日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

計算ソフト： Python sf

小林憲次

Python sf は計算ソフトの L.L. です。エディタでメモ書きしていく数式のままに計算し、またグラフ表示させたくて作りました。前処理を被せることで名前空間を拡張し、計算処理で暗黙のうちに前提としている文脈 : 計算処理を、この拡張名前空間の中に隔離してカスタマイズ実装できるようにしました。この隔離・カスタマイズにより多くの計算を短いワンライナーで記述可能になりました。scipy 等の膨大なソフト資産も活用することで Mathematic/Matlab と同等以上に使える計算ツールになりました。日常行う計算の 9 割以上を、短いワンライナーで無理なく書けるのですから、Python sf は計算ソフトの L.L. だと言えます。

1 はじめに

Python sf は、日常エディタでメモ書きしている数式のままに計算し、またグラフ表示させることを目指しています。前処理を被せることで、Python 文法にアッパー・コンパチを保ちながら、数学記述で暗黙のうちに前提とされていることを反映させた数式記述を可能にしました。これにより、scipy などの数学ソフト資産を活用しながら、同時に数式記述に日常使っている数学での記述に近いものにできました。

数学では x,y,z は加減乗除べき乗算が可能な変数だと暗黙の内に仮定されています。積演算子 * は省略可能です。これらの暗黙の前提を反映した数式記述のままに計算することが強く望まれます。微分は ∂ x, ∂ y, ∂ z 記号で表せるようにすれば、日常メモで書いている数式に殆ど同じにできます。

実際 Python sf では下のような数式記述で、コンピュータに計算処理をさせられます。

Python sf one-liners

・基本関数

```
sin(1)
================================
0.841470984808
```

・基本関数の加減乗除べき乗算と合成

```
(sin(‘X)/‘X + 3cos(‘X^2+‘Y^2))(1,2+5‘i)
================================
1.6924575412
```

・数値微分

```
∂ y(sin(‘X)/‘X + 3 cos(‘X^2+‘Y^2))(1,2)
================================
11.5070909551
```

・シンボリック微分

```
ts(); ∂ y(ts.sin(‘x)/‘x
               + 3 ts.cos(‘x^2+‘y^2))
================================
-6*y*sin(x**2 + y**2)
```

・Pauli 行列 σ x, σ y, σ z の exp 行列

```
t=2; expm( 2 ‘π ‘i t (
                ‘σ x+2 ‘σ y + 3‘σ z))
================================
[[-0.9945+0.0839j   0.0559+0.02796j ]
 [-0.0559+0.0279j  -0.9945-0.08390j]]
---- ClTensor ----
```

変数文字列の前後に逆クォート:‘ を追加した名前を使えるようにすることで、Python の文法と矛盾することなく、数学的オブジェクトを短く表記できるようにしています。‘X,‘Y は、引数タプルの一番目、二番

Calculating soft: Python sf.

Kenji Kobayashi, –, kVerifier Lab.

目の値を取り出す関数では。それらは加減乗除べき乗算が可能な関数にもなっています。ts() は Python でのシンボリックな計算処理を可能にする sympy モジュールを取り込む関数であり、‘x,‘y は sympy でのシンボル変数です。

数式記述で使われる暗黙の前提はユーザーごとに異なります。そのため Python sf 式を実行する前に customize.py ファイル import しています。Python sf 式で定義済みとして使える、暗黙の前提となる Python オブジェクトを customize.py に記述できるようにしています。上で使った ‘X,‘Y, ‘x,‘y, ∂ x, ∂ y, ts, σ x, σ y, σ z は customize.py に定義されています。

Python には scipy パッケージに代表される多くのソフト資産の蓄積があります。Python sf は Python 文法にアッパ・コンパチであり、それらのソフト資産を全て利用できます。上の Python sf 式での sin, cos は numpy パッケージの sin, cos に加減乗除べき乗算を可能にするためのクラスを被せたものです。

プリプロセッサ、逆クォートによる名前空間の拡張と customize.py ファイルなどを活用することにより、ユーザーが日常行っている計算処理の九割以上はワン・ライナーで無理なく記述できるようになります。数式計算では if then else 文を使う必要が殆どないので、ワンライナーでも可読性を損ないません。

このような Python sf は計算ソフトの L.L.:Light weight Language だと言えます。

1.1 Python sf はコマンド・ライン・プログラム

Python sf はコマンド・ライン・プログラムです。Mathematica/Matlab などのように notebook から操作するのではなく、テキスト・エディタでメモ書きしているままで計算させることを理想としています。エディタのコンソール・モードで以下のようなエディタ・マクロを組み込んであることを前提としています。

例えば下の sin の微分値を求めるときを考えましょう。

$$\left.\frac{d\ sin}{dx}\right|_{x=0}$$

そのためには次のようなエディタのコンソール・モードで動くエディタ・マクロを使います。まず上の微分式を意味する下の Python sf 式の書いてあるエディタ画面上で ”∂ x(sin)(0)” 文字列の行にカーソルを置いて、その一行をコピー:yank します。そして

Python sf one liner

```
・Python sf ワンライナー
∂ x(sin)(0)
```

そのコピーした文字列の前に ”python -u sfPP.py ”文字列を追加した下のコマンドをコンソールで実行します。すると sfPP.py がプリプロセッサを呼び出し、”∂ x(sin)(0)” 文字列を Python 2.5 以上で扱える文字列 ”print k__Round_x___(sin)(0)”に変換し、それを Python に実行させ、”==========”と ”0.999999998333” の文字列をコンソールに打ち出させます。

command

```
python -u sfPP.py "∂ x(sin)(0)"
==============================
0.999999998333
```

ここで、計算結果の print 命令は Python sf プリプロセッサが自動的に補完します。Python sf 式のワンライナーが与えられたとき、その最後の式の評価値がユーザーが望んでいる計算結果になっているはずだからです。

ここで sfPP.py は下のような python コードです。プロプロセッサ sfPPrcssr.py を開始させているだけです。

command

```
type sfPP.py

import pysf.sfPPrcssr
pysf.sfPPrcssr.start()
```

計算させたい Python sf 式の規模が大きくなり、ワンライナーで記述するのが不適当になったときは、Python sf ブロック式を使います。このときは次のようなエディタ・コンソール・モード・マクロを実行させます。下のような Python sf ブロック式をエディタで書き、カーソルを ”//@@” と ”//@@@” の間の行に置きます。

Python sf block equations

```
//@@
a= 0
```

```
if a >= 0:
    print ∂ x(sin)(a)
else:
    print ∂ x(cos)(a)
//@@@
//copy c:\#####.### temp.py /y
//python sfPP.py -fs temp.py
//python -u __tempConverted.py

0.999999998333
```

そしてエディタ・マクロに次のような処理をさせます

1. //@@ .... //@@@ の間にあるテキストを C:\#####.### ファイルに書き込み
2. //@@@ の次にある // で始まるコマンドを実行していく

2a C:\#####.### ファイルをカレントディレクトリの temp.py にコピーする

2b python sfPP.py -fs temp.py コマンドで前処理させて Python で実行可能な __tempConverted.py を生成させる

