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

Obsah

Profil absolventa předmětu

- chápe počítač nikoli jako psací stroj anebo černou skříňku pro specializované aplikace, ale především jako flexibilní a vysoce univerzální nástroj pro každodenní úkoly ve zpracování dat, jejich prezentaci a tvorbě dokumentů
- orientuje se v moderních paradigmatech praktické informatiky, programování na úrovni běžných skriptů je pro něj samozřejmostí a díky solidnímu přehledu je schopen se v daném problému zorientovat a vybrat si pro jeho řešení vhodný nástroj
- není nucen vykonávat mechanickou a nudnou činnost, ale úkoly řeší kreativně - raději si ad hoc vytvoří na míru šitý skript, který jej ochrání před lidskou chybou i frustrací z monotónnosti

Průběh výuky

- zadání praktické úlohy, jejíž vyřešení je motivací pro obsah lekce
- přednáška na probírané téma, poskytující jak teoretický základ, tak přehled konkrétních nástrojů a postupů
- samostatná práce na řešení daného problému
- společná diskuse nad jednotlivými řešeními a jejich zhodnocení

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. Potíž 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

Vstupní data

```
0.00000e+00 0.00000e+00
1.00000e-04 1.01447e-03
2.00000e-04 4.62446e-04
3.00000e-04 6.92465e-04
4.00000e-04 4.48142e-03
5.00000e-04 6.95896e-03
6.00000e-04 5.12501e-03
7.00000e-04 2.62076e-03
...
```

Formát CSV

- comma separated values
- zobecnělo ale jako libovolný formát po sloupcích uložených dat
- dobře se zpracovává, importuje do Excelu atd.

Klasické řešení (MS Excel)

- proveďte; 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?

Co by takové automatické řešení mohlo umět?

- načte z daného adresáře soubory se záznamy
- vykreslí graf a uloží ho do souboru
- soubor jednoznačně pojmenuje a zkopíruje na vhodné místo
- uživatel by neměl v ideálním případě dělat vůbec nic

Komponenty pro automatizaci

Funkční části

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

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

dávkobý 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?

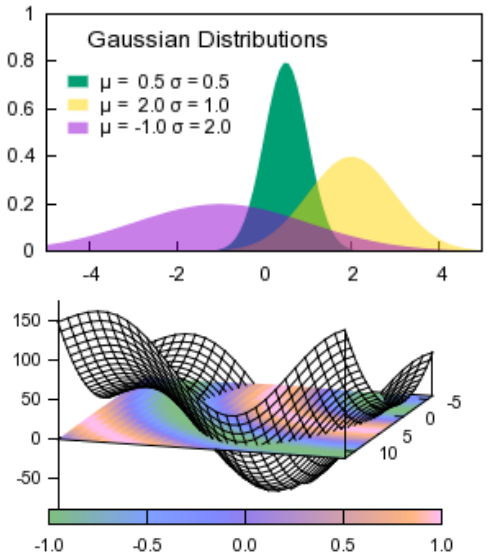
1. vykreslit graf s jedním detektorem
2. se všemi detektory
3. z příkazové řádky
4. z batch souboru

