Introduction: numerical methods

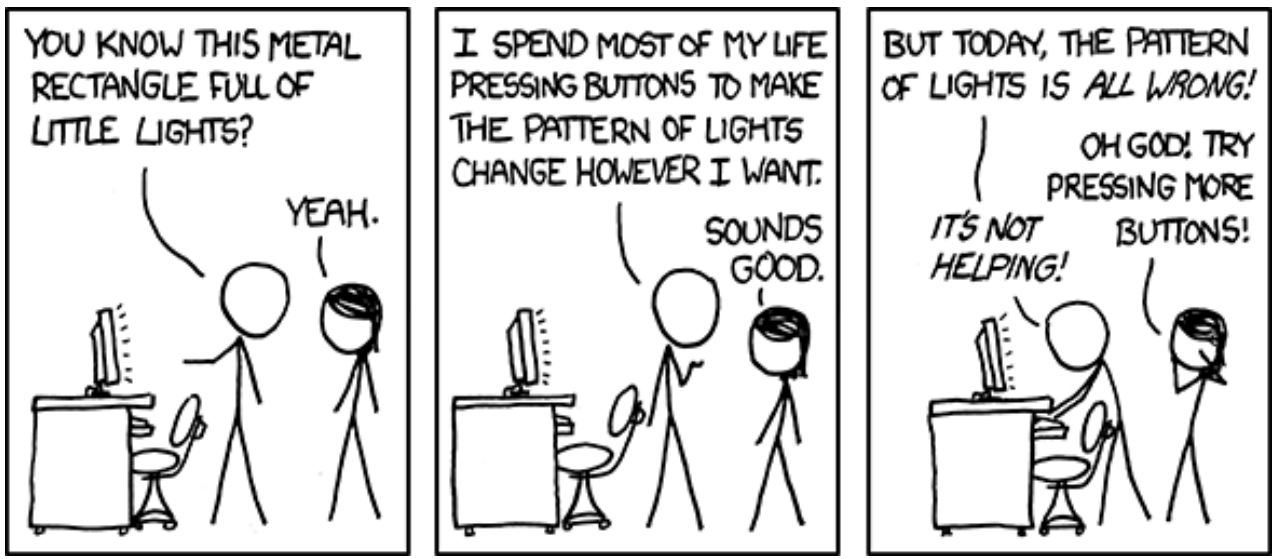
Michał Bejger
N. Copernicus Center, Warsaw

How computers compute?

Outline

- ⋆ Representation of numbers on the computer:
  - ⋆ integers,
  - ⋆ float point numbers,
- ⋆ Roundoff, truncation and aprroximation errors,

and related problems.



- ⋆ Storage in memory: finite number of bits
- ⋆ limitation on range and precision of numbers
- ⋆ Not representing the number precisely enough: roundoff error
- ⋆ information stored in binary representation: "strings" of states: 1's and 0's
- ⋆ One byte ≡ eight bits , variables typically stored in 32-bit or 64-bit *words*

Integers

Represented by a certain number of bits, e.g., $n_{bit} = 32$

- ⋆ in principle $2^{n_{bit}} - 1$ is the largest `unsigned integer`
- ⋆ How about a sign? One bit taken to denote sign:

0	00000000
1	00000001
2	00000010
3	00000011
-1	10000001
-2	10000010
-3	10000011

Will this work?
1 + (-1) = 00000001 + 10000001 = 10000010 = -2.
No good.

Integers

Represented by a certain number of bits, e.g., $n_{bit} = 32$

- ⋆ in principle $2^{n_{bit}} - 1$ is the largest `unsigned integer`
- ⋆ How about a sign? One bit taken to denote sign:

0	00000000
1	00000001
2	00000010
3	00000011

Two's complement representation - flip all the bits and add 1:

-1	11111111
-2	11111110
-3	11111101

Will this work?
1 + (-1) = 00000001 + 11111111 = 100000000 = 0.
All right! (note the overflow bit)

Range of signed integers: $-2^{n_{bit}-1} \leq I \leq 2^{n_{bit}-1} - 1$

Integer overflow

On a 32-bit machine:

```
import numpy as np

for n in range(30, 33):
    i = np.int32(2**n - 1)
    print n, i
```

Output:

```
30 1073741823
31 2147483647
32 -1
```

Compilers won't warn you against this ;-)

Floating point numbers

- ⋆ some fractions (e.g., 1/8) can be represented exactly, some other (e.g., 1/9) cannot,
- ⋆ irrational numbers (e.g., π) cannot be represented at all.