2c __tempConverted.py を Python で実行させる

このようなエディタ・マクロが ”python sfPP.py -fs temp.py” を実行して temp.py をプリプロセッサ処理させると、上の Python sf ブロック式は下のような Python2.5 で実行させられる __tempConverted.py ファイルに変換されられます。

command

```
type __tempConverted.py

from __future__ import division
# -*- encoding: cp932 -*-
from pysf.sfFnctns import *
setDctGlobals(globals())
from pysf.customize import *
if os.path.exists('.\sfCrrntIni.py'):
    from sfCrrntIni import *
a= 0
if a >= 0:
    print k__Round_x___(sin)(a)
else:
    print k__Round_x___(cos)(a)
```

__tempConverted.py の中身を見ていると、Python sf が内部で行っている内容が見えてきます。”from pysf.sfFnctns import *” により、Python sf で使う関数やクラスたちをグローバル名前空間に取り込みます。これにより sin,cos などの関数やグラフ表示関数などを呼びだせるようになります。

customize.py、共通に使う関数や変数を定義しておきます。∂ x も、ここで定義してあります。”from pysf.customize import *” により、それらを Python sf 式で使えるようになります。ユーザー独自の関数などを Python sf 式で使いたいときは、 customize.py をユーザーが追加・修正することになります。

ちなみに ∂ x は customize.py に定義してあり、下のような内容となっています。

Python code

```
k__Round_x___ = __k_P0__          # ∂ x
```

この k__Round_x___ すなわち __k_P0__ の具体的な Python コードは下のようになっています。

Python sf one-liner

```
sc.source(∂ x)
In file: D:\my\vc7\mtCm\pysf\customize.py

def __k_P0__(fnAg):
    if False:
        pass
    else:
        return sf.P_(0,fnAg)

================================
None
```

sc.source の sc は numpy package の別名です。numpy.source 関数は引数に与えた関数やクラスの Python ソース・コードをコンソールに打ち出します。このように Python sf 式のワンライナーは計算以外でも便利に使えます。Python sf 式の処理対象は計算式に限りません。全ての Python コードが扱える全ての対象を Python sf 式で扱えます。

Python sf 式は GUI ではなく、コマンド・ラインで扱われます。漢字文化圏では ∇ □ ∂ △ の数学で多用する特殊記号フォントを使えます。アルファベット文化圏の計算ソフトで微分を扱おうとすると、Mathemaica/Matlab のような Notebook が必要になるのも理解できます。でも ∂ 漢字フォント記号を使えば、微分を表現できます。”∂ x(sin)(0)” により 0 における sin の微分値であることを表現できます。ならば面倒な Notebook ソフトを使うのではなく、エ

ディタから直接計算するほうが便利です。記号やギリシャ文字を多用する計算ソフトでは漢字フォントを活用すべきです。漢字文化圏の人たちでけにかぎらず、アルファベット文化圏の人たちも、数学ソフトでは漢字フォントを活用すべきです。現在の OS はそれが可能になっているのですから。

Python sf 配布パッケージの中に idle に手を加えた kidlelib パッケージを同梱してあります。これに上に述べてエディタ・コンソール・モード・マクロを組み込んであります。でも idle ではエディタ機能が貧弱すぎます。ユーザーの常用するエディタに、上に述べたようなコンソール・マクロを作って組み込んで Python sf を使うことを強く推奨します。

2 Python sf 全体の構成と機能

Python sf パッケージを構成する Python モジュール群を眺めながら、Python sf の全貌を見ていきましょう。

Python sf パッケージ構成		
モジュール名	**機能**	**備考**
sfPPrcssr.py	プリ・プロセッサ	ソース非公開
	sfPPrcssr が取り込むモジュール	
	customize.py	カスタマイズ
	sfCrrntIni.py	カスタマイズ
sfFnctns.py	行列	ソース非公開
	sfFnctns が取り込むモジュール	
	bascFnctns.py	iterator 他
	kNumeric.py	基本関数 他
	vsGraph.py	グラフ表示
	rational.py	線形系有理式
octn.py	八元数など代数系	代換名 oc
kre.py	/,% による正規表現	代換名 kre
kv	kVerifier テスト	代換名 kv
tlRcGn.py	無限数列 無限再起	代換名 tn

表 1 Python sf パッケージ構成表

2.1 プリプロセッサ : sfPPrcssr

プリプロセッサを介すことで、計算式を簡便に・日常使ってるメモ書きの数式と殆ど同じ形式で書けます。

1. 数学での積 : *演算子の省力を可能にします
2. 数学で多用されるギリシャ文字を使えるようにします。
3. 記号∇□∂△を使えるようにします。
4. 「‘」文字を変数名の前後に追加できるようにして、名前文字列に使える範囲を拡張します。
5. Python の lambda 構文で λ を使えるようにします
6. 「^」 演算子をべき乗の意味にします。Python の exor の意味ではなくします。
7. 「~」を演算子記号の前に追加したものをユーザー演算子として使います。演算子を拡張します
8. ~[...] により行列やベクトルを表記します
9. =:, := によりファイル変数の読み書きを行います

Python はプログラム一般を記述するための言語です。そのため Python で記述した計算式は冗長になってしまいます。数学を記述する慣例とも少し異なる記述が必要になります。プリプロセッサは Python sf 式を数学での慣例に近く記述することを、また短く記述できることを可能にします。

2.1.1 * 演算子の自動挿入

例えば数学では積 : *演算子は省力して記述する慣習があります。メモ書きした数式を計算するためには、* 演算子を省略可能にせねばなりません。下のような計算を可能にせねばなりません。プリプロセッサに * 演算子を挿入させねばなりません。

Python sf one-liner

```
a,b=3,4; 2 a b
==============================
24
```

ただし * 演算子の自動挿入は、丸括弧式 : (...) との組み合わせにおいて Python 文法との軋轢を発生させます。(...) には関数引数の意味になることもあるからです。

Python sf one-liners

```
a=2; a (3+4)
================================
14

     2(3+4)
================================
14

a=2; a(3+4)
'int' object is not callable at
excecuting:a(3+4)

a=lambda x:x+5; a(3+4)
================================
12

a=λ x:x+5; a(3+4)
================================
12
```

”2(3+4)” のように、数値と (...) がつながった時は * を挿入できますが、”a=2; a(3+4)” のようにアルファ・ニューメリック文字列と (...) が繋がったときは * を挿入できません。a が関数かもしれないからです。プリプロセッサの段階では a が関数なのか数値なのか判別できないからです。

このような分かり難さが伴っても、プリプロセッサにより *:積演算子を自動挿入することを選択しています。Python sf 式仕様の単純さよりも、* 演算子を省略するという数学での 慣例の方を優先すべきだと考えるからです。演算子を**:べき乗演算子の意味にするのも数学の慣例を優先したいからです。

メモ書きの数式と殆ど同じ形式のままに Python で計算させるということは、このような両者の仕様の矛盾をギリギリの所で妥協させる方策を見出していくことになります。

