Numerical Methods for Partial Differential Equations

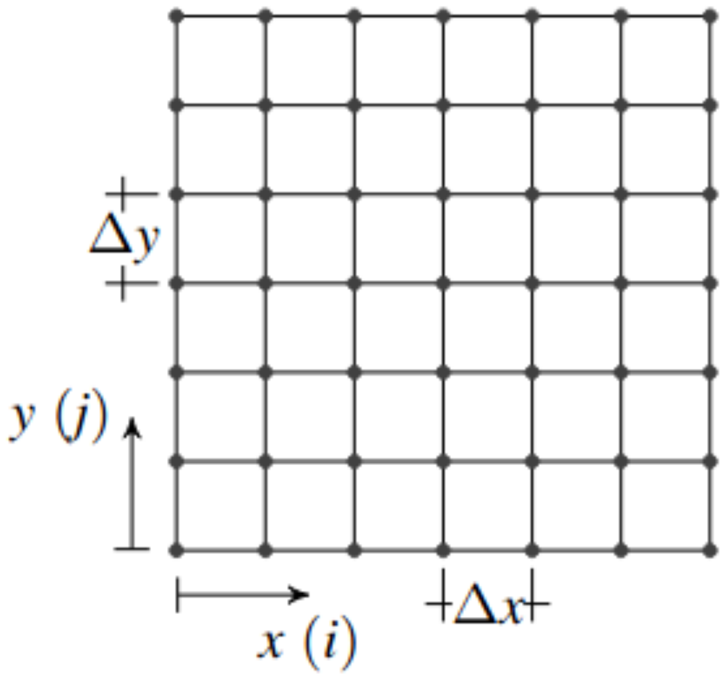
Lecture 3: Introduction to Finite Differences

Finite Difference

- In order to solve numerically a PDE we have to give a discrete representation of the unknown function.
- One approach is to discretize the continuous problem domain so that the unknown functions is considered to exist only at discrete points.
- We establish a grid on the domain by replacing $u(x, y)$ by $u(i\Delta x, j\Delta y)$
- Another approach we approximate a function u(x) defined in an interval [a,b] by some set of basis functions

$$u(x) = \sum_{i=1}^{n} A_i \, \varphi_i(x)$$

- spectral methods use basis functions that are generally nonzero over the whole domain (sines, cosines more generally exponentials (imaginary argument).
- finite element methods use basis functions that are nonzero only on small subdomains



Finite Difference Method

- Let us suppose that we are looking for the derivative of a function `f(x)` at some given point x.
- Assume that the function f(x) is known at equally spaced point x_i, such that $h = x_{i+1} - x_i$ is the spacing between nodes. Let

$$f_i = f(x_i) \qquad \text{for} \quad i = 0, \ldots, N_x - 1$$

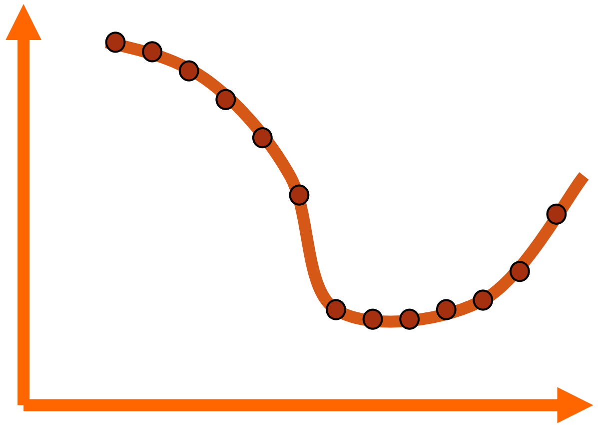
- In order to find the derivative *f'* = *df*/*dx*, the most direct method expands the function using a Taylor series in the neighborhood of x_i:

$$f_{i+1} \equiv f(x_i + h) \approx f_i + f_i' h + \frac{f_i''}{2} h^2 + \frac{f_i'''}{3!} h^3 + O(h^4)$$

- Solving for f'_i, we have the *forward difference (FD)* approximation:

$$\boxed{f_i' \approx \frac{f_{i+1} - f_i}{h} - \frac{f_i''}{2} h}$$

- This approximation has an error proportional to *h*: we can make the approximation error smaller by making *h* smaller, yet precision will be lost through the subtractive cancellation on the left-hand side when h is too small.