$\rightarrow$ floating point numbers are represented approximately.

```
import numpy as np

f = 0.123
print("%.20f" % np.float32(f))
print("%.20f" % np.float64(f))
```

Output:

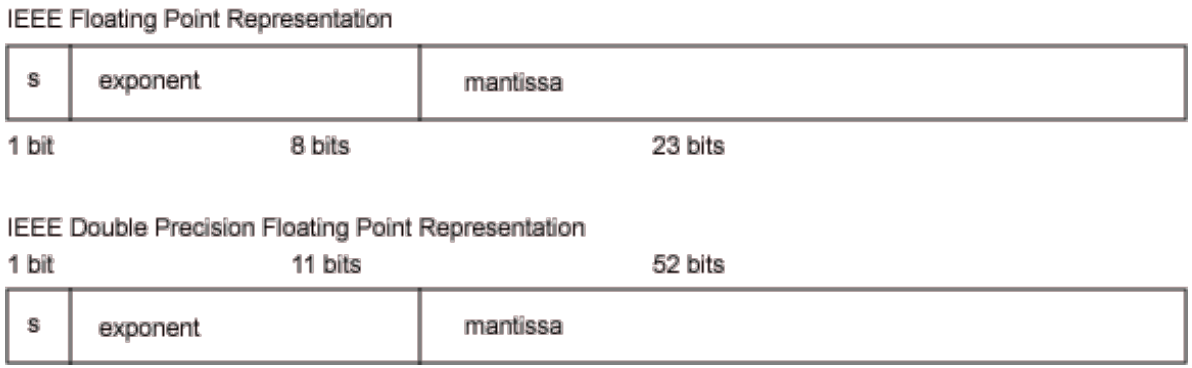
```
0.12300000339746475220
0.12299999999999999822
```

Floating point numbers

$$value = (sign\ value) \times mantissa \times 2^{(exponent\ -\ bias)}$$

	32-bit	64-bit
bits in sign	1	1
bits in exponent	8	11
bits in mantissa	23	52
bias*	127	1023

*In general, bias offset is $2^{(bits\ in\ exponent\ -\ 1)} - 1$ (the value of the exponent is offset from the actual value by the *exponent bias*)



	32-bit	64-bit
largest number	$\simeq 10^{38}$	$\simeq 10^{308}$
smalest number	$\simeq 10^{-38}$	$\simeq 10^{-308}$
precision	$\simeq 10^{-7}$	$\simeq 10^{-16}$

Floating point numbers: underflow & error growth

Precision, „how close the numbers can be represented". In case of 32-bit, mantissa is 23-bit, $2^{-23} \simeq 1.2 \times 10^{-7}$. For example:

```
import numpy as np

eps = 0.1
print "# eps                1 - (1 - eps)"
while (eps > 1e-10):
    print("%.8e %.8e" % (eps, np.float32(1.)-np.float32(1.-eps)))
    eps = eps/10.
```

Output:

```
# eps                1 - (1 - eps)
1.00000000e-01 1.00000024e-01
1.00000000e-02 9.99999046e-03
1.00000000e-03 9.99987125e-04
1.00000000e-04 1.00016594e-04
1.00000000e-05 1.00135803e-05
1.00000000e-06 1.01327896e-06
1.00000000e-07 1.19209290e-07
1.00000000e-08 0.00000000e+00
```

Comparing floating point numbers

Related caveat - direct comparison of floating numbers

```
import numpy as np

print("%.20f %.20f" % (np.float32(0.62/0.2), np.float32(3.1)))
print("%.20f" % np.float32(0.62/0.2 - 3.1))
```

Output:

```
3.09999990463256835938 3.09999990463256835938
-0.00000000000000044409
```

This may lead to accumulation of errors and problems with conditional statements:

```
a = 1.1; b = 2.2; c = 3.3

if a + b == c:
    print "yep!"
else:
    print "nope"
```

Output:

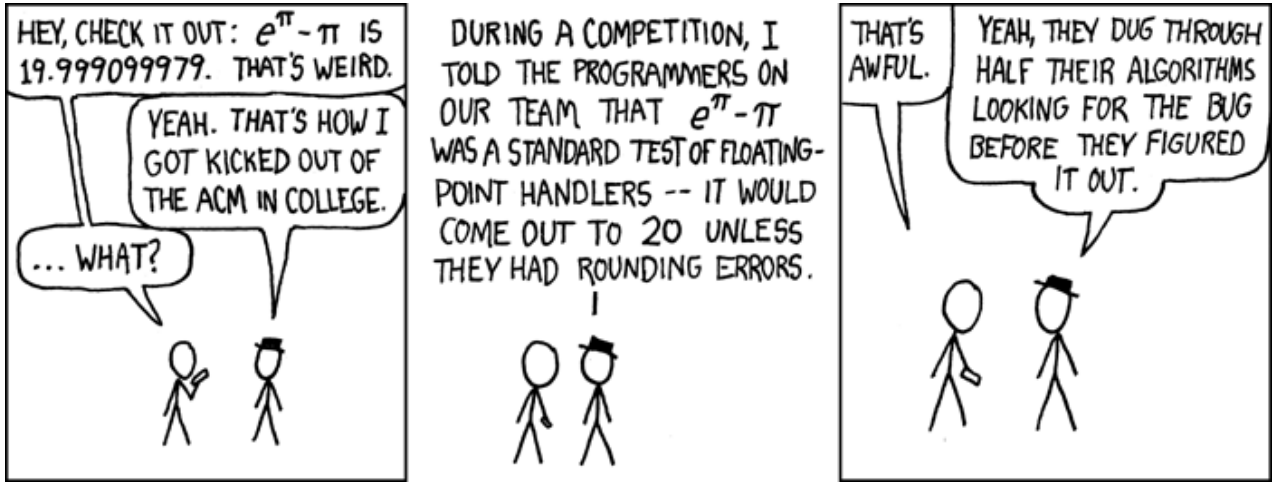
```
nope
```

Q. How many computer scientists does it take to change a lightbulb?

A. 1.99999999. One to change it and one to fix the rounding bug.

Q. How many computer scientists does it take to change 127 lightbulbs?

A. -128. 127 to change the bulbs and 1 to confirm there are no problems with the code.




```
import numpy as np

print("%.20f" % np.float32(np.e**np.pi - np.pi))
print("%.20f" % np.float64(np.e**np.pi - np.pi))
```

Output:

```
19.99909973144531250000
19.99909997918947013318
```

Numerical differentiation

On accuracy and precision

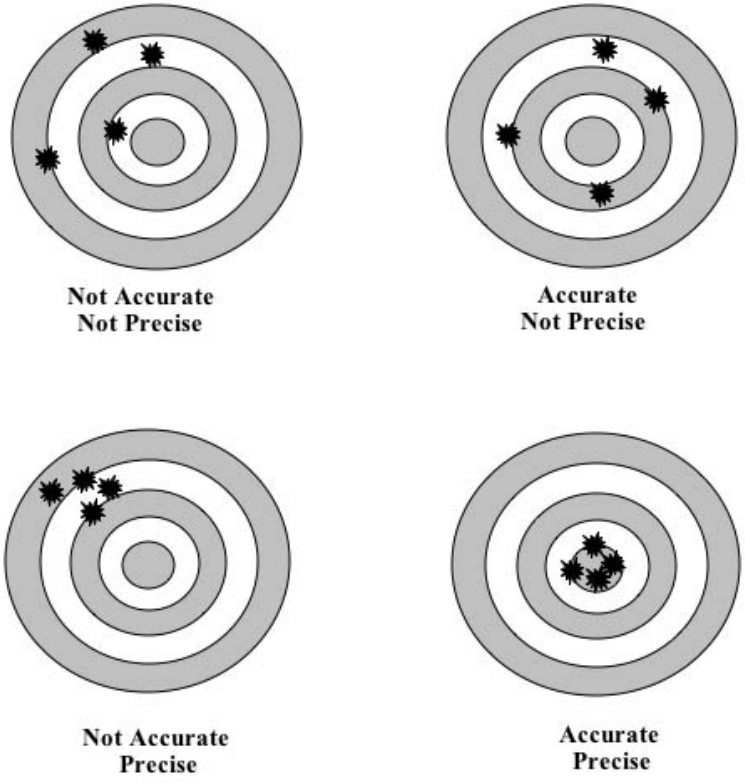
- Absolute vs relative errors,

$$\epsilon_{abs} = |\tilde{x} - x|$$

$$\epsilon_{rel} = \frac{|\tilde{x}-x|}{|x|}$$

  in numerics: because of roundoff & truncation errors

- Approximation errors (approximation of values *or* methods)



How to calculate the derivative?

Forward difference, from the definition:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Potential problems?

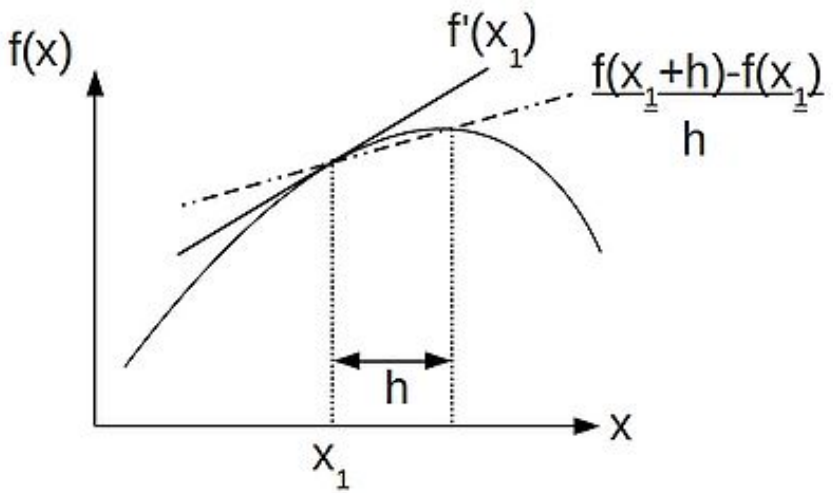
How to calculate the derivative?

Forward difference, from the definition:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Potential problems?

- $h \to 0$ 'difficult' to obtain on a computer (h small finite),
  - h too small: roundoff errors,
  - h too big: error from the approximation.
- assymetry.