2.1.2 ‘:バック・クォートによる名前空間の拡張

‘:バック・クォートによる名前空間の拡張は、既存の Python で使われている名前との衝突を避けながら 短い Python sf 式を書くことを可能にします。Laplace 変換では多項式は s を使うのが慣例になっています。ならば逆に ‘s に有理関数クラス・インスタンスの一時の多項式を割り当てておいてやれば、‘s を使って有理関数の計算ができるようになります。‘s を使ってラプラス変換を使った線形系の計算が扱えるようになります。下のような具合です

```
Python sf one-liners
1/(‘s+1) + 1/(‘s+2)
================================

  2 s + 3
-----------
 2
s + 3 s + 2

# ステップ応答のグラフ表示
(1/(‘s+1) + 1/(‘s+2)).plotAnRspns(10s‘)
```

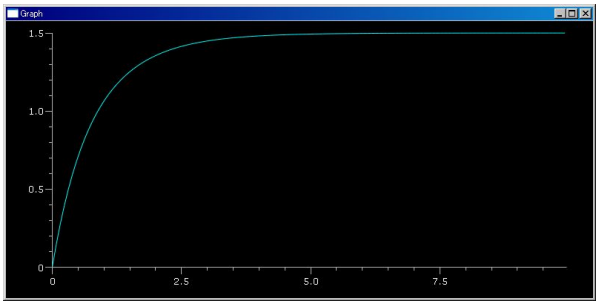

図 1　1/(‘s+1) + 1/(‘s+2) のステップ応答

```
(1/(‘s+1) + 1/(‘s+2)).plotBode(0.01Hz‘, 10Hz‘)
================================
```

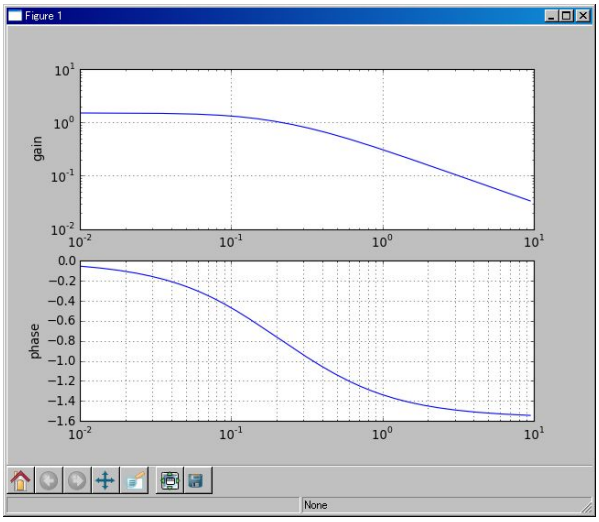

図 2　1/(‘s+1) + 1/(‘s+2) のステップ応答

後にのべるカスタマイ Python ファイルと ‘ による名前拡張を組み合わせることで、ユーザーごとに異なるデフォルトの数学慣習を短い名前を使って実装できます。上のように短い Python sf 式で可読性の高い計算式が記述できます。

本来は Python のバック・クォートには文字列評価の意味があります。でも殆ど使われていません。Python3000 では、なくなってしまいます。ですからバッククォートを名前拡張に流用しても殆ど問題ないと考えます。

なお、名前空間の拡張に ‘:バック・クォートを使ったのは、一般相対論で多用する Γ “__[0,1,0,1] のようなテンソル・インデックスの上げ下げを記述するのに使えることも理由のひとつです。

2.1.3 lambda λ

w

Python の lambda 構文を λを使って表せるのは、可読性を高めます。漢字 λの方が 英語の lambda より文字列の中から浮かび上がってくるからです。

Python sf one-liner
```
f=λ x:(λ y:3)(x+4)
            ; [f(x) for x in range(5)]
==============================
[3, 3, 3, 3, 3]
```

下の lambda 式より、上の lambda 式の方が、λ式の内容を理解しやすいはずです。

Python sf one-liner
```
f=lambda x:(lambda y:3)(x+4)
            ; [f(x) for x in range(5)]
==============================
[3, 3, 3, 3, 3]
```

λ を lambda 構文に割り振ることは、波長などの一般変数に λを使えないことを意味します。λ L などの変数名を使わねばならなくなります。そのような犠牲を払っても lambda 構文に λ を使えるほうが重要だと判断しました。

2.2 カスタマイズ・ファイル: custom.py, sfCrrntIni.py

プリプロセッサが Python sf 式の前処理を行って、Python2.5 以上で実行できる文字列に変換し、それを実行する前に ”from customize import *” ”from sfCrrntIni import *” を実行します。これにより customize.py, sfCrrntIni.py で定義されている変数・関数・クラスを Python sf 式で使えるようになります。

先に ‘s で Laplace 変換の有理式が記述できたのは、customize.py で下のように k__bq_s___ が定義されているからです。

Python code
```
#‘s: s of Laplace transformation
k__bq_s___ = sf.ClRtnl([1,0],1)
```

customize.py には共通するカスタマイズ Python コードを記述します。cusomize.py ファイルは pysf パッケージ・ディレクトリにあります。sfCrrntIni.py には個別のカスタマイズ Python コードを記述します。sfCrrntIni.py ファイルはカレント・ディレクトリに置きます。

2.2.1 ~ によるユーザー演算子拡張

~+,~-,~*,~/,~%,~&,~^,~| によりユーザー演算子を定義できます

例えば customize.py に「~*」を下のように定義してあります。

Python code
```
#  x ~+ y
k__tilda__UsOp_add____ = lambda x,y:x*y/(
                                    x + y)
```

これにより、抵抗の並列和を下のように計算できます。

Python sf one-liner
```
ts(); r1,r2=10 Ω‘,40 Ω‘; r1~+r2
==============================
8*V‘/A‘
```

2.2.2 物理単位

customize.py には物理単位が定義してあります。単なる数値だけの単位と、sympy の単位オブジェクト付きの二種類があります。Python の関数の多くは sympy の単位が付いた数値を受け付けないので両方を使える必要があります。ts() 関数を呼び出すと、sympy の単位オブジェクト付きとなります。

Python sf one-liners
```
10 uF‘ 4.7k Ω‘
==============================
0.047

ts(); 10 uF‘ 4.7k Ω‘
```

```
==============================
0.047*s‘
```

sympy に SI 単位系を定義した units モジュールが入っているのですが、symIntf.py を定義して SI 単位系を修正したものを組み込んでいます。SI 単位系では抵抗の単位が ohm ではなく $m^2 * kg/(A^2 * s^3)$ になり、また電圧の単位が $m^2 * kg/(A * s^3)$ となる欠点があるからです。sympy の unit モジュールでは、その欠点を そのまま実装しているからです。普通のエンジニアには $m^2 * kg/(A^2 * s^3)$ を抵抗の単位とは読めないからです。$m^2 * kg/(A * s^3)$ を電圧の単位とは読めないからです。

symIntf.py では基本単位をに V を追加し抵抗の単位を V‘/A‘ にしています。すなわち MKSA 単位系ではなく MKSA+V 単位系にしています。

Python sf one-liners

```
import sympy.physics.units as ut; ut.ohm
==============================
m**2*kg/(A**2*s**3)

import sympy.physics.units as ut; ut.V
==============================
m**2*kg/(A*s**3)

import sympy.physics.units as ut; ut.J
==============================
m**2*kg/s**2

ts(); ohm‘
==============================
V‘/A‘

ts(); V‘
==============================
V‘

ts(); W‘
==============================
A‘*V‘
```

