モダンな Fortran 活用の動機

f2003 以降の機能、特に OOP について

発表者：坂本 裕一郎

地球流体データ解析・数値計算ワークショップ, 2022/3/31

モダン Fortran 勉強会 (Fortran-jp)

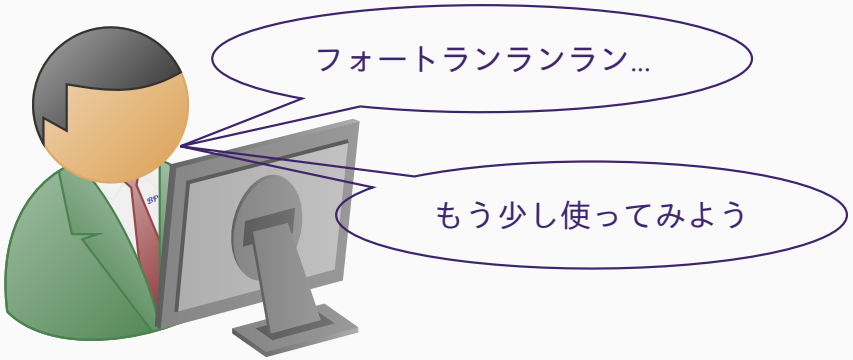
目次

概要

本日の話題

とある Fotran ユーザの所感を紹介

- 従来の FORTRAN に抱いた不満
- 不満解消の為に Fortran ライブラリを自作した話
- モダンな Fortran を活用した標準ライブラリへの期待



この資料内容は、高性能 Fortran 推進協議会, 第 6 回シンポジウム (2021) と同様です

概要

自己紹介

普段の仕事

- 鉄道車両の走行安全性に関わる研究
- 実験：現車試験、試作品試験、など
- 数値解析
  - Fortran によるマルチボディダイナミクス
  - OpenMP, Coarray による並列化
  - R,MySQL を活用したデータ処理

自己紹介

開発環境と利用ツール

OS	Ubuntu 20.04
Editor	Vim
Compiler	gfortran, nagfor
Tools	cmake, autotools, doxygen, bash, sed, awk, ...
Animation	f03gl, OpenGL (<1.5)
Graphics	R, gnuplot, plplot, ogpf

概要

私の FORTRAN 前史 (〜2005 年頃)

学部生の時に C 言語を学ぶ

- 大学の情報処理講座が C 言語
- シューティングゲームを作ってみた
- ポインタが難しいと感じていた

Fortran との出会い (2008 年頃)

研究室配属

- FORTRAN77(若干 f90) のコードを初めて見る。
- コメント無し、変数名短い、計算速度早い。
- ルーチン引数が多い。覚えるのに苦労。
- ドキュメントは論文。実装は論文にない工夫も多い。

Fortran 初心者の思い出 (2009 年頃)

- implicit none
  - 暗黙の宣言型により、l と 1 のタイプミス検出に苦労
- 多目的最適化手法の開発
  - 計算の進行に伴い、多数の解を保存する必要あり
  - 多数の解に対して、走査、挿入、削除を頻繁に行う
  - pointer を活用したリンクリストの作成

fortran2003 に気付く (2011 年頃)

可変サイズの配列を保持するデータ型が欲しくなる。

1. 構造体の中で配列を定義してみた。
2. その配列を allocatable にすれば良いと考えた。
    - コンパイラがエラーメッセージを出した。
    - 「この規格に未対応」という趣旨。
3. 新しい規格の存在に気付く。
    - 2003 年の規格に 8 年ほど経って対応されていない。
    - もしかして fortran(2003) は利用者が少ない？

R と fortran の連携 (2011 年頃)

膨大なデータ整理を R と fortran で

- R はデータ整理に都合が良い
  - 図の作成が簡単
  - メモリに載らなければ SQL を活用出来る
  - vim との連携が可能 (Nvim-R)
- 当時の R は遅かった (私の R プログラミングスキルが低かっただけかも知れません)
  - 重たい処理を fortran(+OpenMP) ルーチン化
  - R から fortran の so を call し劇的な速度向上を達成

fortran である必要はあるのか (2015 年頃)