Backward Difference

- Similarly, we could expand $f(x_i\text{-}h)$:

$$f_{i-1} \equiv f(x_i - h) \approx f_i - f_i' h + \frac{f_i''}{2} h^2 - \frac{f_i'''}{3!} h^3 + O(h^4)$$

and obtain the *backward difference (BD)* approximation

$$\boxed{f_i' \approx \frac{f_i - f_{i-1}}{h} + \frac{f_i''}{2} h}$$

which still has the same error $O(h)$.

- Both the forward and backward approximations are only first-order accurate and would give the correct answer only when $f(x)$ is a linear function.
- For a quadratic function $f(x)=a+bx^2$, for instance, the forward derivative approximation would result in

$$\frac{f_{i+1} - f_i}{h} = 2bx_i + bh$$

- If you compare it with the exact derivative ($f' = 2bx$), this clearly becomes a good approximation only for small h ($h << 2x_i$)

Central Difference

- Now consider both the right and left expansions:

$$\begin{cases} f_{i+1} & \approx & f_i + f_i' h + \dfrac{f_i''}{2} h^2 + \dfrac{f_i'''}{3!} h^3 + O(h^4) \\ f_{i-1} & \approx & f_i - f_i' h + \dfrac{f_i''}{2} h^2 - \dfrac{f_i'''}{3!} h^3 + O(h^4) \end{cases}$$

- Subtracting the two equations yields the *central difference (CD)* approximation

$$\boxed{f_i' = \frac{f_{i+1} - f_{i-1}}{2h} - \frac{f'''}{6} h^2}$$

- During the subtraction, even powers cancel and our approximation is thus second-order accurate: you can expect the cd approximation to be exact for a parabola.

- The FD, BD and CD approximations are quite natural in the sense that they are reminiscent of the incremental ratio used in elementary calculus.

Higher Order Formulas

- It is possible to obtain higher-order, more accurate, approximation by including more points.
- If we now expand also f_{i+2} and f_{i-2}, we obtain a system of equations

$$\begin{cases} f_{i+2} & \approx & f_i + 2f_i'h + \dfrac{f_i''}{2}(2h)^2 + \dfrac{f_i'''}{3!}(2h)^3 + O(h^4) \\ f_{i+1} & \approx & f_i + f_i'h + \dfrac{f_i''}{2}h^2 + \dfrac{f_i'''}{3!}h^3 + O(h^4) \\ f_{i-1} & \approx & f_i - f_i'h + \dfrac{f_i''}{2}h^2 - \dfrac{f_i'''}{3!}h^3 + O(h^4) \\ f_{i-2} & \approx & f_i - 2f_i'h + \dfrac{f_i''}{2}(2h)^2 - \dfrac{f_i'''}{3!}(2h)^3 + O(h^4) \end{cases}$$

- Getting rid of terms up the fourth derivative, we obtain

$$\boxed{f_i' \approx \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12h} + \frac{h^4}{30}f^{(5)}}$$

which is a 4th - order accurate approximation.

