

# Ruby/GSL

常定 芳基

Yoshiki Tsunesada

東京工業大学

第1回 電脳 Ruby ワークショップ

15.Mar.2002

# Ruby/GSLとは？

- GSLのための Ruby インターフェース

## GSLとは？

- GNU Scientific Library
- Cにおける数値計算のためのライブラリ  
featuring
  - ◆ ベクトル、行列
  - ◆ 線形代数計算、固有値問題
  - ◆ 数値積分
  - ◆ 特殊関数
  - ◆ FFT (高速フーリエ変換)
  - ◆ 乱数生成, .....

# なぜGSL?

- C である
- 基本的データ構造が用意されている  
(as `gsl_vector`, `gsl_matrix`, ...)
- フリーである

## 他のC用数値計算ライブラリ

- NRC (Numerical Recipes in C)
  - ベクトル・行列をうまく扱う  
`double* vector`, `double** matrix`
  - フリーではない。
  - 本が出ていて詳しい。

# GSLプログラミング in C

プログラム作成

コンパイル、リンク

実行

- メモリ確保 (構造体生成)
  - データ入出力
  - 計算
  - データ入出力
  - メモリ開放
- ```
% gcc -o xxx x.c -lgsl -lgslcblas -lm % ./xxx
```

## 例: 連立方程式 $Ax = b$ を解く

```
#include <stdio.h>
#include <gsl/gsl_linalg.h>
int main ()
{
    double a_data[] = { 0.18, 0.60, 0.57, 0.96,
                       0.41, 0.24, 0.99, 0.58,
                       0.14, 0.30, 0.97, 0.66,
                       0.51, 0.13, 0.19, 0.85 };
    double b_data[] = { 1.0, 2.0, 3.0, 4.0 };
    gsl_vector *x = gsl_vector_alloc (4);
    gsl_matrix_view m = gsl_matrix_view_array(a_data, 4, 4);
    gsl_vector_view b = gsl_vector_view_array(b_data, 4);
    int s; gsl_permutation * p = gsl_permutation_alloc (4);

    gsl_linalg_LU_decomp (&m.matrix, p, &s);
    gsl_linalg_LU_solve (&m.matrix, p, &b.vector, x);
    gsl_vector_fprintf(stdout, x, "%g");
    gsl_permutation_free (p);
    return 0;
}
```

- たったこれだけで連立方程式が解けてしまう！  
すごいぞ GSL!
- やけにまわりくどい、タイプ量が多い、  
コンパイルが面倒、、、

行列

右辺のベクトル

解ベクトル

作業領域

LU分解

方程式を解く

結果出力: ベクトル  $x$

メモリ開放

そしてコンパイル、、、

# 楽をするには？

- 構造体メモリの確保
- データの与え方
- 作業領域の確保
- メモリ開放

簡単にしたい  
メモリ確保とデータ初期化を同時に

やりたくない

やりたくない

# Ruby/GSLの基本概念

- GSLの構造体 Ruby オブジェクトにラップ  
(対応するクラスを定義)
- GSLの構造体に対して何かする関数  
オブジェクトのメソッドとして実装
- メモリ確保、構造体生成  
new メソッドなどによってオブジェクト生成  
使い慣れたクラスから .to\_XXX
- メモリ開放

Ruby GC!

面倒なことはコンピュータがやれ  
GSL を Ruby の哲学に基づいて再構成

# C/GSL と Ruby/GSL

## C/GSL

Cプログラム作成

- メモリ確保 (構造体生成)
- データ入出力
- 計算 (関数)
- メモリ開放

コンパイル、リンク

```
% gcc -o xxx x.c -lgsl -lgslcblas -lm % ./xxx
```

実行

## Ruby/GSL

Rubyプログラム作成

- オブジェクト生成
- データ入出力
- 計算 (メソッド)
- メモリ開放は Ruby GC

実行

```
% ./xxx.rb
```



# Ruby/GSL で連立方程式を解く

```
#!/usr/bin/ruby
# 拡張ライブラリ gsl.so をロード
require 'gsl'

# 行列をクラス GSL_Matrix として実現
m = GSL_Matrix.new([0.18, 0.60, 0.57, 0.96],
                   [0.41, 0.24, 0.99, 0.58],
                   [0.14, 0.30, 0.97, 0.66],
                   [0.51, 0.13, 0.19, 0.85])

# 右辺のベクトル: to_gv は配列を GSL_Vector にするメソッド
b = [1, 2, 3, 4].to_gv
# LU分解: lu は分解された行列、p は置換情報が入る。
lu, p = m.LU_decomp
# LU分解した行列を使って連立方程式を解く。解はベクトル x に。
x = lu.LU_solve(p, b)
p x
```

$$A x = b, A = [m]$$

$m, b$  から一気に  $x$  を求めるメソッドを作ることはもちろん可能。

# 乱数

```
#!/usr/bin/ruby  
require 'gsl'
```

```
# 乱数オブジェクト  
r = GSL_Rng.new  
10.times { # 10回やれ  
  # 一様乱数  
  p r.uniform  
}
```

C では、

```
#include <stdio.h>  
#include <gsl/gsl_rng.h>  
  
int main () {  
  int i;  
  gsl_rng * r; /* 乱数発生器の構造体 */  
  const gsl_rng_type * T = gsl_rng_default;  
  gsl_rng_env_setup();  
  r = gsl_rng_alloc (T); /* 乱数生成器の初期化 */  
  for (i = 0; i < 10; i++) {  
    printf("%f¥n", gsl_rng_uniform(r));  
  }  
  return 0;  
}  そしてコンパイル。
```

# 数値積分

```
# 被積分関数をクラス GSL_Function で定義
f = GSL_Function.new{ |x|
  exp(-x*x)
}
# オブジェクト f に積分用メソッド integration_qng を起動
# 引数は積分の下端、上端、絶対精度、相対精度
# 戻り値は [積分値、誤差、関数値の評価回数、積分関数の返したステータス]
# を要素とする配列
ans = f.integration_qng(-10.0, 10.0, 0.0, 1.0e-7)
p ans
```

出力: [1.772453851, 1.967820304e-14, 87, 0]



$\pi$

# 統計

```
# 配列データを GSL_Vector に  
v = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].to_gv  
p v.stats_mean          # 平均  
p v.stats_variance     # 分散  
p v.stats_sd           # 標準偏差  
p v.max                 # 最大値  
p v.min_index          # 最小値のインデックス
```

# 現在までの進行状況

- ベクトル、行列
- 線形代数、固有値問題
- 数値積分
- 統計、ヒストグラム
- 特殊関数
- ソート
- 乱数

# Ruby/GSL の今後

- GSL の機能の網羅 ( ? )
  - ODE
  - Nonlinear Least-Square Fitting
  - Interpolation
- ラッパーの自動生成 (SWIG)
- Arno Erpenbeck 氏との collaboration  
(もう1つの GSL の Ruby 化)
- フロントエンドプログラムを作るか？
  - 例えば octave, MaTX のような
  - irb との親和性を高める
- ドキュメント (!)