軽い計算をするだけなら

Python Scipy, Numpy があればほぼ何でもできる。

Julia 計算は早い。開発途上。

Octave Matlab 互換。

R Python,fortran,C を Call できる。

fortran の現状に対する疑問と不満

- FORTRAN(77) は引数がやたら多い
- fft, 補間, フィルタ処理をどう実現するか

fortran でも OOP を使えば Scipy のようなライブラリができそうなのに · · ·

fortran の重要な役割は何か

非常に重たい処理をスパコン等で効率的に実行すること。

- 計算は早い方が良い
  - メモリコピー、動的型付けの比較はしたくない。
  - 必要なメモリはプログラム開始時に割付 (FORTRAN77 的記述方法)
- 高度な並列化 (MPI,OpenMP,Coarray,OpenACC,CUDA)
- 配列演算を簡単に書きたい (fortran>=C++>C)

一方、速度より使いやすさを重視するユーザがついて来るか？

2015 年時点の fortran に関する感想

新規ユーザー獲得が難しいのではないか。

- Scipy のような統一感のあるライブラリが無い。
- 補間や数値積分のためにワーク配列まで意識するのはつらい。(よくある iwork,work 仮引数)
- f2008 までを網羅した fortran 日本語書籍の不在。

ひとまず、再利用しやすい自分専用ライブラリを作る。

- fortran の OOP を活用
- Octave,Scipy のような利用のしやすさ

概要

オブジェクト指向のメリット

- データと手続きの一体化
  - iwork, work など、よく有るワーク配列を隠蔽できる
  - 滅多に変えない引数へデフォルト値を設定可能
- コードの再利用、並列化が容易
  - オブジェクト自身がデータを持っている
  - スレッド並列等で private,public の指定が容易
- 型を拡張可能
  - 仮引数を増やさずに追加データを考慮可能

概要

どのようなライブラリが欲しいか

(例) 実験と解析の橋渡し

- 実験で得られた特性曲線の関数化
- 数値積分, 求根, 常微分方程式の解法

netlib で大抵は入手可能であるが...

- DIERCKX, QUADPACK, MINPACK, ODEPACK, RKSUITE, ..
  - ファイル I/O ライブラリがない
  - netlib はドキュメント方式が多様で難解

Python,Octave,R でもできるが...

- うまくできたら fortran プログラムで使いたい
- 最初から fortran で簡単に試行錯誤したい

ファイル I/O の例 (CSV)

Octave

```
x=dlmread(file,sep=',')
```

R

```
library(tidyverse)
x <- read_csv2(file)
```

Python

```
import pandas as pd
x=pd.read_csv(file,header=0,index_col=0)
```

fortran

```
integer :: i,uni,n=10
real(kind=real64) :: x(10,n)
open(newunit=uni,file=fname)
do i=1,n
  read(uni,*) x(:,i)
end do;close(uni)
```

こうしたい

```
type(dfstats) :: x
call x%dlmread(fname,sep=',')
```

概要

ODEPACK(常微分方程式初期値問題)の例

一階の常微分方程式初期値問題 ($\boldsymbol{X}$：状態変数)

$$\frac{d\boldsymbol{X}}{dt} = F(t, \boldsymbol{X})$$

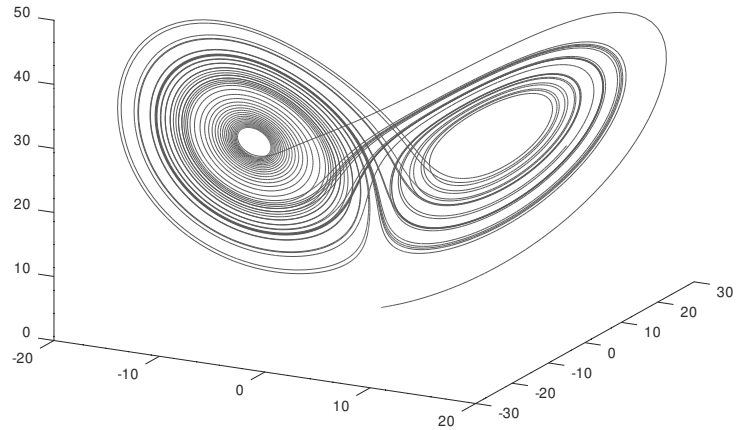
lsode, lsoda

- Octave, Python(scipy)の背後で動くルーチンの一つ

Octave で ODE (with ODEPACK)