MSKA + V 単位系は 電気系のエネルギーが W s の単位になってしまう欠点があります。電気系の Watt を Joule に直すには、ユーザーが明示的に J‘/s‘/W‘ を掛けてやらねばなりません。私は、これは欠点ではなく、利点だと考えています。大部分のエンジニアは Watt と Joule を、そのように変換するものとして理解しているからです。大部分のエンジニアは Volt も Ampere と同様な基本単位だからです。ただし、この問題は理解してもらうことが難しくて困っています。

電気系の MKSA 単位系では、エネルギーは J に統一されます

Python sf one-liners

```
import sympy.physics.units as ut;
    R, I = 3.3 ut.kilo ut.ohm, 1 ut.A;
    R I^2
==============================
3300.0*m**2*kg/s**3

ts(); R, I = 3.3k Ω‘, 1A‘; R I^2
==============================
3300.0*A‘*V‘

ts(); R, I = 3.3k Ω‘, 1A‘;
                R I^2 J‘/s‘/W‘
==============================
3300.0*m‘**2*kg‘/s‘**3
```

2.2.3 行列・ベクトル表記

~[...] によりベクトルや行列を表せます。プリプロセッサは ~[...] を krry__(*[...]) に変換します

Python sf one-liners

```
・ベクトル
~[1,2,3]
==============================
[ 1.  2.  3.]
---- ClTensor ----

・行列
~[[1,2,3],[4,5,6]]
==============================
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
---- ClTensor ----

・リスト内包表記による行列
~[x^2 for x in range(5)]
==============================
[  0.   1.   4.   9.  16.]
---- ClTensor ----
```

2.2.4 =:, := によるファイル変数の読み書

少し複雑な問題になるとワンライナーでは計算しきれないことが多くなります。でも Python sf では計算結果をファイル変数に残せます。カレント・ディレクトリのファイルに計算結果を記録しておき、その計算結果をファイルを読み出すことで再利用できます。

”name := value” で、カレントディレクトリの name.pvl ファイルに value を値を書き込みます。”=:name” で、カレント・ディレクトリの name.pvl を読み出し、その値を Python 変数 name に設定します。

```
Python sf one-liners
test:=3+4
==============================
7

=:test;test*5
==============================
35
command
type test.pvl
# python object printed out by pprint
7
```

ファイル変数は、明示的に消去しない限り、カレント・ディレクトリに何時までも残ります。何時でも再利用できます。ファイル変数は意外と便利に使えます。

2.2.4.1 _dt

Python sf ワン・ライナー式がの最後で := を含まない式のとき、最終式の値がファイルに残せる値のときはカレント・ディレクトリの _dt.pvl ファイルにファイル変数として、その値が記録されます。直後に必要になったときに再利用できます。

```
Python sf one-liner
3+4
==============================
7
command
type _dt.pvl
# python object printed out by pprint
7
Python sf one-liner
=:_dt; 5 + _dt"
==============================
12
```

2.3 sfFnctns

sfFnctns.py には Python sf で使う ClTensor(実数 or 複素数行列), ClFldTns(一般体係数を要素とする行列) が定義してあります。また basicFnctns.py, kNumeric.py, vsGraph.py, rational.py を取り込みます。もともと、これらのファイルのコードは sfFnctns.py 側にあったのですが、できるだけ多くのコードを公開するために別ファイルにしました。

2.3.1 ClTensor

ClTensor は int,float, complex 値を要素とする行列クラスです。numpy.ndarray を継承しています。

整数、実数、複素数を要素とする行列を [...] で作ると ClTensor 行列インスタンスが生成されます。行列要素のタイプを指定しないときは、デフォルトで float または complex 型になります。ClTensor 行列インスタンスどうしで 行列ベクトルの加減乗除、整数べき乗算が可能です。ClTensor ベクトル同士の掛け算のときは内積になります。numpy.ndarray 行列での掛け算は、要素同士の掛け算になります。numpy.ndarray での行列同士の積、行列とベクトルの積を計算させるときは sc.dot(,) 関数を呼び出さねばなりません。

```
Python sf one-liners
# default float vector
vct=~[1,2,3];vct
==============================
[ 1.  2.  3.]
---- ClTensor ----

# integer vector
vct=~[1,2,3,int];vct
==============================
[1 2 3]
---- ClTensor ----

# vector * integer
vct=~[1, 2,3];vct*3
==============================
[ 3.  6.  9.]
---- ClTensor ----

# vector + integer
vct=~[1,2,3]+100
==============================
[ 101.  102.  103.]
```

```
---- ClTensor ----

# vector+ vector
vct1,vct2=~[1,2,3],~[4,5,6]; vct1+vct2
==============================
[ 5.  7.  9.]
---- ClTensor ----

# vector* vector: 内積
vct1,vct2=~[1,2,3],~[4,5,6]; vct1*vct2
==============================
32.0

# default float matrix
mt=~[[1,2],[3,4]];mt
==============================
[[ 1.  2.]
 [ 3.  4.]]
---- ClTensor ----

# int matrix
mt=~[[1,2],[3,4], int];mt
==============================
[[1 2]
 [3 4]]
---- ClTensor ----


# matrix * lst
mt=~[[1,2],[3,4]];mt [10,100]
==============================
[ 210.  430.]
---- ClTensor ----

# matrix * matrix
mt=~[[1,2],[3,4]];mt mt
==============================
[[  7.  10.]
 [ 15.  22.]]
---- ClTensor ----

# inverse matrix by divide
mt=~[[1,2],[3,4]];1/mt
==============================
[[-2.   1. ]
 [ 1.5 -0.5]]
---- ClTensor ----

# inverse matrix by power
mt=~[[1,2],[3,4]];mt^-1
==============================
[[-2.   1. ]
 [ 1.5 -0.5]]
---- ClTensor ----

# inverse matrix * list
mt=~[[1,2],[3,4]];mt^-1 [100,200]
==============================
[  2.84217094e-14   5.00000000e+01]
---- ClTensor ----
```

ClTensor コンストラクタは整数引数のみであっても実数型の行列を生成します。明示的に int 型を指定しないと int 型の行列にはしません。

整数型行列の要素に float 値を代入したとき、小数点以下が切り捨てられてしまいます。理工学系の計算をしているとき、整数型行列・ベクタ要素に float 値を入れてしまうことは、どんなに注意していても やってしまいます。このミスはエラーを返さないうえに、計算結果も予想値に近いので、なかなか気付けません。理工系の計算ソフトとしてはデフォルトで float 値の行列・ベクトルにすべきです。

2.3.2 ClFldTns

ClFldTns は 一般体の値を要素とする行列クラスです。このような行列は scipy に用意されていません。一般体の値を要素とする行列は多方面に応用が利きます。ClFldTns 行列はブール体などの行列を扱えるようになります。デジタル回路を扱えるようになります。Python sf の基本関数とその組み合わせは加減乗除算が可能であり、体とみなせます。ClFldTns には __call__ が実装してあり、int, float, complex 引数を引き渡せます。これにより ベクトル関数が容易に記述できます。

