Cviceni 4a

March 19, 2020

4 Jednoduché programy s jedním cyklem a několika proměnnými

V této třídě programů lze vyzkoušet spoustu dílčích problémů souvisejících s programováním jaké fyzik potká.

Nejprve se vrátíme k potížím s konečnou přesností reálných čísel. Na jedné straně jde o drobnost, na druhé pak ale musíme získat cit, pro omezení jež tato drobnost přináší

4.1 Iterativní výpočet obvodu ∞-úhelníka

Připomeňme si, náš poslední problém. Vzoreček založený na aproximaci kružnice pravidelným n-úhelníkem, u nějž poměr obvodu a (jednotkového) průměru dává aproximaci hodnoty π.

$$\pi_n = \frac{1}{2}O_n = n \sin \frac{2\pi}{2n}$$

Protože 2n-úhelník má poloviční úhly vrcholů oproti n-úhelníku a protože $\sin(x/2) = \sqrt{(1-\cos x)/2} = \sqrt{(1-\sqrt{1-\sin^2 x})/2}$ můžeme z hodnoty π_n spočíst π_{2n} jako

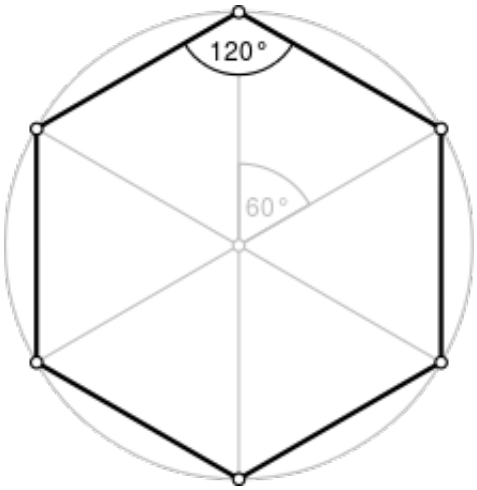
$$\pi_{2n} = 2n\sqrt{\frac{1-\sqrt{1-(\frac{\pi_n}{n})^2}}{2}}$$

Na základě toho vyjdeme-li z šestiúhelníka
$n = 6, \pi_6 = 3$ máme:

```
In [0]: import math
        n = 6
        pi_n = 3

        while n<1E9:
            print( n, pi_n, sep="\t")
            pi_n = 2*n*math.sqrt((1-math.sqrt(1-(pi_n/n)**2))/2)
            n = n*2
```

```
6       3
12      3.1058285412302498
24      3.132628613281237
48      3.139350203046872
96      3.14103195089053
```



alt text

```
192         3.1414524722853443
384         3.141557607911622
768         3.141583892148936
1536         3.1415904632367617
3072         3.1415921060430483
6144         3.1415925165881546
12288         3.1415926186407894
24576         3.1415926453212157
49152         3.1415926453212157
98304         3.1415926453212157
196608         3.1415926453212157
393216         3.141593669849427
786432         3.1415923038117377
1572864         3.1416086962248038
3145728         3.1415868396550413
6291456         3.1416742650217575
12582912         3.1416742650217575
25165824         3.1430727401700396
50331648         3.1598061649411346
100663296         3.181980515339464
201326592         3.3541019662496847
402653184         4.242640687119286
805306368         6.0
```

Vysvětlení:

Potíž spočívá ve velké nepřesnosti výpočtu rozdílu dvou blízkých čísel $1 - \sqrt{1 - (\frac{\pi_n}{n})^2}$. Pro velká n je $(\pi_n/n)^2$ velmi malé, takže hodnota odmocniny je velmi blízko jedné, takže ke konci dokonce vyjde $\pi \approx 6$.

Problém 1:

Vyjděte z úpravy

$$1-\sqrt{1-y}=\frac{(1-\sqrt{1-y})(1+\sqrt{1-y})}{1+\sqrt{1-y}}=\frac{1-(\sqrt{1-y})^2}{1+\sqrt{1-y}}=\frac{y}{1+\sqrt{1-y}}$$

a vyzkoušejte, že kód dává správný výsledek:

```
6       3
12      3.105828541230249
...
402653184     3.14159265358979
805306368     3.14159265358979
```

Můžete modifikovat kód výše nebo si napsat vlastní v **Pascalu**.

Reálná a `reálná` čísla

Vidíme, že důsledky odlišnosti mezi skutečnými reálnými čísly a těmi, jež máme jako jejich náhradu v běžných programovacích jazycích se může nečekaně projevit.

Místo reálného čísla se totiž používají **celá** čísla (tzv. mantisa a exponent) $m = -2^{53}..2^{53}, e = -970..970$ (zhruba, detaily vynechávám) a reálné číslo se aproximuje hodnotou

$$x \approx m.2^e$$

Lze si také představovat, že při výpočtech nemáme pod kontrolou hodnotu patnácté cifry za desetinnou čárkou při vyjádření čísla ve tvaru 4.45×10^{21}. Je zřejmé, že zde jsou k problémům náchylnější operace odečítání, které mohou z cifry daleko od první platné cifry učinit cifru významnější. Je ale přeci jen rozdíl mezi 2^e a 10^e, takže "v počítači" je i číslo 0.1 uloženo přibližně, protože jeho binární zápis je

$$(0.1)_d = (0.0001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ ...)_b$$

Zajímavost 1 Protože operace $x := x + \epsilon$ pro dostatečně malá ϵ nezmění hodnotu x, naivní pokus spočíst

$$\sum_{k=1}^{\infty} \frac{1}{k}$$

dá konečnou hodnotu, i když ve skutečnosti součet této řady diverguje. Je ale zřejmé, že chvíli bude trvat než při sčítání zaznamenáme, že se již výsledek proměnné určené k uložení hodnotu součtu nemění. Nedoporučuji tím ztrácet čas.

Zajímavost 2 Mohlo by vás zajímat, jak velké číslo lze ještě přičíst k jedničce, aby se tím tato nezměnila. Takové číslo lze snadno najít algortmem půlení intervalu, jak jej ještě budeme zkoušet.

```
In [0]: a = 0.0
        b = 1.0
        while b-a > 1E-12*b:
```

```
            c = (a+b)/2
            x = 1
            y = 1+c

            if y > x:
                # y se změnilo, tedy c je moc velké
                b = c
            else:
                # y se nezměnilo, tedy c je moc malé
                a = c

        print(a)
```

```
1.1102230246251565e-16
```

Zajímavost 3 Velká celá čísla také někdy znamenají potíž. V Pascalu jsme viděli, že hranicí za niž se nesmíme vydat je konstanta Maxint. Zde v Pythonu 3 tato nástraha (za cenu jistého zpomalení) odpadá.

Problém Porovnejte chování následujcích kódů

```
i   =  1
x   =  1.0

for n in range(0,310):
     print(n,':', x - i)
     i = i*10
     x = x*10

   a
```

```
{$R+}  // zkuste plus změnit na mínus
program notsobig;
var i:integer = 1;
    x:real    = 1;
    n:integer;
begin
    for n:=0 to 10 do begin
        Writeln(i,':',x - i);
        i := i*10;
        x := x*10;
    end;
end.
```

Mít možnost nebát se používat velká čísla (Python 3, Maple, Mathematica...) je pohodlná, ale jak uvidíme, někdy nám ztráta rychlosti, jíž za to platíme bude vadit.