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \boldsymbol{f}(t, \boldsymbol{x}) = \begin{bmatrix} a(x_2 - x_1) \\ bx_1 - x_2 - x_1 x_3 \\ x_1 x_2 - cx_3 \end{bmatrix}$$

```
function dx=f(x,t)
  a=10;b=28;c=2.666;
  dx(1)=a*(x(2)-x(1));
  dx(2)=b*x(1)-x(2)-x(1)*x(3);
  dx(3)=x(1)*x(2)-c*x(3);
endfunction
x0=[0.5,0.5,0.5];
t=linspace(0,50,10000);
y=lsode('f',x0,t);
plot3(y(:,1),y(:,2),y(:,3))
```



fortran で ODE (with ODEPACK)

```
program fortran_lsode
  implicit none
  integer :: itol,itask,istate,iopt,lrw,liw,mf
  double precision,dimension(:),allocatable :: y,rtol,atol,rwork
  ..........
  call dlsode(fun,neq,x,t,tout,itol,rtol,atol,itask,istate,&
           & iopt,rwork,lrw,iwork,liw,jac,mf)
  ..........
  contains
    subroutine fun(neq,t,x,dx)
      integer :: neq
      real(kind=real64) :: x(neq),dx(neq),a,b,c,t
      a=10d0;b=28d0;c=2.666d0
      dx(1)=a*(x(2)-x(1))
      dx(2)=b*x(1)-x(2)-x(1)*x(3)
      dx(3)=x(1)*x(2)-c*x(3)
    end subroutine
end program
```

同じライブラリを使っているのに面倒すぎる!!

lsode を使いやすくする (準備編:データの定義)

lsode の引数を構造体で保持

```
type,private :: lsod_base
  integer :: neq, itol, itask, istate, iopt, lrw, liw
  integer,dimension(:),allocatable :: iwork
  real(kind=dpkind) :: rtol, t
  real(kind=dpkind),dimension(:),allocatable :: y,atol, rwork
  procedure(odepack_fcn),nopass,pointer :: fcn => null()
  procedure(odepack_jac1),nopass,pointer :: jac => null()
  contains
    final :: final_lsod_base
end type
type,extends(lsod_base),public :: lsode
  integer :: mf
  contains
    procedure :: setup => setup_lsode
    procedure :: run => run_lsode
end type
```

lsode を使いやすくする (準備編:初期化部分)

setup ルーチン

- オプショナル引数機能を活用
- rwork,iwork 配列の一部に設定する値を型で定義

```
type,public :: lsode_options !default value
   real(kind=dpkind) :: h0=0d0,hmax=0d0,hmin=0d0
   integer :: mxstep=1000000,mxhnil=0,mxordn=0
end type
```

全ての引数を指定すると

```
call t_lsode%setup(fcn,neq,t,itol,rtol,atol,itask,istate, &
                   & opt,jac,mf)
```

最低限必要な引数のみを指定すると

```
call t_lsode%setup(fcn,neq,jac=jac)
```

lsode を使いやすくする (準備編:実行部分)

run ルーチン定義部分

```
subroutine run_lsode(self,tout)
  class(lsode) :: self
  real(kind=dpkind) :: tout
```

run ルーチンの呼び出し方

```
call t_lsode%run(tout)
```

- setup ルーチンで必要なパラメタを設定済み
- 任意の時刻 t を指定するだけで $\boldsymbol{X}$ が得られる

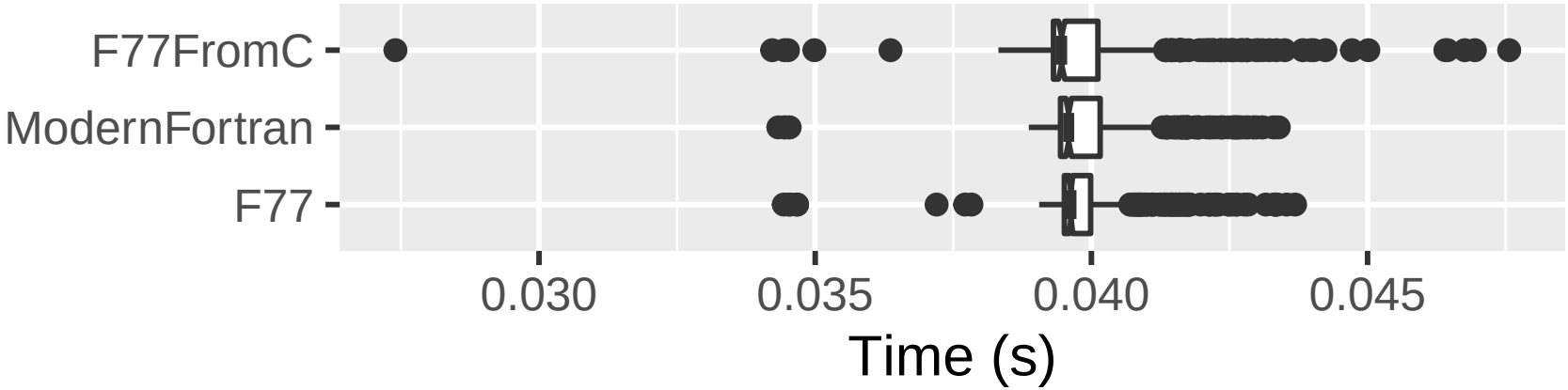
lsode を使いやすくする (実践編)

```
program modernized_lsode
  use mod_ode ! definition of 'lsode'
  implicit none
  type(lsode)        :: t_lsode
  real(kind=real64) :: dt,time
  call t_lsode%setup(fcn,3,jac=jac,mf=10) ! setting up
  dt=1d-2; time=0d0
  t_lsode%y(:)=[0.5d0,0.5d0,0.5d0] !initial value
  do
    call t_lsode%run(time) !proceeding with the time step
    time=time+dt; print *, time,t_lsode%y(:)
    if(time>50d0)exit
  end do
  contains
    subroutine fun(neq,t,x,dx)
      ...
    subroutine jac(neq,t,x,ml,mu,pd,nrowpd)
end program
```