Forward difference: error estimates

Expand our function f in Taylor series: $\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \dots$$

Approximation error is then

$$\delta_{appr} = \underbrace{\frac{f(x+h)-f(x)}{h}}_{\text{Forward difference}} - f'(x) = \underbrace{\frac{h}{2!}f''(x)}_{\text{Leading order}} + \frac{h^2}{3!}f'''(x) + \dots$$

Roundoff error. If ϵ is the *machine epsilon*,

$$f(x+h) - f(x) \propto \epsilon f(x), \quad \text{so that} \quad \delta_{roff} \propto \frac{\epsilon f(x)}{h}$$

Total error:

$$\delta = \mathcal{C}\epsilon\frac{f(x)}{h} + \frac{h}{2!}f''(x).$$

Case of double precision ($\epsilon \simeq 10^{-16}$), best δ is 10^{-8}

Centered difference

Slightly *different* definition:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h}$$

Expanding in Taylor series gives

$$f(x+\frac{h}{2}) - f(x-\frac{h}{2}) = f(x) + \frac{h}{2}f'(x) + \frac{1}{2!}\left(\frac{h}{2}\right)^2 f''(x) + \frac{1}{3!}\left(\frac{h}{2}\right)^3 f'''(x) + \dots$$

$$-f(x) + \frac{h}{2}f'(x) - \frac{1}{2!}\left(\frac{h}{2}\right)^2 f''(x) + \frac{1}{3!}\left(\frac{h}{2}\right)^3 f'''(x) + \dots$$

$$= hf'(x) + \frac{2}{3!}\left(\frac{h}{2}\right)^3 f'''(x) + \dots$$

Approximation error is then

$$\delta_{appr} = \underbrace{\frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h}}_{\text{Centered difference}} - f'(x) = \frac{h^2}{24} f'''(x) + \dots$$

(i.e., δ_{appr} is smaller for smaller h)

Centered difference improved (?)

Consider the following:

$$\frac{df}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

Expanding in Taylor series gives

$$f(x+h) - f(x-h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \dots$$

$$-f(x) + hf'(x) - \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \dots$$

$$= 2hf'(x) + \frac{2}{3!}h^3 f'''(x) + \dots$$

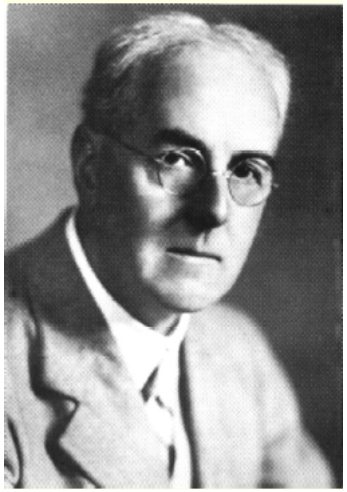
(h^3 term 8 times bigger than before)

Centered difference improved: Richardson error extrapolation

What's the difference?