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

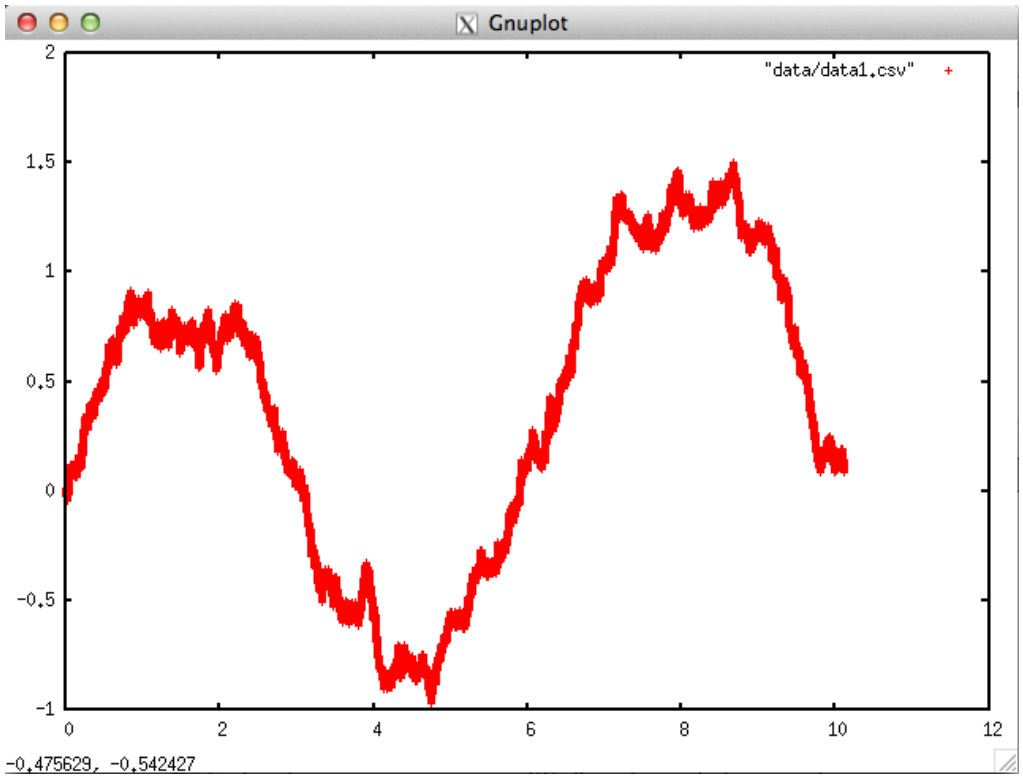


Vykreslení jednoho grafu v gnuplotu

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

Vykreslení jednoho grafu v gnuplotu

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

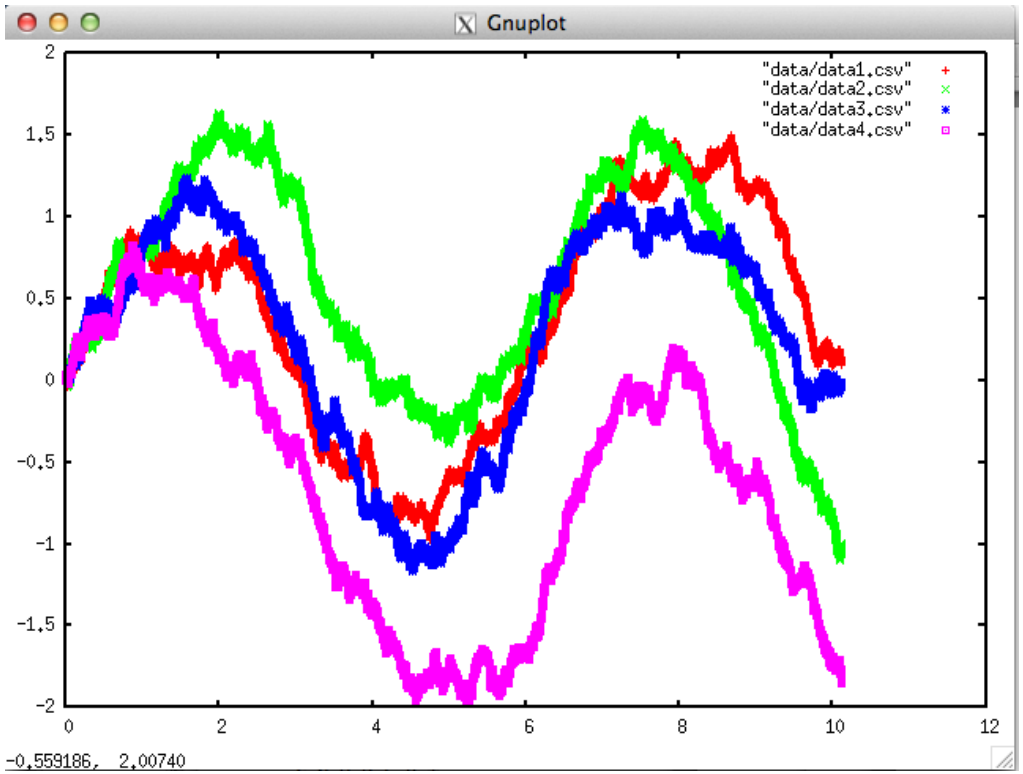


Vykreslení všech grafů v gnuplotu

```
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"
```