コードがスッキリ見えませんか?

計算時間の比較(F77 vs ModernFortran)

- LSODE で同じ問題を解く (Corei7-2600)
- 500 回の計算時間 (中央値) を比較...ほぼ同じ



概要	F77	ModernFortran	F77FromC
計算時間 (s)	0.03963	0.03959	0.03946
F77 との比率	1.00000	0.99894	0.99572

概要

DIERCKX を使ったスプライン補間の例

(F77) スプライン関数の生成に多数の変数定義が必要。

```
real(kind=real64),dimension(m) :: x,y,w
real(kind=real64) :: xb,xe,...
integer :: iopt,m, ...
call curfit(iopt,m,x,y,w,xb,xe,k,s,nest,n,t,c,fp,wrk,&
   &lwrk,iwrk,ier)              !generate spline function
call splev(t,n,c,k,x,y,m,ier) !evaluation
```

(OOP 化) 構造体を活用すると変数の扱いがラクに。

```
real(kind=real64),dimension(m) :: x,y
real(kind=real64) :: xeval
type(curfit1) :: spltype
call spltype%sett(x,y)   !generate spline function
yeval=spltype%eval(xeval) !evaluation
```

OOP 化により、Scipy,Octave 並の簡易さに

概要

F77 を OOP でラッピングする時の注意点

- スレッドセーフ性
  - COMMON 文とパラレルリージョンの関係に注意
  - ルーチンと引数を構造体に入れても、呼び出し先で用いられる COMMON 変数は構造体の管轄外 (private/share 属性に気をつける)
  - lsode は COMMON 文が含まれている
- procedure(interface) の定義
  - “わざと実引数と仮引数の型ミスマッチのまま使う” など、F77 のライブラリは、インターフェイスを定義しない方が良い場合もある

COMMON 文が無い場合は並列化が簡単

同じアイデアで並列化を簡単に(OpenMP)

- Runge Kutta 法を f2003 で新規コーディング
- a,b,c をスレッド毎に変えてみる

```
type,extends(rk)  :: rkwithpara  ! extended type
  real(kind=real64) :: a=10d0,b=28d0,c=2.66666d0 ! parameter
end type
type(rkwithpara) :: rkobj(600)
real(kind=real64),parameter :: x0(3)=[1d0,1d0,1d0]*0.5d0
... !set up a,b,c
!$OMP parallel do private(i)
do i = 1,600
  call rkobj(i)%setup(fcn,1d-4,3,"rkg",x0) !fun,dt,nx,solver,initial
  do
    if ( rkobj(i)%time >50d0 ) exit
    call rkobj(i)%drive(rkobj(i)%time+1d-2)
  end do
enddo
```

Ryzen5 2600: 1 スレッド:約 70 秒 → 6 スレッド:約 12 秒

同じアイデアで並列化を簡単に (Coarray)

```
type,extends(rk4) :: rkwithpara
  real(kind=real64) :: a=10d0,b=28d0,c=2.66666d0
end type
type(rkwithpara) :: rkobj[*] !coarray
... ! pre processing (set different a,b,c)
call rkobj%setup(fcn,1d-4,3"rkg",x0)
do
  if(rkobj%time > 50d0) exit
  call rkobj%drive(rkobj%time+1d-2)
end do
... ! post processing
```

- 宣言部の変更のみで、あとは 1CPU と全く同じコード
- 並列化の詳細は ”pre, post processing” で

拡張型を使う利点

```
type,extends(rk)  :: rkwithpara  ! extended type
  real(kind=real64) :: a=10d0,b=28d0,c=2.66666d0 ! parameter
end type
```

- a,b,c を渡すために仮引数を変更しなくて良い
- メソッドが利用するデータを型内部のデータと仮引数に限定することで、スコープを明瞭にできる
- 配列要素数や引数の数に悩まない。記憶力に優しい。

コードの再利用がしやすい ≃ 他人が見ても理解しやすい

概要

バージョン管理の重要性(個人開発)

手動

1. ファイル編集
2. ファイル名変更
3. diff new.f90 old.f90

この変更の意図は?