$$8\left(f(x+\frac{h}{2})-f(x-\frac{h}{2})\right)-(f(x+h)-f(x-h))$$

$$=6hf'(x)-\frac{h^5}{80}f^{(5)}(x)+\ldots$$



Lewis Fry Richardson

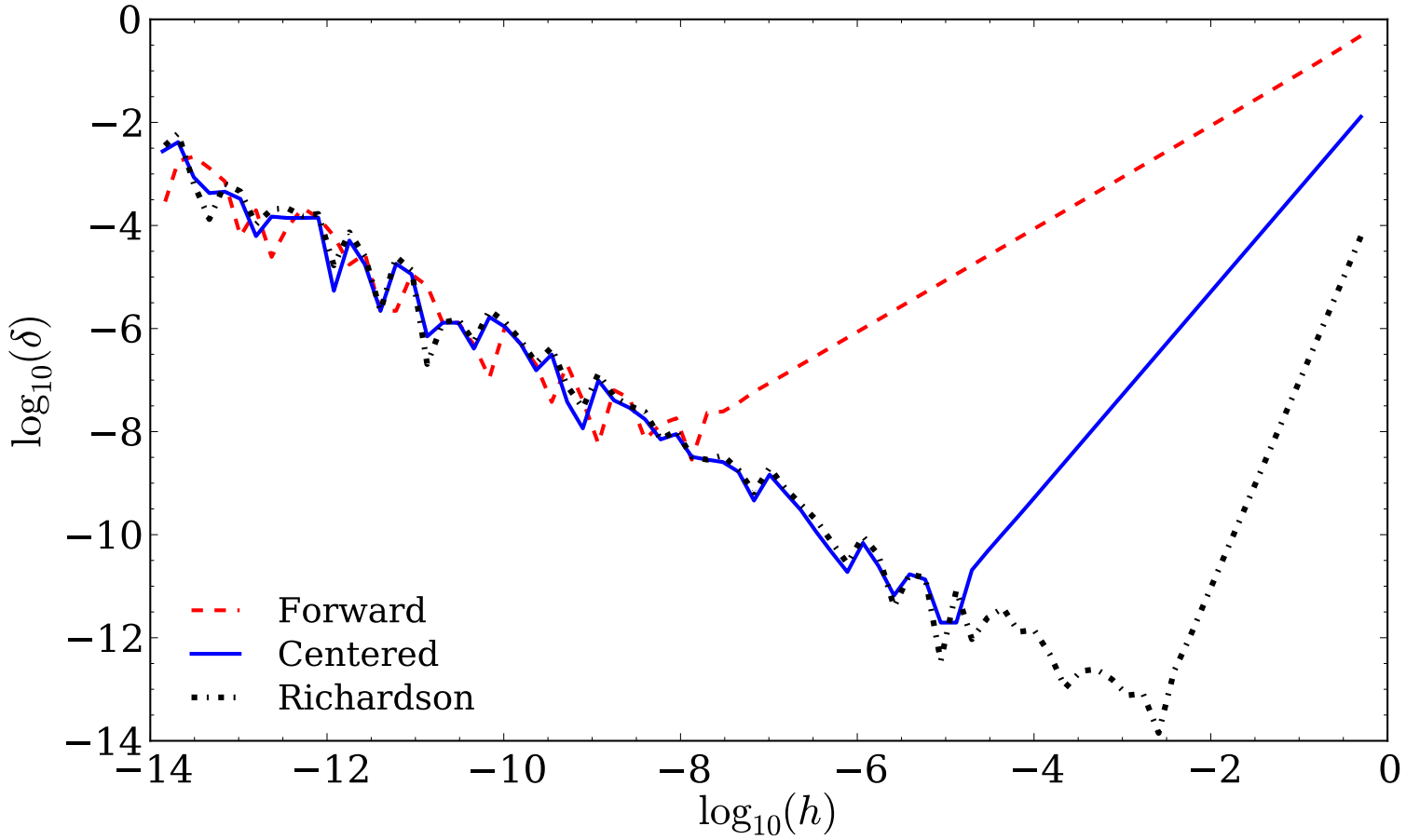
Approximation error is then

$$\delta_{appr}=\frac{1}{6h}\left(8\left(f(x+\frac{h}{2})-f(x-\frac{h}{2})\right)-(f(x+h)-f(x-h))\right)$$

$$=-\frac{h^4}{480}f^{(5)}(x)+\ldots$$

(i.e., error leading order h^4 makes the error even smaller)

Example: $f(x) = x^e$



Ordinary differential equations (ODEs)

- Radioactive decay $\frac{dN}{N} = -\lambda dt$,
- Newton's law of motion $m\frac{\mathrm{d}^2 x(t)}{\mathrm{d}t^2} = F(x(t))$,
- Nonlinear pendulum $L\frac{d^2\theta}{dx^2} = -g\sin\theta$,
- Harmonic oscillator $\frac{d^2u}{dx^2} + \omega^2 u = 0$,
- Growth/decay/oscillations $\frac{dy}{dx} + \alpha y = 0$
- ...

Initial value problem: $\frac{dy}{dt} + f(y,t) = 0, \quad y(t_0) = y_0$

Euler method

Let's consider $\frac{dy}{dt} + f(y,t) = 0$

Integrate:

$$\int_{t_n}^{t_{n+1}} \frac{dy}{d\tau} d\tau = y(t_{n+1}) - y(t_n) = -\int_{t_n}^{t_{n+1}} f\left(y(\tau), \tau\right) d\tau$$

For small $h = t_{n+1} - t_n$,

$$\int_{t_n}^{t_{n+1}} f(y(\tau),\tau) d\tau \approx f(y(t_n)), t_n) h \quad \rightarrow \quad y_{n+1} = y_n - f_n h$$

We have recovered the *forward difference*:

$$-f_n = \frac{y_{n+1} - y_n}{h} = \frac{dy}{dt}$$

(1st order accurate - $\mathcal{O}(h^2)$ terms truncated)

Euler method: stability

At some point the numerical solution deviates by δy from the solution of

$$y_{n+1} = y_n - f_n h$$

that is,

$$y_{n+1} + \delta y_{n+1} = y_n + \delta y_n - \left(f_n + \left.\frac{df}{dy}\right|_n \delta y_n + \ldots \right) h$$

Subtracting the two equations

$$\delta y_{n+1} = \underbrace{\left(1 - h \left.\frac{df}{dy}\right|_n \right)}_{S} \delta y_n$$

$\rightarrow$ method is stable if the *growth factor* $|S| < 1$

For example, $dy/dt + \alpha y = 0$

$$|S| = |1 - h\alpha| \quad \text{that is} \quad -1 < 1 - h\alpha < 1 \quad \rightarrow \quad \text{stable if } h < 2/\alpha$$

Backward Euler method

Let's consider again $\dfrac{dy}{dt} + f(y,t) = 0$

For small $h = t_{n+1} - t_n$,

$$\int_{t_n}^{t_{n+1}} f(y(\tau),\tau)d\tau \approx f(y(t_{n+1})), t_{n+1})h$$

$$\rightarrow y_{n+1} = y_n - f_{n+1}h$$

Again, assume the numerical solution deviates by δy from $y_{n+1} = y_n - f_{n+1}h$.

