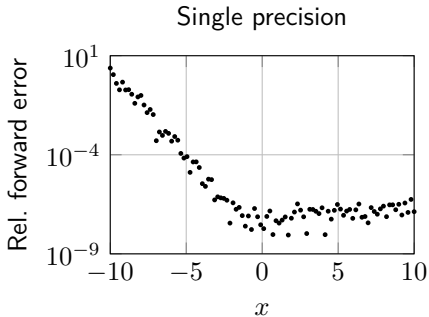
Algorithm: add terms until the sum does not change due to finite precision

```
float sum=1, oldsum=0; term = 1;
int k = 1;
while (sum != oldsum) {
    oldsum = sum;
    term *= x/k++;  // recursive update
    sum += term;
}
```

- recursion based on identity $\frac{x^k}{k!} = \frac{x^{k-1}}{(k-1)!}\frac{x}{k}$ (for $k = 1, 2, \ldots$)
- implies that $\frac{x^k}{k!} \to 0$ as $k \to \infty$

Example (continued)





- relative condition number for series summation is $\exp(|x| - x)$
- relative forward error is large when x is negative (but $\mathrm{cond}(f, x) = |x|$ is small)
- quick fix: use identify $x = |x|\mathrm{sgn}(x)$ to rewrite $f(x) = \exp(|x|)^{\mathrm{sgn}(x)}$

Disclaimer: this is not how the exponential function is implemented in practice

Error analysis: sequential summation

- Compute sum sequentially as follows

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

- Using our floating-point model

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

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 $|\delta_k| \leq u$ and inequality $1 + x < e^x$, $x \neq 0$

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

- Apply inequality $e^{-x} \geq 1 - x$ to 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

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

  which leads to the following upper bound on the relative error

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

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

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

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 7.2 PB (petabytes)
- sequential summation at 300 Gflops/s 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 $c = (t - \mathrm{sum}) - y = 0$
- associative property $(a + b) + c = a + (b + c)$ not satisfied for FP arithmetic
- it can be shown that the relative forward error satisfies the bound

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

- several other compensated sum algorithms exist

C uses *call-by-value* method to pass arguments

```c
#include <stdio.h>
void swap(int a, int b);   // Function prototype
int main(void) {
    int a = 1, b = 3;
    swap(a,b);
    printf("a = %d and b = %d\n",a,b);
    return 0;
}
void swap(int a, int b) {
    int c = a;  // Store value of a in c
    a = b;      // Overwrite a with b
    b = c;      // Overwrite b with c
    return;
}
```

What is the value of `a` and `b` after calling `swap(a,b)`?

Pointers

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

Pointers as arguments

```c
#include <stdio.h>
void swap2(int * a, int * b);   // Function prototype
int main(void) {
    int a = 1, b = 3;
    swap2(&a,&b);
    printf("a = %d and b = %d\n",a,b);
    return 0;
}
void swap2(int * a, int * b) {
    int c = *a;  // Store value of *a in c
    *a = *b;     // Overwrite *a with *b
    *b = c;      // Overwrite *b with c
    return;
}
```

What are the values of `a` and `b` in `main` after calling `swap2(&a,&b)`?

Pointers and arrays

```c
/* 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?

Example: sum of elements in array of length n

```
double sum(double * a, int n) {
    double s=0.0;
    for (int i=0; i<n; i++)
        s += a[i];    // same as s += *(a+i)
    return s;
}
```

```
double b[] = {1.0,2.0,3.0};
printf("sum(b) = %.4f\n", sum(b,3));
```

Example: sum of elements in two-dimensional array of size m-by-n

```
double sum2d(double * a, int m, int n) {
    double s=0.0;
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++)
            s += a[j+i*n];  // row-major storage
    }
    return s;
}
```

```
double c[2][3] = {{1.0,2.0,3.0},{4.0,5.0,6.0}};
printf("sum2d(*c) = %.4f\n", sum2d(*c,2,3));
```

`*c` points to `c[0][0]` whereas `**c` is the value `c[0][0]`

Pointers as return values

```
int * max(int * a, int * b) {
    if (*a > *b)
        return a;
    else
        return b;
}
```

Never return a pointer to an *automatic* local variable.

```
int * f(void) {
    int i;
    ...
    return &i;
}
```

Protecting arguments with const

```
void mean_var(const double *a, int n, double *mean, double *var) {
    if (n < 1) { return; }
    *mean = 0.0;
    for (int i=0;i<n;i++) *mean += a[i];
    *mean /= n;
    *var = 0.0;
    for (int i=0;i<n;i++) {
        *var += (a[i] - *mean)*(a[i] - *mean);
    }
    if (n > 1) *var /= (n-1);
}
```

- compiler objects to write operations like `*a = 0;` and `a[i] = 4;`
- the pointer itself, `a`, is a local variable (*passed by value*) and can be overwritten
- `const double *` means *pointer to a const double*
- `double * const` means *const pointer to a double*

Exercises

- Conditioning and stability
- Quiz
- Exercises on Autolab
- Analyze/rewrite programs with pointers