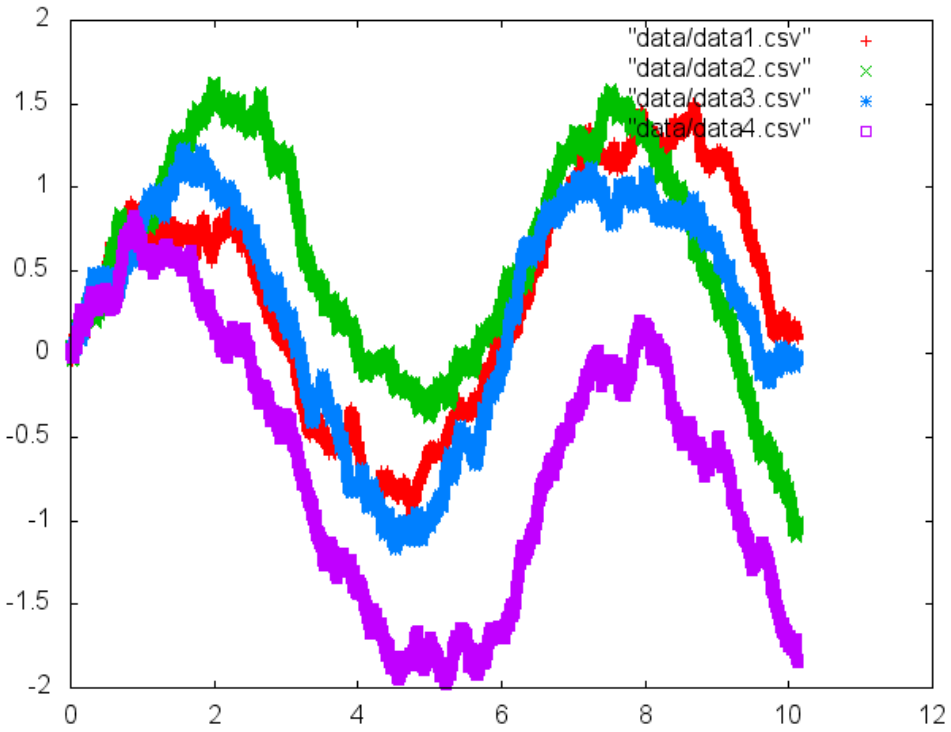


Dávkové použití

```
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?

- jeden skript místo excelovské anabáze
- máme znovupoužitelný nástroj – můžeme proces kdykoliv zopakovat
- nelze udělat žádnou “ruční chybu”
- skript můžeme dát kolegovi a ten má práci hotovou úplně zadarmo
- skript lze periodicky spouštět bez účasti uživatele, možnost např. zobrazovat na intranetu aktuální grafy atd.

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

“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

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 platforem
- 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 umořilo osla (opakování, ne cyklus)

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

Ukázka Ruby (4)

Pětkrát nic umořilo 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 = "looooong string"
```

Proměnné, výpis na terminál

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

```
a = 5
a = a * a
long_string = "looooong 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

Problém č. 2: mnoho výpočtů, inženýrova smrt

Zadání

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?

Co máme k dispozici?

MCNP

Pokud připravíme vstupní soubor (v netriviální formě obsahující polohy regulačních tyčí R1 a R2), spočítá nám keff.

Potřebovali bychom ale něco na:

1. vytvoření velkého množství vstupních souborů
2. extrakci keff z výstupních souborů
3. popřípadě na vyhodnocení získaných poloh tyčí a keff

Pracovní postup

1. načíst keff z výstupního souboru MCNP

Pracovní postup

1. načíst keff z výstupního souboru MCNP
2. vygenerovat potřebné vstupní soubory

Pracovní postup

1. načíst keff z výstupního souboru MCNP
2. vygenerovat potřebné vstupní soubory
3. vyrobit BAT soubor na spuštění výpočtů

Pracovní postup

1. načíst keff z výstupního souboru MCNP
2. vygenerovat potřebné vstupní soubory
3. vyrobit BAT soubor na spuštění výpočtů
4. načíst výsledky ze všech výstupních souborů do jedné tabulky

Pracovní postup

1. načíst keff z výstupního souboru MCNP
2. vygenerovat potřebné vstupní soubory
3. vyrobit BAT soubor na spuštění výpočtů
4. načíst výsledky ze všech výstupních souborů do jedné tabulky
5. buď zpracovat ručně (Excel), nebo být Myšpulín a vyrobit skript (úkol s hvězdičkou)

Nejprve najdeme, kde je ve výstupu z MCNP žádané keff:

```
.....
         the k(trk length) cycle values appear normally distributed at the 95 percent confide

------------------------------------------------------------------------------------------
|
| the final estimated combined collision/absorption/track-length keff = 1.00353 with an estima
|
| the estimated 68, 95, & 99 percent keff confidence intervals are 1.00329 to 1.00377, 1.0030
|
| the final combined (col/abs/tl) prompt removal lifetime = 1.0017E-04 seconds with an estimat
.....
```

Algoritmus

1. najít řádek s keff

Algoritmus

1. najít řádek s keff
2. vytáhnout z něj keff, takže například:

Algoritmus

1. najít řádek s keff
2. vytáhnout z něj keff, takže například:
3. rozdělit podle rovnítka

Algoritmus

1. najít řádek s keff
2. vytáhnout z něj keff, takže například:
3. rozdělit podle rovnítka
4. druhou část rozdělit podle mezer

Algoritmus

1. najít řádek s keff
2. vytáhnout z něj keff, takže například:
3. rozdělit podle rovnítka
4. druhou část rozdělit podle mezer
5. vzít první prvek

Realizace (1/5)

