

Informatika pro moderní fyziky (1) základy automatizace; jednoduché zpracování a vizualizace dat

František HAVLŮJ

e-mail: haf@ujv.cz

ÚJV Řež

oddělení Reaktorové fyziky a podpory palivového cyklu

semestr 2012/2013

4. prosince 2012

1 Úvod

- Syllabus semináře
- K čemu je počítač?
- Problém č. 1: vykreslování dat z detektoru
- Problém č. 2: mnoho výpočtů, inženýrova smrt

2 Ruční a poloautomatická řešení

- Problém č. 1: rozbor situace
- Řešení
- Zhodnocení
- Problém č. 2: jak na to?

3 Skriptovací jazyky

- Úvod do skriptování
- Úvod do jazyka Ruby
- První kroky s Ruby

4 Řešení problému č. 2

- Obecný rozbor
- Načítání výstupního souboru
- Generování vstupních souborů
- Zápis všech výsledků do tabulky

Obsah

1 Úvod

- Sylabus semináře
- K čemu je počítač?
- Problém č. 1: vykreslování dat z detektoru
- Problém č. 2: mnoho výpočtů, inženýrova smrt

2 Ruční a poloautomatická řešení

3 Skriptovací jazyky

4 Řešení problému č. 2





K čemu je počítač?

- počítače udělají cokoliv, pokud na to existuje postup
- pokud na něco existuje postup, není na to potřeba člověk
- existuje-li proces, existuje také algoritmus
- kdo má algoritmus, může napsat program

Proč se zabývat automatizací?

- mechanická práce je otravná
- program neudělá (náhodnou) chybu
- skript trvá stejně dlouho pro libovolný objem dat
- pokud je potřeba něco pozměnit nebo jen zpracování zopakovat, je ruční práce vepsí

Zadání

1

Na konci provozní směny je potřeba vyhodnotit signály ze čtyř detektorů a vykreslit je do grafu (signál v závislosti na čase). Data dostáváte v jednoduchém textovém souboru (dva sloupce, spousta řádků). Je potřeba vykreslit do jednoho grafu všechny čtyři detektory. Potíží je, že taková data přicházejí každý den - tento úkol je tedy potřeba řešit opakovaně.

S hvězdičkou

Počet detektorů je proměnný (1 až 9).

Zadání

2

Při přípravě základního kritického experimentu je pomocí MCNP potřeba najít kritickou polohu regulační tyče R2. Jak se tato poloha změní při změně polohy tyče R1?

Obsah

- 1 Úvod
- 2 **Ruční a poloautomatická řešení**
 - Problém č. 1: rozbor situace
 - Řešení
 - Zhodnocení
 - Problém č. 2: jak na to?
- 3 Skriptovací jazyky
- 4 Řešení problému č. 2

Klasické řešení (MS Excel)

- proved'te; jaké všechny kroky je potřeba udělat?
- na který z provedených kroků byl potřeba člověk – co z toho by nemohl stejně dobře udělat počítač sám?
- jaké jsou výhody a nevýhody ručního řešení?
- jak by mělo takové automatické řešení fungovat?

Komponenty pro automatizaci

Funkční části

výkonné programy (např. kreslení grafů, generování tabulek/reportů, spouštění výpočtů) – předpokladem je možnost spouštět program v neinteraktivním (dávkovém) režimu

Jak to slepit dohromady

dávkový soubor (BAT) nebo skript – je nutno vždy vhodně volit použité prostředky ve vztahu k jednoduchosti, požadavkům na funkce, přenositelnosti

Jak postupovat s automatickým řešením?

- 1 vykreslit graf s jedním detektorem

Jak postupovat s automatickým řešením?

- 1 vykreslit graf s jedním detektorem
- 2 se všemi detektory

Jak postupovat s automatickým řešením?

- 1 vykreslit graf s jedním detektorem
- 2 se všemi detektory
- 3 z příkazové řádky

Jak postupovat s automatickým řešením?

- 1 vykreslit graf s jedním detektorem
- 2 se všemi detektory
- 3 z příkazové řádky
- 4 z batch souboru

Jak postupovat s automatickým řešením?