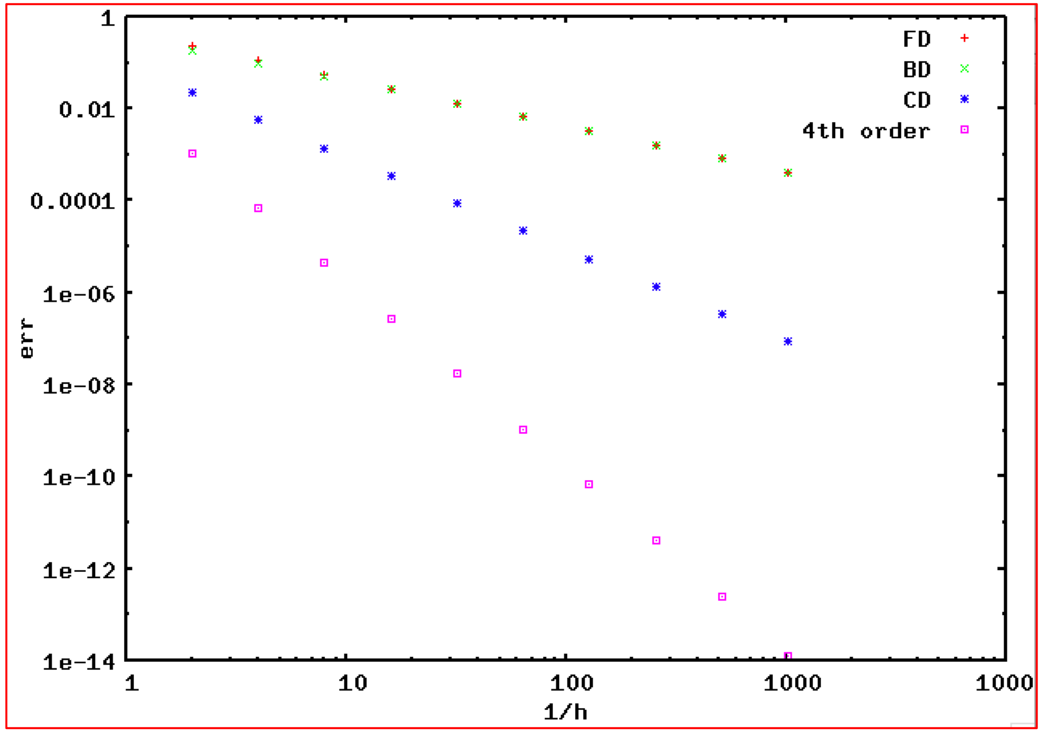
Example #1

- Write a program to compute the numerical derivative `f(x) = sin(x)` in *x=1* using FD, BD and CD (or higher) using different increments *h=0.5,0.25,0.125, …*

  Plot the error

$$\epsilon = |f'_{\mathrm{num}} - f'_{\mathrm{ex}}|$$

as a function of h using a log-log scaling.



2^{nd}- and Higher-order Derivatives

- For higher order derivatives we can still make use of the Taylor expansion and solve for the second (or higher) derivative.
- From

$$\begin{cases} f_{i+1} & \approx & f_i + f_i' h + \dfrac{f_i''}{2} h^2 + \dfrac{f_i'''}{3!} h^3 + O(h^4) \\ f_{i-1} & \approx & f_i - f_i' h + \dfrac{f_i''}{2} h^2 - \dfrac{f_i'''}{3!} h^3 + O(h^4) \end{cases}$$

we can solve, e.g., for the 2^{nd} derivative:

$$\boxed{f_i'' \approx \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2)}$$

Including more points:

$$\boxed{F''(x) = \frac{-F(x+2\Delta x) + 16F(x+\Delta x) - 30F(x) + 16F(x-\Delta x) - F(x-2\Delta x)}{12\Delta x^2} + O((\Delta x)^4)}$$

Example #2

- In order to increase accuracy, is it better to decrease *h* or *increase* the order (i.e. the stencil) ?
- Compute the 2° derivative of the function e^x for h = 0.1, 0.01, … 10^{-5}. Is the error decreasing or not ?

```
 --------------------------------------------------
1.000000e-01  2.265990e-03
1.000000e-02  2.265242e-05
1.000000e-03  2.265441e-07
1.000000e-04  3.780617e-08
1.000000e-05  5.988602e-06
```

- The error does not decrease for small h because function values becomes very close → loss of accuracy.
- Sources of error:

1. Finite number representation (round-off error);
2. Truncation error (finite number of terms in, e.g., Taylor series).

Arithmetic Precision

- Where is the error coming from ?