$$y_{n+1} + \delta y_{n+1} = y_n + \delta y_n - \left(f_{n+1} + \left.\frac{df}{dy}\right|_{n+1} \delta y_{n+1} + \ldots \right) h$$

$$\text{Subtracting, } \delta y_{n+1} = \underbrace{\left(1 + h \left.\frac{df}{dy}\right|_{n+1} \right)^{-1}}_{S} \delta y_n$$

For example, $dy/dt + \alpha y = 0$

$$-1 < 1/(1+h\alpha) < 1 \quad \rightarrow \quad h\alpha > 0 \quad \text{(stable for all } \alpha > 0)$$

Implicit vs explicit

Many ways to obtain the y_{n+1}:

- ⋆ Explicit methods example - forward Euler:

$$\frac{dy}{dt} = f(y,t) \quad \rightarrow \quad \frac{y_{n+1} - y_n}{h} = f(y_n, t_n)$$

$$y_{n+1} \text{ given by } y_{n+1} = f(y_n, t_n)h + y_n$$

- ⋆ Implicit methods example - backward Euler:

$$\frac{dy}{dt} = f(y,t) \quad \rightarrow \quad \frac{y_{n+1} - y_n}{h} = f(y_{n+1}, t_{n+1})$$

$$y_{n+1} \text{ from the solution of } y_{n+1} - f(y_{n+1}, t_{n+1})h = y_n$$

- ⋆ Crank-Nicholson = (forward Euler + backward Euler)/2

$$y_{n+1} = y_n + (f_n + f_{n+1})\,\frac{h}{2}$$

also known as *trapezoidal rule*.

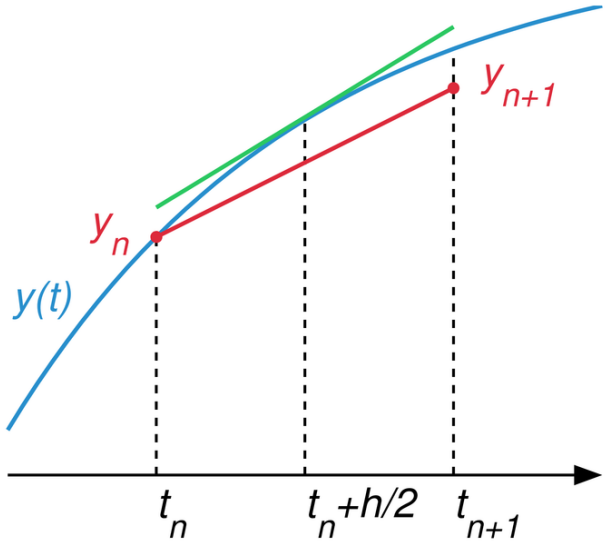
Midpoint method

Again, $\frac{dy}{dt} + f(y,t) = 0$

For small $h = t_{n+1} - t_n$,

$$\int_{t_n}^{t_{n+1}} f(y(\tau),\tau)d\tau \approx f(y(t_{n+1/2}), t_{n+1/2})h$$

$$\rightarrow y_{n+1} = y_n - f(y_n + \frac{h}{2}f(y_n,t_n), t_n + \frac{h}{2})h$$



- ⋆ Modification of the Euler method (accuracy $\mathcal{O}(h^2)$), also called 2nd order *Runge-Kutta* method,
- ⋆ Explicit & two-stage: second slope s_2 evaluated from first one, s_1:

$$s_1 = f(y_n, t_n), \quad s_2 = f\left(y_n + \frac{h}{2}s_1, t_n + \frac{h}{2}\right)$$

Predictor-corrector method

Let's take Crank-Nicholson

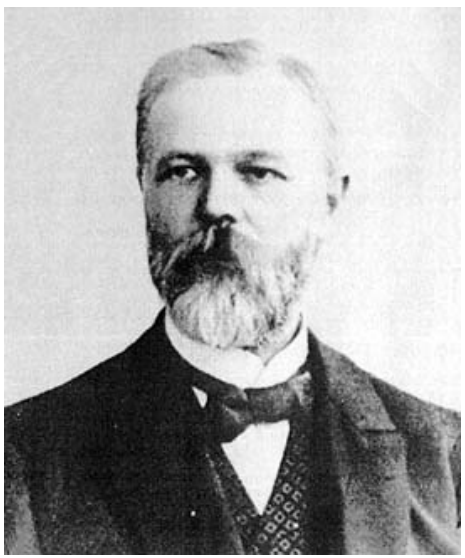
$$y_{n+1} = y_n + (f_n + f(y_{n+1}, t_{n+1}))\,\frac{h}{2}$$

and replace the *implicit* part in f_{n+1} with and *explicit* approximation by forward Euler, $y_{n+1} = y_n + hf(y_n, t_n)$:

$$y_{n+1} = y_n + (f_n + f(y_n + hf(y_n, t_n), t_{n+1}))\,\frac{h}{2}$$

This is called "Improved Euler" (Heun method).