一般体を要素とする行列を [...] で作ると ClFldTns 行列インスタンスが生成されます。ClFldTns 行列インスタンスどうしで 行列ベクトルの加減乗除、整数べき乗算が可能です。

```
Python sf one-liners
~[0,1,2], bool
==============================
[False  True  True]
---- ClTensor ----

# Bool Field Tensor
```

```
~[0,1,2, oc.BF]
================================
[0 1 0]
- ClFldTns:<class 'pysf.octn.BF'>


~[1,‘X,‘X^2]
================================
[<pysf.basicFnctns.ClAFX object at ..>
 <pysf.basicFnctns.ClAFX object at ..>
 <pysf.basicFnctns.ClAFX object at ..>]
- ClFldTns:<class 'pysf.basicFnctns..'>
unpicklable

~[1,‘X,‘X^2](2)
================================
[ 1.  2.  4.]
---- ClTensor ----

~[1,‘s,‘s^2]
================================
[ClRtnl([1.],[1.]),ClRtnl([1., 0.],[1.]),
       ClRtnl([ 1., 0., 0.],[1.])]
-- ClFldTns:<class 'pysf.rational.Cl..'>

krry([1,‘s,‘s^2])(2)
================================
[ 1.  2.  4.]
---- ClTensor ----


~[sin, cos, ‘X]
================================
[<pysf.basicFnctns.ClAF object at ..>
 <pysf.basicFnctns.ClAF object at ..>
 <pysf.basicFnctns.ClAFX object at..>]
- ClFldTns:<class 'pysf.basicFnctn..'>
 Error in pickling

~[sin, cos, ‘X] + ~[1, ‘X, ‘X^2]
================================
[<pysf.basicFnctns.ClAF object at ..>
 <pysf.basicFnctns.ClAF object at ..>
 <pysf.basicFnctns.ClAFX object at ..>]
- ClFldTns:<class 'pysf.basicFnctn..'>
unpicklable
```

2.4 sfFnctns がグローバルに取り込むモジュール

sfFnctns.py は basicFnctns.py, kNumeric.py, vsGraph.py, rational.py のモジュールたちを from ... import * します。グローバル変数に取り込みます。Python sf 式を実行する前には from sfFnctns import * が実行されているので、これらのモジュールに定義されている変数・関数・クラスらもグローバル変数に取り込まれており、Python sf 式からグローバル変数としてアクセスできます。以下、これらのモジュールに含まれている変数・関数・クラスらを見ていきましょう

2.4.1 basicFnctns

basicFnctns.py は mrng, arsq, masq, enmask, mitr, enmitr, klsp, combinate, permutate のループ記述を容易にするためのイタレータを実装します。

2.4.1.1 mrng

mrng は range 関数を複数組み合わせて書くループを一回の mrng で済ませます。下のコードを見れば何をやっているか分かると思います。

Python sf one-liners

```
list(mrng(4,4))
================================
[(0, 0), (0, 1), (0, 2), (0, 3),
 (1, 0), (1, 1), (1, 2), (1, 3),
 (2, 0), (2, 1), (2, 2), (2, 3),
 (3, 0), (3, 1), (3, 2), (3, 3)]

dct={}; for idx in mrng(4,4)
        :dct[idx]=sum(idx);~[dct]
================================
[[ 0.  1.  2.  3.]
 [ 1.  2.  3.  4.]
 [ 2.  3.  4.  5.]
 [ 3.  4.  5.  6.]]
---- ClTensor ----
```

Python に詳しければ、mrng は下手な説明よりも、そのコードを見たほうが分かりやすいとも思います。如何でしょうか。

Python sf one-liner

```
sc.source(mrng)
def mrng(*args):
    """' multiple range generator
         省略
    '"""
    head, tail = args[0], args[1:]
```

```
    if type(head) in [int, long, float]:
        head = range(int(head))
    elif isinstance(head, tuple) or\\
         isinstance(head, list):
        head = range(*head)
    if tail:
        for i in head:
            for j in mrng(*tail):
                if isinstance(j, tuple):
                    yield (i,)+j
                else:
                    yield (i, j)
    else:
        for i in head:
            yield i

================================
None
```

2.4.1.2 klsp

klsp(1+‘i, 2+3‘i, 5)

klsp(start, stop. num=50) は [start, stop] と start で始まり、stop で終わる領域を num-1 等分した ClTensor ベクトルを返します。ここで、num 当分するとき両端を含んで num-1 等分します。ですから ベクトルのサイズは num になります。start, stop には in, float, complex 値を指定できます。klsp(..) は scipy.linspace と同じです。返す値が ClTensor インスタンスであることだけが異なります。

klsp(...) が返すベクトルは両端を含むのでグラフの描画データの作成に便利です。[-N,N] 領域を偶数等分すると 0 を避けられるので、原点で無限大となる関数を扱うとき便利です

Python sf one-liners

```
klsp(-1,1, 10)
================================
[-1.         -0.77777778 -0.55555556
 -0.33333333 -0.11111111  0.11111111
  0.33333333  0.55555556  0.77777778
  1.        ]
---- ClTensor ----

[(1/‘X)(x) for x in klsp(-1,1, 6)]
================================
[-1.0, -1.6666666666666667, -5.0,
  5.0,  1.6666666666666665, 1.0]
```

その他の Python sf 基本イタレータを説明していると細かくなりすぎます。help(...) の説明を参照願います。

Python sf one-liner

```
help(arsq)
arSqnc(startOrSizeAg, sizeAg=None
                      , strideAg=1)
    等差数列タプルを start, size
    , stride 引数で生成して返す
       return arithmetic sequence
       generated with argument
       start, size, stride
    e.g.
   N=10;arsq(1,N, 3)
   ================================
  (1, 4, 7, 10, 13, 16, 19, 22, 25, 28)

================================
None
```

http://www.nasuinfo.or.jp/FreeSpace/kenji/ sf/pysf/manual/readMeC の「Python sf での繰り返し構文とイタレータ」の節により食わして説明があります。

2.4.1.3 ClAF

関数を加減乗除べき乗算と関数合成可能なものにするための ClAF クラスを実装しています。Python sf の三角関数などは ClAF インスタンスにしてあるので関数の加減乗除算が可能です。ClAF は次のようなメソッドを追加するクラスともみなせます。

ClAF
m_fn
__add__, __sub__
__mul__, __div__
__pow__
__call__

ClAF は、 Python の関数では無理な関数の加減乗除べき乗算を、無理やり可能にさせるクラスだともいます。

例えば $(sin + cos)(\pi)$ を次の Python sf 式で計算できます。math モジュールの math.sin, math.cos では関数の足し算ができません。

Python sf one-liners

```
(sin+cos)(‘π)
==============================
-1.0

import math as m;(m.sin+m.cos)(3.14)
unsupported operand type(s) for +:
’builtin_function_or_method’ and ’
builtin_function_or_method’ at
excecuting:(m.sin+m.cos)(3.14)
```

基本関数の加減乗除べき乗算を可能にすることは可読性の高い短い Python sf 式を記述するのに役立っています。‘X も加減乗除べき乗算が可能な関数であり $\int_{-\infty}^{\infty} exp(-x^2)\,dx$ を、関数のべき乗と関数合成で被積分関数を表し次の Python sf 式で計算できます