1. Discretization error (approximation to given order for the derivative → *truncation error*);
2. Internal number representation (→ *round off error*)

Float and Double precision datatype

- *Singles* or *floats* is shorthand for *single- precision floating-point numbers and* occupy 32 bits: 1 bit for the sign, 8 bits for the exponent, and 23 bits for the fractional mantissa:

	s	e		f	
Bit position	31	30	23	22	0

```
EXAMPLE: IEEE-754 Single-Precision representation of: 3.141590

 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0
|- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -|
|s|      exp       |                  mantissa                  |
```

- The sign bit *s* is in bit position 31, the biased exponent *e* is in bits 30–23, and the fractional part of the mantissa *f* is in bits 22–0. Since 8 bits are used to store the exponent *e* and since $2^8 = 256$ → $0 \leq e \leq 255$.
- Likewise $-126 \leq e \leq 127$.
- In summary, single-precision (32-bit or 4-byte) numbers have six or seven decimal places of significance and magnitudes in the range

$$\boxed{1.4 \times 10^{-45} \leq \text{single precision} \leq 3.4 \times 10^{38}}$$

Float and Double precision datatype

- Doubles are stored as two 32-bit words, for a total of 64 bits (8 B). The sign occupies 1 bit, the exponent e, 11 bits, and the fractional mantissa, 52 bits:

	s	e	f	f (cont.)
Bit position	63	62 52	51 32	31 0

- The fields are stored contiguously, with part of the mantissa f stored in separate 32-bit words.
- Doubles have approximately 16 decimal places of precision (1 part in 252) and magnitudes in the range

$$\boxed{4.9 \times 10^{-324} \leq \text{double precision} \leq 1.8 \times 10^{308}.}$$

C and C++ Data-Type Range

In 1987, the Institute of Electrical and Electronics Engineers (IEEE) and the American National Standards Institute (ANSI) adopted the IEEE 754 standard for floating-point arithmetic. When the standard is followed, you can expect the primitive data types to have the precision and ranges given by the following table

Key word	Size in bytes	Interpretation	Possible values
bool	1	boolean	true and false
unsigned char	1	Unsigned character	0 to 255
char (or signed char)	1	Signed character	-128 to 127
wchar_t	2	Wide character (in windows, same as unsigned short)	0 to 2^{16}-1
short (or signed short)	2	Signed integer	-2^{15} to 2^{15} - 1
unsigned short	2	Unsigned short integer	0 to 2^{16}-1
int (or signed int)	4	Signed integer	- 2^{31} to 2^{31} -1
unsigned int	4	Unsigned integer	0 to 2^{32} - 1
Long (or long int or signed long)	4	signed long integer	- 2^{31} to 2^{31} -1
unsigned long	4	unsigned long integer	0 to 2^{32} -1
float	4	Signed single precision floating point (23 bits of significand, 8 bits of exponent, and 1 sign bit.)	$3.4*10^{-38}$ to $3.4*10^{38}$ (both positive and negative)
long long	8	Signed long long integer	-2^{63} to 2^{63} -1
unsigned long long	8	Unsigned long long integer	0 to 2^{64} -1
double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	$1.7*10^{-308}$ to $1.7*10^{308}$ (both positive and negative)
long double	8	Signed double precision floating point(52 bits of significand, 11 bits of exponent, and 1 sign bit.)	$1.7*10^{-308}$ to $1.7*10^{308}$ (both positive and negative)