- ⋆ predictor: Euler,
- ⋆ corrector: corrected implicit Crank-Nicholson.



Karl Heun

RK4 (4th order Runge-Kutta, *the* Runge-Kutta method)

With the initial value problem,

$$\frac{dy}{dt} - f(y,t) = 0, \quad y(t_0) = y_0$$

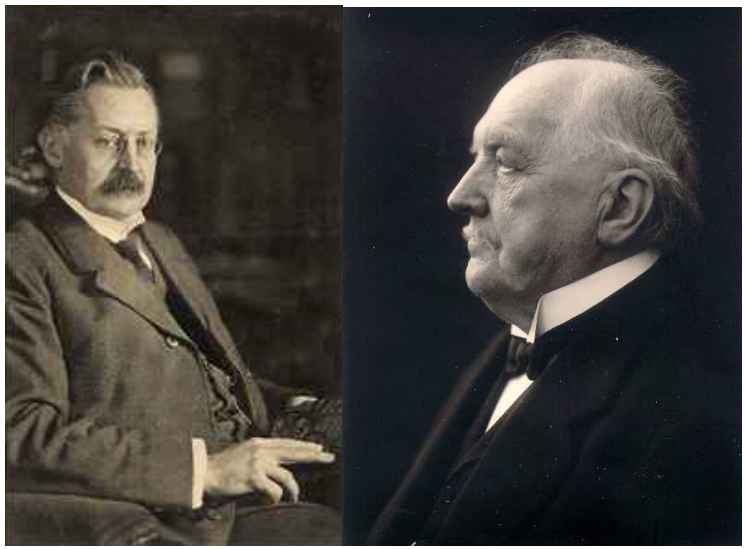
$$y_{n+1} = y_n - \tfrac{1}{6} h \left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

with

$$\begin{aligned} k_1 &= f(y_n, t_n), \\ k_2 &= f(y_n + \tfrac{h}{2} k_1, t_n + \tfrac{h}{2}), \\ k_3 &= f(y_n + \tfrac{h}{2} k_2, t_n + \tfrac{h}{2}), \\ k_4 &= f(y_n + h k_3, t_n + h). \end{aligned}$$

Weighted average of four increments, evaluated using the slope

- ⋆ at the beginning (k_1),
- ⋆ in the middle (k_2, k_3),
- ⋆ at the end (k_4).



Carl Runge & Martin Kutta

RK4 from the Simpson's rule

$$\int_{t_n}^{t_{n+1}} f(y,\tau)d\tau \approx \frac{1}{3}\frac{t_{n+1}-t_n}{2}\left(f_n + 4f_{n+1/2} + f_{n+1}\right)$$

$$= \frac{h}{6}\left(f_n + 4f_{n+1/2} + f_{n+1}\right)$$

→ Let's call $k_1 = f_n$,

First $2f_{n+1/2} = f\left(y(t_n + \frac{h}{2}), t_n + \frac{h}{2}\right)$:

approximate $y(t_n + \frac{h}{2}) \simeq y_n + \frac{h}{2}k_1$,

→ $k_2 = f(y_n + \frac{h}{2}k_1, t_n + \frac{h}{2})$,

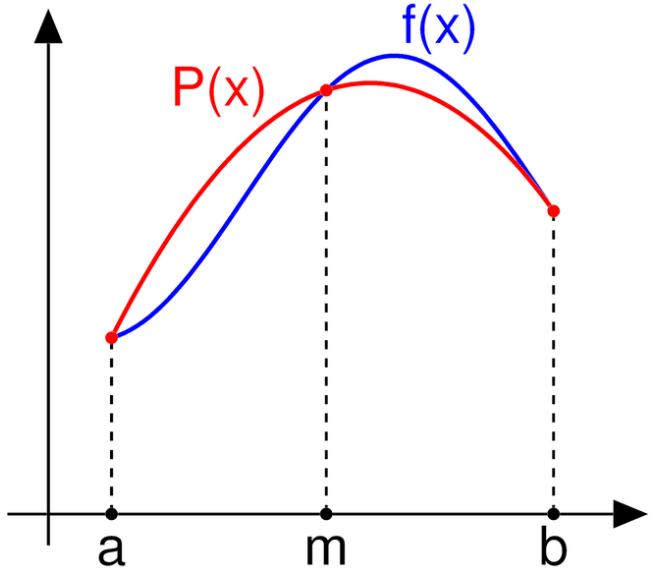
Second $2f_{n+1/2} = f\left(y(t_n + \frac{h}{2}), t_n + \frac{h}{2}\right)$:

approximate $y(t_n + \frac{h}{2}) \simeq y_n + \frac{h}{2}k_2$,

→ $k_3 = f(y_n + \frac{h}{2}k_2, t_n + \frac{h}{2})$,

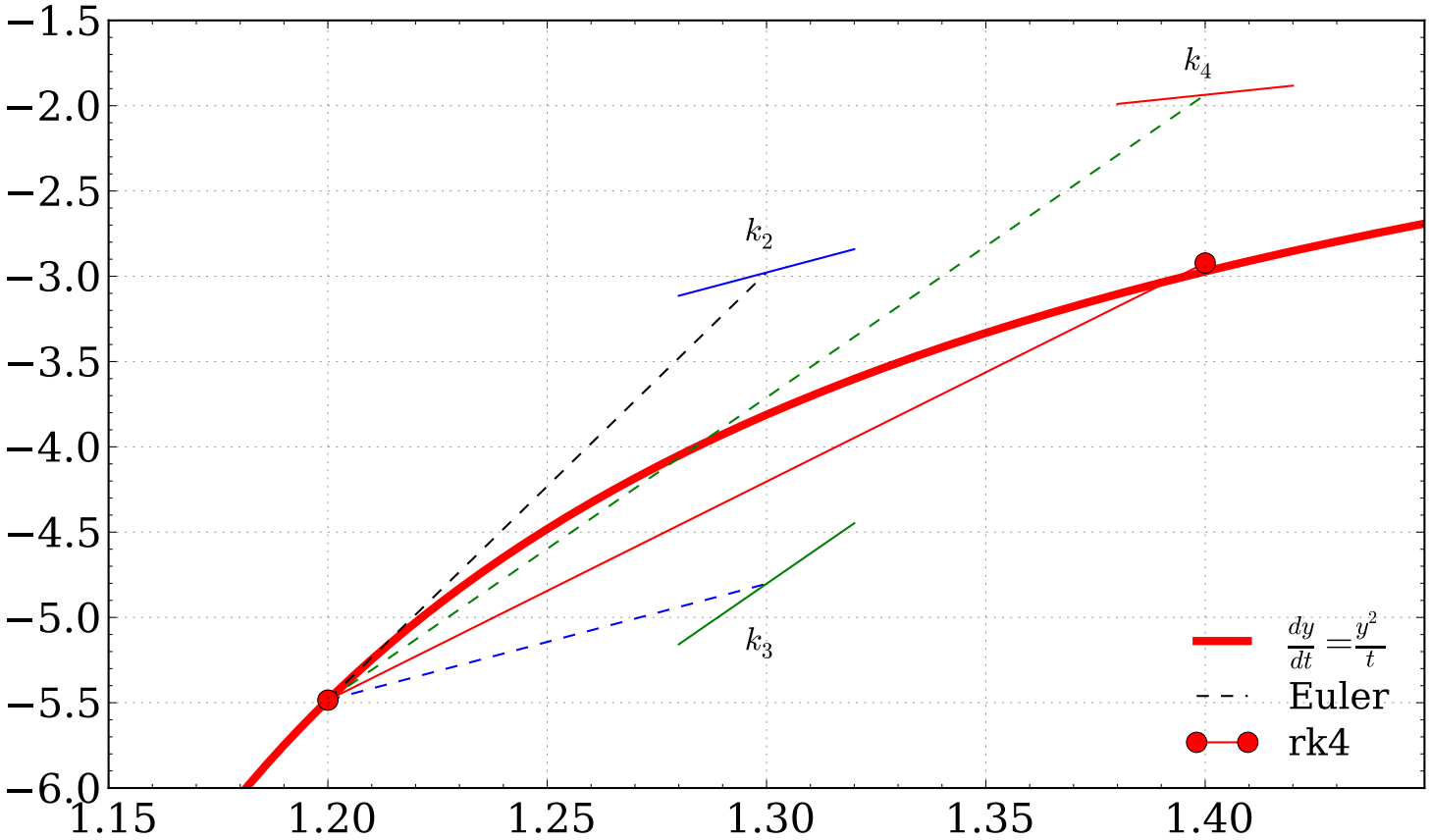
approximate $y(t_{n+1}) \simeq y_n + hk_3$

→ $k_4 = f(y_n + hk_3, t_{n+1})$



Approximating a given function by a parabola through beginning, middle and end points

Example: $\frac{dy}{dt} = y^2/t$



y_n = -5.48481494775, h = 0.2, k_1 = 25.0693291759, k_2 = 6.82137029658, k_3 = 17.7428578345, k_4 = 2.67788459275

y_{n+1} = -2.97201341199,

$y_{n+1,RK4}$ = -2.92229261339

Further reading...

- ⋆ Other methods:
  - ⋆ Adams-Bashforth, Adams-Moulton,
  - ⋆ Leap-frog, Verlet, symplectic methods (2nd order ODEs),
  - ⋆ Classification of RK methods (Butcher's tableau & group),
- ⋆ Sage (`http://www.sagemath.org`),
- ⋆ `numpy`, `scipy`, `matplotlib` manuals,
  - → Stack Overflow,
- ⋆ Fun from yesteryear:
  - → Jargon File & "The story of Mel, a Real Programmer"