- 1 vykreslit graf s jedním detektorem
- 2 se všemi detektory
- 3 z příkazové řádky
- 4 z batch souboru
- 5 se jménem adresáře jako parametrem

Gnuplot

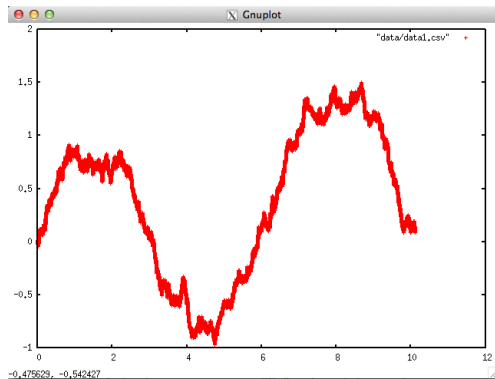
- interaktivní i dávkový režim – ideální pro automatizaci
- slušně konfigurovatelné 2D i 3D grafy
- i bez nastavení funguje velmi přijatelně
- široká paleta výstupních formátů

Vykreslení jednoho grafu v gnuplotu

```
gnuplot> plot "data1.csv"
```

Vykreslení jednoho grafu v gnuplotu

```
gnuplot> plot "data1.csv"
```

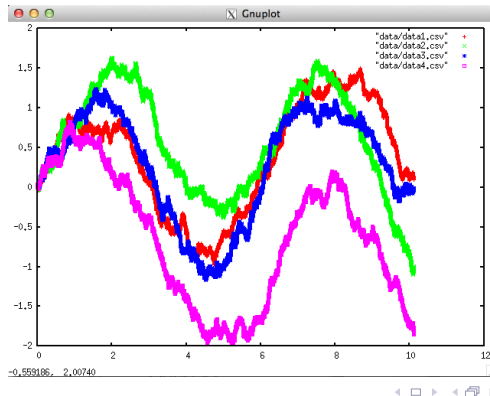


Vykreslení všech grafů v gnuplotu

```
gnuplot> plot "data1.csv", "data2.csv",  
             "data3.csv", "data4.csv"
```

Vykreslení všech grafů v gnuplotu

```
gnuplot> plot "data1.csv", "data2.csv",  
              "data3.csv", "data4.csv"
```

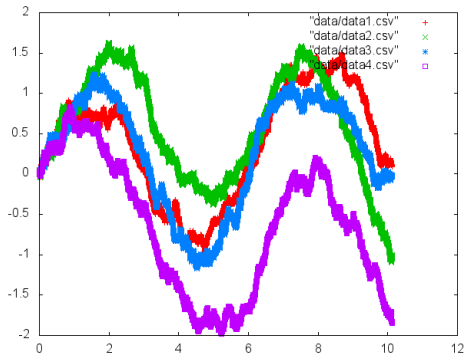


Dávkové použití

```
set terminal png
set output "plot4.png"
plot "data/data1.csv", "data/data2.csv", \
     "data/data3.csv", "data/data4.csv"
```

Dávkové použití

```
set terminal png
set output "plot4.png"
plot "data/data1.csv", "data/data2.csv", \
     "data/data3.csv", "data/data4.csv"
```



BAT soubor

Je pracné pokaždé vypisovat parametry na příkazovou řádku.
.BAT soubory ve Windows fungují jednoduše, prostě se do nich dá psát jako do terminálu a připravit si tak jednodušší skript.

```
gnuplot plot.gp
```

BAT soubor s parametrem

Co takhle adresář pro každý den? Nemá smysl pokaždé ručně kopírovat vstup pro gnuplot a tak dále...

Stačí vědět, že BAT soubor může mít na příkazové řádce parametry. První parametr je uložen do proměnné %1 a to se nám bude hodit.

```
cd %1  
gnuplot ../plot.gp  
cd ..
```

BAT soubor s parametrem - vylepšení

Pokud si budeme chtít prohlédnout grafy, bude nutné vždy vlézt do adresáře a otevřít `plot.png`. Jde to ovšem vylepšit pomocí jednoduchého triku:

```
cd %1
gnuplot ../plot.gp
copy plot.png ../%1.png
cd ..
```

Jak moc jsme si pomohli?



Stačí nám to na řešení problému č. 2?

Zatím nevíme, jak:

- vygenerovat vstupní soubory pro MCNP
- spustit hromadu MCNP výpočtů
- vytahat výsledky z MCNP výstupního souboru

Budeme potřebovat nějaký těžší kalibr.

