Lecture 9

Monday, January 27, 2020

Sources of errors:

① Physical modeling:

When modeling a problem in real life by a mathematical problem, some simplifications are made to simplify problem or to isolate the difficulty. For example, when modeling the motion of a pendulum, one usually assumes that the string is massless, non-expanding nor contracting, and that there is no air resistance.

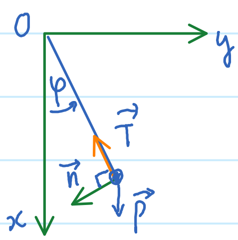
The equation of the angle $\varphi = \varphi(t)$ is

$$\ddot{\varphi} + \frac{g}{\ell} \sin\varphi = 0. \qquad (*)$$

This equation is a model of the real motion of the pendulum.

* Side notes:

Equation $(*)$ can be derived using Newton's second law as follows.



$$\vec{P} + \vec{T} = m\vec{a} \qquad (**)$$

where $\vec{P}$ is the gravitional force,

$\vec{T}$ is the tension force caused by the string,

$\vec{a}$ is the acceleration vector.

We have $\vec{P} = m\vec{g} = mg(1,0)$.

The position vector is $\vec{r} = (\ell\cos\varphi, \ell\sin\varphi)$

The velocity vector is $\vec{v} = \frac{d\vec{r}}{dt} = (-\ell\dot{\varphi}\sin\varphi, \ell\dot{\varphi}\cos\varphi)$

The acceleration vector is $\vec{a} = \frac{d\vec{v}}{dt} = (-\ell(\dot{\varphi})^2\cos\varphi - \ell\ddot{\varphi}\sin\varphi,$
$-\ell(\dot{\varphi})^2\sin\varphi + \ell\ddot{\varphi}\cos\varphi).$

The direction vector of the position vector is $(\cos\varphi, \sin\varphi)$. A normal vector of the position vector is $\vec{n} = (-\sin\varphi, \cos\varphi)$.

Project both sides of (**) on $\vec{n}$ (in other words, take the dot product of both sides of (**) with $\vec{n}$):

$$\vec{P}\cdot\vec{n} + \underbrace{\vec{T}\cdot\vec{n}}_{=0} = m\vec{a}\cdot\vec{n}$$

Thus, $mg(1,0)\cdot(-\sin\varphi, \cos\varphi) = m(-l(\dot\varphi)^2\cos\varphi - l\ddot\varphi\sin\varphi,$
$\qquad -l(\dot\varphi)^2\sin\varphi + l\ddot\varphi\cos\varphi)\cdot(-\sin\varphi, \cos\varphi)$
$\qquad = ml\ddot\varphi$

We get $\ddot\varphi + \dfrac{g}{l}\sin\varphi = 0.$

② Mathematical approximation:

This is a type of error due to alternating the equation. For example, by making the approximation $\sin\varphi \approx \varphi$, one can write (**) as

$$\ddot\varphi + \frac{g}{l}\varphi = 0$$

The approximation makes it easier to solve for φ as a function of t. Taylor approximation also causes this type of error because we approximate a function by a polynomial.

This type of error is usually controllable. We have seen, for example, that the error term coming from Taylor approximation can be made small by increasing n (the order of Taylor polynomial).

③ Machine representation:

This type of error is quite random in nature. It is hard to control, but is usually small For example,

Let $f(x) = x^2$. Compute $f'(1)$.

One can easily see that $f'(x) = 2x$. Thus, $f'(1) = 2$.

But let's assume that a neat formula for $f'(x)$ is not available.

By the definition of derivative,

$$f'(1) = \lim_{h\to 0} \frac{(1+h)^2-1}{h} \underset{\substack{\uparrow \\ \text{error by} \\ \text{mathematical} \\ \text{approximation}}}{\approx} \frac{(1+10^{-6})^2-1}{10^{-6}} \underset{\substack{\uparrow \\ \text{error by} \\ \text{machine} \\ \text{representation}}}{\approx} 2.000000999$$

In practice, the errors caused by different sources are accummulated. Let e_1 be the error caused by mathematical approximation, and e_2 be the error caused by machine representation. Suppose these are the only kinds of errors we are interested in. The accumulated is

$$e = e_1 + e_2.$$

We want that e_1 is small, but much larger than e_2 so that e is essentially e_1, which is controllable. If e_1 is too small, e_2 will dominate e_1 and e is essentially e_2. Then we don't have a control on e. For example,

$$f'(1) = \lim_{h\to 0} \frac{(1+h)^2-1}{h} \underset{\substack{\uparrow \\ e_1 \text{ is} \\ \text{very small}}}{\approx} \frac{(1+10^{-16})^2-1}{10^{-16}} \underset{\substack{\uparrow \\ e_2 \text{ is big}}}{\approx} 0.$$

Being ambitions to make e_1 too small may cause us to pay a price (big e_2).

<u>Root-finding problems</u>

A lot of problems in real life can be modeled as solving for x from the equation $f(x) = 0$

Here x can be a number, a vector, a matrix, a sequence or a function. Function f can be as simple as a degree-one polynomial and as complex as a differential operator.