Overflow and Underflow

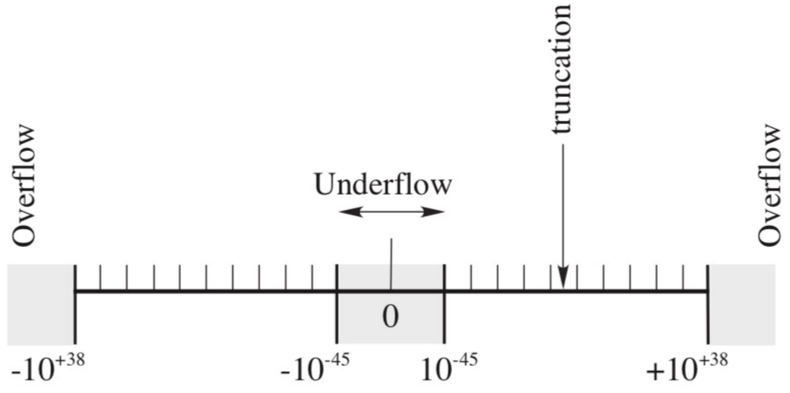


Figure 1.7 The limits of single-precision floating-point numbers and the consequences of exceeding these limits. The hash marks represent the values of numbers that can be stored; storing a number in between these values leads to truncation error. The shaded areas correspond to over- and underflow.

- If a single-precision number $x > 2^{128}$, a fault condition known as an *overflow* occurs. The resulting number x_c may end up being a machine-dependent pattern, not a number (NAN), or unpredictable.
- If $x < 2^{-128}$, an *underflow* occurs. The resulting number x_c is usually set to zero, although this can usually be changed via a compiler option.
- In our experience, *serious scientific calculations almost always require at least 64-bit (double-precision) floats*. And if you need double precision in one part of your calculation, you probably need it all over, which means double-precision library routines for methods and functions.

Example #3: determining machine precision

- The loss of precision is categorized by defining the *machine precision* ϵ_m as the maximum positive number that can be added unity without changing it:

$$\boxed{1_c + \epsilon_m \stackrel{\text{def}}{=} 1_c,}$$

where the subscript *c* is a reminder that this is a computer representation of 1.

- Consequently, an arbitrary number *x* can be thought of as related to its floating- point representation x_c by

$$x_c = x(1 \pm \epsilon), \quad |\epsilon| \leq \epsilon_m,$$

but the actual value for ε is not known.

- In other words, except for powers of 2 that are represented exactly, we should assume that all single-precision numbers contain an error in the sixth decimal place and that all doubles have an error in the fifteenth place.
- `precision.cpp:` write a computer program to determine the machine precision. Define 1 in float (or double) precision arithmetic and keep adding epsilon (→epsilon/10) until 1+eps = 1.

Example #4: Function evaluation

- Consider the polynomial

$$f(x) = x^7 - 7x^6 + 21x^5 - 35x^4 + 35x^3 - 21x^2 + 7x - 1$$

- Write a code that employs single-precision to produce equally spaced values in the range $0 < x < 2$ using NX = 250 points.
- Plot your data around x=1. What do you see ? Why ? Can you improve the situation ?

A Special Class of Functions: Polynomials

- Consider $P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$
- If you're thinking about doing `F(x) = an*pow(x,n) + an-1*pow(x,n-1) + ... a1*x + a0` by using looping like:

```
double P = 0
for (int i = 0; i <= n; i++) P += a[n]*pow(x,n);     // NOOOOO !!!!!!!
```

don't even dare! (It's obvious that there's a lot of repetitive computations being done by raising x to successive powers).

This method is quite inefficient: it requires *n additions* and *n(n+1)/2 multiplications*.

- A possibility would be an *iterative method*, by simply keeping the previous power of x between iterations:

```
double P = 0.0, xn = 1.0;
for (int i = 0; i <= n; i++){
  P += a[i]*xn;
  xn *= x;              // the current power of x
}
```

It's easy to see that there are *2n multiplications* and *n additions* for each computation. The algorithm is now linear instead of quadratic.

Horner's Method for Polynomial Evaluation

- An even cheaper solution is given by ***Horner***'s Method. Take

$$P(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

- Divide the polynomial into monomials starting from the largest power: the result obtained from one monomial is added to the result obtained from the next monomial and so forth in an addition fashion. Then you rewrite

$$P(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + xa_n))$$

  Each monomial involves a maximum of one multiplication and one addition processes: *n multiplications* and *n additions* are involved !

- With a simple modification, we can also obtain the derivative at the same time:

```
p    = a[n];
dpdx = 0;
for (int j = n-1; j >= 0; j--){
  dpdx = dpdx*x + p;
  p    = p*x + a[j];
}
```