Obsah

- 1 Úvod
- 2 Ruční a poloautomatická řešení
- 3 Skriptovací jazyky**
 - Úvod do skriptování
 - Úvod do jazyka Ruby
 - První kroky s Ruby
- 4 Řešení problému č. 2

“klasické” programování – Pascal, C++

- napsat zdroják, zkompilovat, slinkovat ...
- ... muset řešit binárku, která někde funguje a někde ne ...
- ... moc práce!
- (i když výhody jsou zřejmé)
- bylo by lepší mít někdy místo motorové pily sekeru

Interpretované jazyky / skripty

- textový vstupní soubor (zdrojový kód) + interpret
- vhodné pro aplikace bez vysokých nároků na systém
- nebo tam, kde je zásadní snížit nároky na vývoj
- tj. ideální pro jednoúčelové a krátkodobě žijící programy
- většinou “volnější” pojetí programování, z čehož plyne například řádově elegantnější práce s textem

Vlastnosti skriptovacích jazyků

Výhody

- dokonalá přenositelnost (textové vstupní soubory)
- nic se nekompiluje
- většinou syntakticky úsporné

Nevýhody

- zdrojový kód je otevřený (ne vždy se to hodí)
- pomalé a paměťově náročné
- bez kontroly správnosti při kompilaci

Přehled hlavních jazyků

- BAT** – vhodné pouze pro to nejjednodušší použití; i když jsou k mání některé trochu složitější funkce, jejich použití je hodně neobratné a neefektivní
- BASH** – podstatně mocnější alternativa BAT souborů v prostředí Unixu; nepříliš intuitivní syntaxe a absence náročnějších operací
- Perl** – kompaktní a efektní jazyk, který je všude nainstalovaný, ale nedá se (vůbec) číst a už i pole apod. jsou nekřesťansky obskurní
- Python** – velmi slušný jazyk, který snese i “vážnější” využití, ale za cenu trochu vyšší obtížnosti
- Ruby** – elixír síly a zázračná pilulka: intuitivní, snadný, všemocný, rozšířený a k tomu ryze objektový

Jazyk Ruby

- čistě objektový interpretovaný jazyk
- interprety existují pro širokou škálu platform
- velmi elegantní syntaxe
- nevýhodou je stále ještě relativní pomalost
- aktuální verze 1.9.3

Ukázka Ruby (1)

Každý programátor tím začíná ...

```
puts "Hello world!"
```

Ukázka Ruby (1)

Každý programátor tím začíná ...

```
puts "Hello world!"
```

```
Hello world!
```

Ukázka Ruby (2)

Proměnné, `print` vs. `puts`, aritmetika

```
a = 4  
b = 5  
print "4 + 5 = "  
puts a + b
```

Ukázka Ruby (2)

Proměnné, `print` vs. `puts`, aritmetika

```
a = 4  
b = 5  
print "4 + 5 = "  
puts a + b
```

```
4 + 5 = 9
```

Ukázka Ruby (3)

In-line výrazy v řetězcích

```
a = 4  
b = 5  
puts "#{a} + #{b} = #{a+b}"
```


Ukázka Ruby (3)

In-line výrazy v řetězcích

```
a = 4  
b = 5  
puts "#{a} + #{b} = #{a+b}"
```

```
4 + 5 = 9
```

Ukázka Ruby (4)

Rozsahy a cykly

```
(1..5).each do |i|  
  puts "#{i} * #{i} = #{i * i}"  
end
```

Ukázka Ruby (4)

Rozsahy a cykly

```
(1..5).each do |i|  
  puts "#{i} * #{i} = #{i * i}"  
end
```

```
1 * 1 = 1  
2 * 2 = 4  
3 * 3 = 9  
4 * 4 = 16  
5 * 5 = 25
```

Ukázka Ruby (4)

Pětkrát nic umožilo osla (opakování, ne cyklus)

```
5.times do  
  puts "nic"  
end
```

Ukázka Ruby (4)

Pětkrát nic umožilo osla (opakování, ne cyklus)

```
5.times do  
  puts "nic"  
end
```

```
nic  
nic  
nic  
nic  
nic
```

IRb

Pro první ozkoušení (a i pro některé úkoly v praktickém životě) se hodí “příkazová řádka” Ruby, tzv. **Interactive Ruby** (IRb):