<u>Ex</u>:

$$\underbrace{\ddot{\varphi} + \frac{g}{\ell}\varphi}_{f(\varphi)} = 0$$

How to solve for x knowing f?

One can notice that simple functions f can make it very difficult to find x exactly.

- If $f(x) = ax+b$ then $x = -b/a$
- If $f(x) = ax^2+bx+c$ then $x = \dfrac{-b \pm \sqrt{b^2-4ac}}{2a}$
- If $f(x) = ax^3+bx^2+cx+d$ then x is given by Cardano's formula (1540s).
- If $f(x) = ax^4+bx^3+cx^2+dx+e$ then there is a formula for one of the roots (by Ferrari in 1540s).
- If $f(x)$ is of order 5 or higher, one cannot solve for x by only using the four basic operations (addition, subtraction, multiplication, division) and radicals (square root, third root, fourth root, ...) This was proved by Abel and Ruffini in 1820s.

From a practical point of view, one often only wants an approximate value of x given a prescribed error. For example, given $\varepsilon = 10^{-6}$, find an approximate root x_0 of $f(x)=0$ such that $|x_0 - \underbrace{x_*}_{\text{true root}}| < \varepsilon$.

* Bisection method:

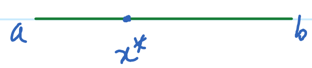
This simple method is based on an idea similar to that of binary search. Suppose we are to search for element $x=2$ in an array n of 100 numbers

$$a_1, a_2, a_3, \dots, a_{100}.$$

A simple method is to examine from left to right, checking if a_k is equal to 2. This method is called linear search. The number of checkings is of order n $(=100)$.

A fast method is as follow. First, we arrange the sequence $a_1, a_2, \dots, a_{100}$ increasingly. We now can assume $a_1 \le a_2 \le \dots \le a_{100}$.

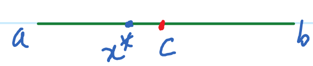
Then we compare 2 with a_{50}. If $2 > a_{50}$ then we only look for 2 in the sub-array $a_{51}, a_{52}, \dots, a_{100}$. If $2 < a_{50}$ then we only look for 2 in the sub-array $a_1, a_2, \dots, a_{49}$. Suppose the first case happens. Then we compare 2 with a_{75}. If $2 > a_{75}$ then we look for 2 in the sub-array $a_{76}, \dots, a_{100}$. Otherwise, we look for 2 in $a_{51}, \dots, a_{74}$. Continue until 2 is found or until the array has only one number. This method is called binary search. The complexity of this algorithm is of order $\log n = \log 100$.



Imagine that we are looking for the true root x^* of f on the interval $[a, b]$. Suppose $f(a)f(b) < 0$. By Intermediate Value Theorem, we know that there must be a root between a and b. We then consider the midpoint

$$c = \frac{a+b}{2}.$$

If $f(a)$ and $f(c)$ are of different sign then we know that there is a root between a and c. Now we regard c as the "new b". Then we continue the process.



If $f(b)$ and $f(c)$ are of different signs then we know that there is a root between c and b. We then regard c as the "new a". Then we continue the process.

Each time, the interval to search for a root is narrowed down by a half. We can summarize the process as an algorithm:

1) Start with an interval $[a_0, b_0]$ such that $f(a_0)$ and $f(b_0)$ have different signs.
2) Compute $c_0 = \frac{1}{2}(a_0 + b_0)$ and $f(c_0)$
3) If $f(a_0)$ and $f(c_0)$ have different signs then $a_1 = a_0$, $b_1 = c_0$.

If $f(b_0)$ and $f(c_0)$ have different signs then $a_1 = c_0$, $b_1 = b_0$.

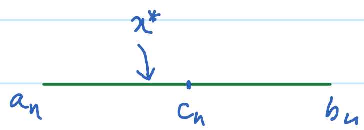
4) Continue: half the interval $[a_n, b_n]$ by the midpoint $c_n = \frac{1}{2}(a_n + b_n)$

Then compare the sign of $f(c_n)$ and the signs of $f(a_n)$ and $f(b_n)$

Then choose the interval $[a_{n+1}, b_{n+1}]$ accordingly.

[See practice on the worksheet.]

How close is c_n (midpoint of the n'th interval) to the true root?



The error $|x^* - c_n|$ is at most $\frac{1}{2}(b_n - a_n)$ because x^* lies in $[a_n, b_n]$.

Note that the length of the n'th interval is only a half of the length of the (n-1)'th interval.

Thus,

$$|c_n - x^*| \le \frac{1}{2}(b_n - a_n) = \frac{1}{2}\cdot\frac{1}{2}(b_{n-1} - a_{n-1})$$

$$= \frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}(b_{n-2} - a_{n-2})$$

$$= \cdots = \underbrace{\frac{1}{2}\cdot\frac{1}{2}\cdots\frac{1}{2}}_{n+1 \text{ factors}}(b_0 - a_0)$$

$$= \frac{1}{2^{n+1}}(b_0 - a_0).$$