ツール : git, subversion

1. ファイル編集
2. git add
3. git commit -m “msg”
4. git diff
5. git log
6. git tag

ツールの利用で確実に開発の記録が取れる。

git : 分散型バージョン管理

バージョン管理の重要性(GitHub で共同開発)

folk	開発元のソース等を GitHub 上で複製
clone	開発元 or folk のソースをローカルにコピー
branch	機能追加版の開発
push	レポジトリに変更点を送信
PR	プルリクエスト (GitHub 上の操作)

pull request

- 開発元に自分の変更点を知らせる機能
- レビュー後、問題なければマージしてもらえる

コードとドキュメントの同時開発

自分で書いたコードの意図を完全に思い出せますか？

Doxygen による文書化

- コメント形式でコードに説明を追記
- HTML 等の形式でドキュメントを生成
- call,caller グラフを自動生成性
- ルーチンの検索窓も付属
- (例)`http://www.netlib.org/lapack/explore-html/index.html`

Doxygen サンプルソースコード

```
!=============================
!>@name 統計処理ルーチン
!>@{
!=============================
!>@brief ベクトルの平均値
!>
!>@f$\displaystyle\bar x=\frac{1}{n}\sum_{i=1}^{n}x_i@f$
pure function average(x) result(xave)
  real(kind=dpkind),dimension(:),intent(in) :: x !< ベクトル
  real(kind=dpkind) xave !< ベクトルxの平均値
  integer :: n
  n=size(x)
  xave=sum(x)/real(n,kind=dpkind)
end function
!>@}
```

Doxygen により生成される HTML の例

◆ average()

pure real(kind=dpkind) function math::average (real(kind=dpkind), dimension(:), intent(in) x)

ベクトルの平均値

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

引数

[in] **x** ベクトル

戻り値

xave ベクトルxの平均値

Mod_Math.f90 の 471 行目に定義があります。

```
471         real(kind=dpkind),dimension(:),intent(in) :: x
472         real(kind=dpkind) xave
473         integer :: n
474         n=size(x)
475         xave=sum(x)/real(n,kind=dpkind)
```

Cmake or Autotools

ビルド・テスト・インストールの手順

1. git pull
2. コンフィギュレーション (cmake または ./configure)
3. コード修正
4. ビルド (make)
5. テスト (ctest または make check)
6. git commit & git push
7. インストール (make install)

(参考)

```
Autotools https://www.gnu.org/software/autoconf/
  Cmake https:
        //qiita.com/ijknabla/items/05270ae5e597705d0dae
```

概要

Fortran Standard Library (2019 年〜)

望んでいたライブラリの開発が始まった

- fortran2003 以上 (Scipy 並の簡単さでコーディングできそう)
- 適度にカプセル化されたユーザフレンドリな API
- 良く整理されたドキュメント (FORD による)

Goals and Motivation

The Fortran Standard, as published by the ISO, does not have a Standard Library. The goal of this project is to provide a community driven and agreed upon de facto ”standard” library for Fortran, called a Fortran Standard Library (stdlib).

ISO (`https://wg5-fortran.org/`)

Fortran Standard Library

Scope

Utilities strings, files, OS/environment integration, logging, ...

Algorithms searching and sorting, ...

Mathmatics linear algebra, fft, random numbers, statistics, ordinary differential equations, optimization, ...

- これだけ揃えば自作ライブラリは不要。
- デスクトップからスパコンまでをシームレスに。

使いやすくしたいので、バグ報告をしてみた。

Fortran Standard Library

fortran-lang `https://fortran-lang.org/`

fstdlib `https://github.com/fortran-lang/stdlib`

fpm `https://github.com/fortran-lang/fpm`

- fortran の情報集約ページができた！
- stdlib, この使用感が欲しかった！
- fortran のパッケージマネージャ (fpm) も開発中！

おわり

Fortran Standard Library の現在 (2021)

- 活発に開発が続いています。
- レビュアーが足りていません。
- 国際学会 FortranCon で 2 年続けて講演されました。

ぜひ開発にご協力ください

ご清聴頂きありがとうございました。