IRb

Pro první ozkoušení (a i pro některé úkoly v praktickém životě) se hodí “příkazová řádka” Ruby, tzv. **Interactive Ruby** (IRb):

```
1.9.2-p290 :001 > 2+2  
=> 4  
1.9.2-p290 :002 > a = 5  
=> 5  
1.9.2-p290 :003 > b = 6  
=> 6  
1.9.2-p290 :004 > a * b  
=> 30
```

Proměnné, výpis na terminál

V Ruby (jak je u skriptů zvykem) se proměnné nedeklarují:

```
a = 5  
a = a * a  
long_string = "loooooong string"
```


Proměnné, výpis na terminál

V Ruby (jak je u skriptů zvykem) se proměnné nedeklarují:

```
a = 5  
a = a * a  
long_string = "loooooong string"
```

Výpis se děje pomocí print, resp. puts (bez/s koncem řádku);
#{...} vkládá do řetězce libovolný výraz:

```
print a  
puts "a = #{a}"
```

Pole a hashe

Pole je seznam:

```
a = []  
a << 5  
a += [6]  
puts a.size
```

Pole a hashe

Pole je seznam:

```
a = []  
a << 5  
a += [6]  
puts a.size
```

Hash, neboli slovník či asociativní pole:

```
b = {}  
b[3] = 7  
b["foo"] = "bar"
```

Rozsahy a cykly

Rozsahy (ranges) - se dvěma tečkami včetně posledního elementu, se třemi bez něj

```
a = (1..5)
b = (1...5)
puts "yay!" if a.size == b.size + 1
```

Rozsahy a cykly

Rozsahy (ranges) - se dvěma tečkami včetně posledního elementu, se třemi bez něj

```
a = (1..5)
b = (1...5)
puts "yay!" if a.size == b.size + 1
```

Ruby nepoužívá klasický cyklus, ale iterátor (přes téměř cokoliv):

```
(1..5).each do |i|
  puts i * i
end
b = {}; b["key1"] = 6; b["key2"] = 8
b.each do |key, value|
  puts "#{key} => #{value}"
end
```

Práce s řetězci, include, split, sub

Řetězce v Ruby jsou neomezené délky (pár mega se tam určitě vejde) a dá se s nimi provádět ledacos.

```
s = "lazy dog"
if s.include?("lazy")
  puts "lazy!!!"
end
```

Práce s řetězci, include, split, sub

Řetězce v Ruby jsou neomezené délky (pár mega se tam určitě vejde) a dá se s nimi provádět ledacos.

```
s = "lazy dog"
if s.include?("lazy")
  puts "lazy!!!"
end
```

Rozdělit? Nahradit?

```
puts s.sub("lazy", "crazy")
a = s.split
puts "#{a[1]} #{a[0]}"
```

Načítání a zápis do souboru

Soubor a terminál, to je vlastně jedno:

```
File.open("animals.txt", "w") do |f|  
  f.puts "quick brown fox"  
end
```


Načítání a zápis do souboru

Soubor a terminál, to je vlastně jedno:

```
File.open("animals.txt", "w") do |f|  
  f.puts "quick brown fox"  
end
```

Nejjednodušší čtení je po řádcích:

```
IO.foreach("data.csv") do |line|  
  ...  
end
```

Práce s adresářem

Jak projít všechny soubory v adresáři? V Pascalu utrpení, v Ruby iterátor:

```
Dir["*"].each do |filename|  
  puts filename  
end
```

Práce s adresářem

Jak projít všechny soubory v adresáři? V Pascalu utrpení, v Ruby iterátor:

```
Dir["*"].each do |filename|  
  puts filename  
end
```

Lze použít podle očekávání libovolnou masku nebo cestu:

```
Dir["data/*.csv"].each do |filename|  
  IO.foreach(filename) do |line|  
    puts line  
  end  
end
```

Obsah

- 1 Úvod
- 2 Ruční a poloautomatická řešení
- 3 Skriptovací jazyky
- 4 Řešení problému č. 2**
 - Obecný rozbor
 - Načítání výstupního souboru
 - Generování vstupních souborů
 - Zápis všech výsledků do tabulky

Určení poloh tyčí

```
c -----  
c polohy tyci (z-plochy)  
c -----  
c  
67 pz 47.6000      $ dolni hranice absoberu r1  
68 pz 40.4980      $ dolni hranice hlavice r1  
69 pz 44.8000      $ dolni hranice absoberu r2  
70 pz 37.6980      $ dolni hranice hlavice r2
```

A to je vše, přátelé!