Python sf one-liner

```
quadR(exp(-‘X^2), -sc.inf, sc.inf)
==============================
1.77245385091
```

2.4.2 kNumeric

2.4.2.1 基本関数

kNumeric.py は基本関数 exp, sin, cos, tan, sinh, cosh, tanh, arcsin arccos, arctan, log, log10, sqrt, absF を実装しています。これらの基本関数は ClAF クラス・インスタンスになっており加減乗除べき乗算と関数合成が可能です。– これらの基本関数は sympy の 単位付き変数まで受け付けます。。ただし、sqrt, ‘X,‘Y,‘Z,‘T を除いて単位が打ち消しあっていないと AssertionError になります。

2.4.2.2 norm, normSq

シーケンス・データの ノルム : norm, ノルムの自乗:normSq を計算します。

2.4.2.3 fft, ifft

kNumeric.py は fft:Faset Fourier Transform, ifft:Inverse Fast Fourier Transform を実装しています。sc.fft.fft, sc.fft.ifft と殆ど同じですが、ClTensor インスタンスを返すことと fft,ifft 変換の前後で norm を変えないようにしていることが異なります。

2.4.2.4 行列関数

kNumeric.py は行列関数 diag, logm, eig, eigvals, eigh, eigvalsh を実装します。ClTensor インスタンスを返すこと以外は numpy.diag, numpy.logm, numpy.eig, numpy.eigvals, numpy.eigh, numpy.eigvalsh と同じです。

kNumeric.py は行列の exp,log 関数 expm, logm を実装します。ClTensor インスタンスを返すこと以外は scipy.expm, scipy.logm と同じです。

2.4.2.5 Jacobian 数値微分

kNumeric には Jc_ の名前で Jacobian 数値部分を行う関数を実装してあります。Python sf 式では ∂ J の名前でも呼び出せます。Jacobian 微分を普通に思われている以上に有用です。rotation, divergence, gradient を統一的に扱えるからです。四次元でも rotation を簡単に定義・計算できるからです。

Python sf one-liner

```
・複素数関数を含んだ四次元ベクトルの
  Jacobian
∂ J(~[‘i/norm(‘X,‘Y,‘Z),0,0,0])
                        (1,2,3,4)
==============================
[[ 0.-1.J  0.+0.j  0.+0.j  0.+0.j]
 [ 1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]]
---- ClTensor ----

・四次元の rotation
mt=∂ J(~[‘i/norm(‘X,‘Y,‘Z),‘X,‘Y,‘Z])
          (1,2,3,4); mt - mt.t
==============================
[[ 0.+0.j -1.+0.j  0.+0.j  0.+0.j]
 [ 1.+0.j  0.+0.j -1.+0.j  0.+0.j]
 [ 0.+0.j  1.+0.j  0.+0.j -1.+0.j]
 [ 0.+0.j  0.+0.j  1.+0.j  0.+0.j]]
---- ClTensor ----
```

2.4.2.6 数値積分

quadR 実数関数の積分, quadC 複素関数の積分, quadAn 解析関数の積分などを実装してあります。これらは scipy.integrate.quad を元に実装してあります。scipy の quad は計算誤差の推定値も一緒に帰してくれるので、Python sf ワンライナーでは、他の計算との組み合わせがやりにくいので、書き直しました。「使いにくいときは使いやすいように実装しなおせ」の例です。Python ですから簡単です。一番面倒な quadAn でも 36 行で実装されています。ソース

を見てみてください。

```
Python sf one-liners

quadR(cos, 0,pi)
================================
4.92255263497e-17

・フーリエ変換積分
f=0.1;quadC(λ x   :
    exp(-x^2) exp(2 pi ‘i x f),
    -sc.inf, sc.inf)
================================
(1.60587519197+0j)

・複素解析関数の閉ループ積分
quadAn(log, [1+‘i,1-‘i
               , -1-‘i,1-‘i,1+‘i])
================================
(-2.220-016j, 3.23-009, 3.23-009)
```

2.4.3 vsGraph.py

vsGraph.py モジュールは、できるだけ単純に少ない手数でグラフを表示させるために設けたグラフ表示のためのモジュールです。pylab, mlab などの Python 表示パッケージよりも手軽に表示する意味で使えます。

よく使うのは plotGr(..):一変数グラフの表示, plotTrajectory:二次元、三次元の軌跡の表示、plot3dGr:二変数の関数、または、一変数の複素関数の表示を行う関数です。

plotGr(sin(‘X)/‘X, -3‘π,3‘π)

2.5 octn.py

octn.py モジュールは代数系のクラスを集めたモジュールです。oc のモジュール名で Python sf 式からアクセスします。

八元数:ClOctonion:Oc, 有限体 $GF(2^8)$:RS(CD や DVD のエラー訂正で使われる Galois 体を実装してあります), 有限体:Zp(N)(素数の剰余体です), ブール体:BF, 対称群:Sn(N), 一般体係数の多項式:Pl : (多項式の割り算と剰余も扱えます) を実装してあります。数学の専門家にはとても不十分なないようですが、それ以外の方には結構使いでがあると思います。これらの代数系が ClFldTns の行列ベクタ・インスタントして扱え、また 多項式として扱えるのですから。

とくに八元数は重宝します。八元数は四元数を含みます。ですから、八元数一つで、可換体、非可換体、さらには推移律さえ満たさない代数系を得られたことになります。行列よりも手軽に扱えるので、反例と具体例の境目を探すときに重宝します。

以下のような演算が可能です

```
Python sf one-liners

・oc.Oc クラスを四元数として扱う
oc.Oc(1,2,3,4)
================================
(1, 2, 3, 4)

O=oc.Oc; O(1,2,3,4) O(0,1,0,0)
================================
(-2, 1, 4, -3)

O=oc.Oc; O(1,2,3,4)^-1
================================
(0.0333333333333, -0.0666666666666,
-0.1000000000000, -0.1333333333333)


・oc.Oc クラスを八元数として扱う
oc.Oc(1,2,3,4,5,6,7,8)
================================
(1, 2, 3, 4, 5, 6, 7, 8)

O=oc.Oc; O(1,2,3,4,5,6,7,8)
                    O(0,1,0,0)
================================
(-2, 1, 4, -3, 6, -5, -8, 7)

O=oc.Oc; O(1,2,3,4,5,6,7,8)
                    O(0,0,0,0,1)
================================
(-5, -6, -7, -8, 1, 2, 3, 4)

・八元数の推移律が成り立たない例
O=oc.ClOctonion;(O(0,1) O(0,0,1))
                    O(0,0,0,0,1)
================================
(0, 0, 0, 0, 0, 0, 0, 1)

O=oc.ClOctonion; O(0,1)
        (O(0,0,1) O(0,0,0,0,1))
================================
(0, 0, 0, 0, 0, 0, 0, -1)
```

3 ダイナミック・インポート・パッケージ

Python sf は scipy, sympy, visual, mayavi.mlab パッケージをダイナミックに import しています。これらの package は非常に有用なのですが、scipy, sympy, vpython packege 三つを import しようとすると、import するだけで Atom N280 ネットブックで 2.7 秒掛かってしまいます。mayavi.mlab にいたっとて、これだけで 22 秒掛かってしまいます。Atom N280 ネットブックで Python sf のワンライナーの計算を 1 秒で済ませられているのは、これらのモジュールの import をダイナミックに行わせているからです。