```
keff = nil

IO.foreach("c1_1o") do |line|

end

puts keff
```

Realizace (2/5)

```
keff = nil

IO.foreach("c1_1o") do |line|
  if line.include?("final estimated combined")

  end
end

puts keff
```

Realizace (3/5)

```
keff = nil

IO.foreach("c1_1o") do |line|
  if line.include?("final estimated combined")
    a = line.split("=")

  end
end

puts keff
```

Realizace (4/5)

```
keff = nil

IO.foreach("c1_1o") do |line|
  if line.include?("final estimated combined")
    a = line.split("=")
    b = a[1].split

  end
end

puts keff
```

Realizace (5/5)

```
keff = nil

IO.foreach("c1_1o") do |line|
  if line.include?("final estimated combined")
    a = line.split("=")
    b = a[1].split
    keff = b[0]
  end
end

puts keff
```

Určení poloh tyčí

Ve vstupním souboru si najdeme relevantní část:

```
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
```

Výroba šablon

Jak dostat polohy tyčí do vstupního souboru? Vyrobíme šablonu, tzn nahradíme

```
67 pz 47.6000     $ dolni hranice absoberu r1
```

Výroba šablon

Jak dostat polohy tyčí do vstupního souboru? Vyrobíme šablonu, tzn nahradíme

```
67 pz 47.6000     $ dolni hranice absoberu r1
```

nějakou značkou (*placeholder*):

```
67 pz %r1%     $ dolni hranice absoberu r1
```

Chytáky a zádrhele

- kromě samotné plochy konce absorbéru je nutno správně umístit i z-plochu konce hlavice o 7,102 cm níže
- obecně je na místě ohlídat si, že placeholder nebude kolidovat s ničím jiným

Doporučené nástroje jsou:

- již známá funkce `sub` pro nahrazení jednoho řetězce jiným
- pro pragmatické lenochy funkce `IO.read` načítající celý soubor do řetězce (na což nelze v Pascalu ani pomyslet)
- možno ovšem použít i `IO.readlines` (v čem je to lepší?)

Realizace

```
DELTA = 44.8000 - 37.6980

template = IO.read('template')
(0..10).each do |i1|
  (0..10).each do |i2|
    r1 = i1 * 50
    r2 = i2 * 50
    File.open("inputs/c_#{i1}_#{i2}", "w") do |f|
      s = template.sub("%r1%", r1.to_s)
      s = s.sub("%r1_%", (r1 - DELTA).to_s)
      s = s.sub("%r2%", r2.to_s)
      s = s.sub("%r2_%", (r2 - DELTA).to_s)
      f.puts template
    end
  end
end
```

Jak na to

Máme všechno, co potřebujeme:

- načtení keff z jednoho výstupního souboru (`IO.foreach`, `include` a `split`)
- procházení adresáře (`Dir.each`)
- zápis do souboru (`File.open` s parametrem `w`)

Takže už to stačí jen vhodným způsobem spojit dohromady!

Realizace

```
Dir["*o"].each do |filename|
  keff = nil

  IO.foreach(filename) do |line|
    if line.include?("final estimated combined")
      a = line.split("=")
      b = a[1].split
      keff = b[0]
    end
  end

  puts "#{filename} #{keff}"
end
```

Výstup

Výsledkem je perfektní tabulka:

```
outputs/c_0_0o 0.94800
outputs/c_0_10o 0.99800
outputs/c_0_1o 0.94850
outputs/c_0_2o 0.95000
outputs/c_0_3o 0.95250
outputs/c_0_4o 0.95600
...
```

Hloupé je, že nikde nemáme tu polohu tyčí.

Chytrá horákyně

... by jistě vyrobila toto:

```
0 0 0.94800
0 10 0.99800
0 1 0.94850
0 2 0.95000
0 3 0.95250
0 4 0.95600
...
```

Nápovědou je funkce `split` (podle podtržítka) a funkce `to_i` (co asi dělá?)

Realizace chytré horákyně

```
Dir["outputs/*o"].each do |filename|
  keff = nil

  IO.foreach(filename) do |line|
    if line.include?("final estimated combined")
      a = line.split("=")
      b = a[1].split
      keff = b[0]
    end
  end

  s = filename.split("_")
  puts "#{s[1].to_i} #{s[2].to_i} #{keff}"
end
```

A to je vše, přátelé!