ダイナミック・インポート・パッケージ			
package 名	組込関数	代換名	機能
scipy	sy()	sy	Matlab 相当の数値計算パッケージ
	scipy sub package		
	package	代換名	機能
	optimize	so	特定の条件を満たす変数値を求める
	integrate	si	積分を行う関数群
	linalg	sl	行列の線形処理を行う関数群
	special	ss	特殊関数群
	signal	sg	線形システムを扱う関数群
sympy	ts()	ts	シンボリック数式処理パッケージ
vpython	vs()	vs	表示パッケージ
mlab	mlb()	mlb	mayavi.mlab

表 2 Python sf ダイナミック・インポート・パッケージ

3.1 scipy

scipy パッケージは膨大な数値計算パッケージです。その計算機能は多くのユーザーにとって Matlab に匹敵するほどです。

3.1.1 scipy と numpy

scipy は numpy を包含します。でも Python sf がデフォルトで import しているのは numpy です。scipy は sy() 関数を呼び出したときだけです。

Atom N280 で scipy の import が時間が 1 秒であるのに対し numpy の import 時間が 0.5 秒で済むので、numpy で済ませられ計算は、numpy 側ですませています。

3.1.2 so,si,sl,ss,sg

sy() 関数をよびだすと so:optimize, si:intergrate, sl:linalg, ss:special, sg:signal パッケージも同時に import します。これらは so,si,sl,ss,sg の名前でアクセスできるようになります。scipy を import したあとは、これらの import 時間は Atom N280 でも 0.1 秒もかからないからです。

scipy の機能を説明しはじめと膨大すぎてきりがありません。詳細は「Numpy and Scipy Documentation」http://docs.scipy.org/doc/ を参照ください。

3.2 sympy

sympy は Python でシンボリックな数式処理を可能にするパッケージです。2010 年 8 月現在で 0.67 のバージョンですが、十分に実用になります。

ts() 関数を呼び出すことで sympy がインポートされます。パッケージ名は ts に割り当てます。ts() を実行すると ‘x,‘y,‘z,‘n_, ‘p,‘q,‘t が定義済みの sympy 変数として Python sf 式で使えるようになります。以下のような計算が可能です。

Python sf one-liner

・シンボリック計算

```
ts();ts.log(ts.sqrt(‘x))
================================
log(x**(1/2))

ts();ts.trigsimp(ts.sin(‘x)^2
                    +ts.cos(‘x)^2)
================================
1

ts();ts.trigsimp(ts.sin(‘x+‘y))
================================
sin(x + y)
```

```
ts(); ts.expand((‘x+‘y)^5)
================================
10*x**3*y**2 + 10*x**2*y**3 +
 5*x*y**4 + 5*y*x**4 + x**5 + y**5

ts();ts.limit((‘x^2+1)/(2+‘x+3*‘x^2)
                     , ‘x, ts.inf)
================================
1/3

ts();ts.diff(ts.acos(‘x),‘x)
================================
-1/(1 - x**2)**(1/2)

ts();f = ts.exp(‘x) ts.log(‘x);
                    ts.diff(f,‘x,3)
================================
exp(x)*log(x) - 3*exp(x)/x**2
    + 2*exp(x)/x**3 + 3*exp(x)/x

ts();ts.solve(‘p*‘x^2 + ‘q*‘x + ‘t, ‘x)
================================
[-(q + (-4*p*t + q**2)**(1/2))/(2*p),
  (-q + (-4*p*t + q**2)**(1/2))/(2*p)]

ts();f = ‘x^2 + 432/‘x;ts.solve(
            ts.Eq(ts.diff(f,‘x),0),‘x)
================================
[-3 + 3*I*3**(1/2), 6,
               -3 - 3*I*3**(1/2)]

ts();B = ts.Matrix([[1,2,3],[3,2,1]
                    ,[1,2,1]]); B^-1
================================
[   0,  1/2, -1/2]
[-1/4, -1/4,    1]
[ 1/2,    0, -1/2]
```

sympy の機能を説明しはじめと膨大すぎてきりがありません。詳細は「Welcome to SymPy ’s documentation!」http://docs.sympy.org/index.html を参照ください

4 Python sf の応用例

4.1 元利均等払い

元利均等払いの計算方法は、高校数学の級数の応用問題にすぎません。簡単な計算です。$(x^n-1)/(x-1) == (x^{(n-1)} + x^{(n-2)} + ...x + 1)$ の公式を理解できれば、元利均等払いの計算ができます。

でも、原理が簡単でも、その計算自体は電卓では計算しきれないだけの多くの計算が必要になります。しかし、Python sf ならば、計算量が多いといってもベクトルとして扱えます。Python sf 式より元利均等払いのベクトル計算を電卓でのように扱っていけます。この計算の様子を以下に示します。

年利 5% で 1000 万円の金額を 20 年の元利均等払いで返却することを考えて見ましょう。元本の返却分にも利子が付きます。一回に返す元本を P0 円とするとし、利率 r=1.05, n=20 回とすると $10,000,000 == P0r^{(n-1)} + P0^{(n-2)} + ... + P0 == P0(r^n - 1)/(r - 1)$ とならねばなりません。ですから $P0 == 10,000,000(r - 1)/(r^n - 1)$ となります。

これを Python sf 式で具体的に計算させると次のようになります

Python sf one-liner

```
T, N, r =1e7, 20, 1.05; P0 = T (1-r)/(1-r^N)
================================
302425.871907
```

実際に 302425.871907 を 20 回複利で返却すると下のように 1000 万円になります

Python sf one-liner

```
T, N, r =1e7, 20, 1.05; P0 = T (1-r)/(1-r^N);
                  sum([P0 r^n for n in range(N)])
================================
10000000.0
```

この複利での元本の返却の数列を実際に打ち出してみます。この数列の金額がの順序で元本の 1000 万円が減っていくわけです。

Python sf one-liner

```
T, N, r =1e7, 20, 1.05;
    P0 = T (1-r)/(1-r^N);
    ~[P0 r^n for n in range(N)]
================================
[ 302425.87190691  317547.16550226
  333424.52377737  350095.74996624
  367600.53746455  385980.56433778
  405279.59255467  425543.5721824
  446820.75079152  469161.7883311
  492619.87774765  517250.87163504
  543113.41521679  570269.08597763
  598782.54027651  628721.66729033
  660157.75065485  693165.63818759
  727823.92009697  764215.11610182]
```

```
---- ClTensor ----
```

一回ごとの金利は、上のベクタから計算される、残っている元本に年間の金利を掛けたものになります。この金利分を毎回返却することになります。Python sf 式では、この金利の数列は下のように書けます。

Python sf one-liner

```
T, N, r =1e7, 20, 1.05;
    P0 = T (1-r)/(1-r^N);
    vct=~[P0 r^n for n in range(N)];
    [sum(vct[k:]) (r-1) for k in range(N)]
==============================
[500000.00000000047, 484878.70640465483,
 469001.34812954185, 452330.12194067327,
 434825.3344423612,  416445.30756913364,
 397146.27935224457, 376882.2997245111,
 355605.12111539097, 333264.08357581485,
 309805.99415925989, 285175.00027187716,
 259312.45669012531, 232156.78592928589,
 203643.33163040446, 173704.20461657897,
 142268.12125206221, 109260.2337193196,
 74601.951809939885, 38210.755805091154]
```

一回ごとの元本返却分のベクタと、金利のベクタを足し合わせてみましょう。Python sf 式では下のように書けます。

Python sf one-liner

```
T, N, r =1e7, 20, 1.05;
    P0 = T (1-r)/(1-r^N);
    vct=~[P0 r^n for n in range(N)];
    [sum(vct[k:]) (r-1) for k in range(N)]
                                    + vct
==============================
[ 802425.87190691  802425.87190691
  802425.87190691  802425.87190691
  802425.87190691  802425.87190691
  802425.87190691  802425.87190691
  802425.87190691  802425.87190691
  802425.87190691  802425.87190691
  802425.87190691  802425.87190691
  802425.87190691  802425.87190691
  802425.87190691  802425.87190691
  802425.87190691  802425.87190691]
---- ClTensor ----
```

一見では不思議な感じもするのですが、毎回の返却金額が全て一致しました。これが均等払いと言われる所以です。

1000 万円を 5

Python sf one-liner

```
T, N, r =1e7, 20, 1.05; P0 = T (1-r)/(1-r^N);vct=~[P0 r^n
==============================
16048517.4381
```

途中で借り換えをするときには、元利均等払いのベクトルの残りの部分を足し合わせたものが残っている未返却の元本になります。この金額で、新しい金利と、返却回数で同じことを計算することになります。

4.2 特殊相対性理論

特殊相対性理論は、時間と空間の認識を大きく変える理論です。自然観・人生観を変える理論だとさえいえます。でも、ブルー・バックスなどの通俗書を読んでも、自然観・人生観を変えるような特殊相対性理論の理解には至れません。自分で特殊相対性理論を租借する作業を行えないからです。そのためには、実際に特殊相対性理論を適用してみて、実際に発生する現象として特殊相対性理論を体得する必要があるからです。でも そんな計算は簡単にはできません。

いや Python sf ならば、特殊相対性理論の計算は簡単です。四次元時空の線形代数の計算にすぎません。多くが Python sf 式のワンライナーで計算できてしまいます。以下、Python sf で相対性理論をどのように扱っていくのか見てみましょう。

4.2.1 特殊相対性理論とビッグ・バン

特殊相対性理論というと列車と光の思考実験が有名でよく使われるのですが、私はビッグ・バンを使ったほうが、もっと直感的な説明が可能だと思います。ですので、ビッグ・バンを題材に特殊相対性理論を展開してみます。

現在は宇宙がビッグ・バンが始まってから 137 億年が経過したとされています。このビッグ・バンを認めるならば、特殊相対論を認めないと辻褄が合いません。

下の絵は、ビッグ・バンのときに発生した光が現在の我々の所に到達する様子の模型です。宇宙全体を球面で現し、左したのビッグ・バンに発生したひかりが右上の現在の我々の位置に到達する様子を現しています。三次元宇宙を球の二次元表面で代用しているわけです。

この図より、我々が望遠鏡で見ている宇宙の端っこ

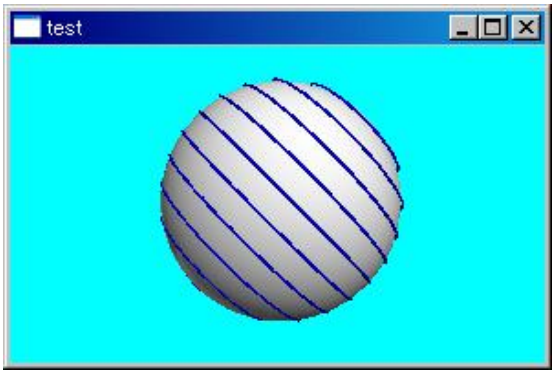

図 3　big bang の様子

はビッグ・バンの一点を見ていることが分かります。ビッグ・バンの光が我々に位置に到着するまでの時間は 137 億年であり、その光が移動した距離は 137 億光年です。この光が我々の位置 a に到着するまでの時間と時刻を ta, xa とすると $-(cta)^2 + xa^2 == 0$ であることが分かります。

さて、我々の位置は宇宙のなかの特別の位置ではありません。宇宙の別の銀河の位置 b でも同様にビッグ・バンの光が届いていると思うべきでしょう。その時刻と時間を tb, xb とすると $-(ctb)^2 + xb^2 == 0$ でなければ、なりません。そこでも我々と同様なビッグ・バンの光のリングが到着していなければなりません。でも我々の位置 a で同心円に見えている光のリングが、位置 b では同心円の光のリングになっていません。さあ、ta,xa と tb, xb の関係をどう考えましょうかと言う訳です。

ここで時間と位置は宇宙が膨張し続けている時空の中での ta,xa と tb, xb です。この二つの位置の ta,xa と tb,xb はどのように関係していると思うべきなのでしょうか。ただ一つの時刻があって ta と tb が同時に進んでいるとすることは、ビッグ・バンの光リングの模型をみただけで無理だと思えるでしょう。ただしビッグ・バンの光のリング $-(cta)^2 + xa^2 == 0 == -(ctb)^2 + xb^2$ は満たされねばなりません。

4.2.2　光速は 1 で時間は虚数

時空の物理を考えることは、時空をベクトルとして一つの物理量として考えることです。ならば時間と空間の同じ物理単位で表すべきです。でなければ、時空の幾何学を考えることはできません。

一方で Hawkin と は「ビッグバン以前の時間は虚時間だった」と言っています。これは、現在の時間を実数としているためです。逆に現在の時間を虚数だとみなせば「ビッグバン以前の時間は実時間だった」と言えます。このほうが、ビッグバン以前は時空が一体であり、互いを行き来できたことになります。ビッグ・バンは実数だった時間が虚数に飛び跳ねたと解釈できます。時間と空間を互いに行き来できなくなったが、時空の Lorentz 変換の関係は残っていると解釈できます。時間と空間を互いに行き来できなくなったことにより、時間が物理現象のパラメータの働きに限定されてしまうようになったと解釈できます。

多くの物理現象で時間の自乗がマイナスになる:時間の虚数性が見られます。$\partial_x^2\psi+\partial_y^2\psi+\partial_z^2\psi-\partial_t^2\psi == 0$ の波動方程式の時間項にマイナスが付くのは単なる偶然とは思えません。特殊相対論でも Minkovsky 時空では時間を虚数で表します。

このように時空が一体みなすとき、時間にも空間と同じ m‘ の単位で表記したくなります。Python sf では mLght の単位を次のように定義してあります。1mLght‘ は 1m‘ を光が進む時間であり、SI 単位系で表すと 3.33564095198152ns‘ になります。

```
Python sf one-liner
ts(); mLght‘
==============================
I*m‘

mLght‘
==============================
1j

ts();1 mLght‘/c‘
==============================
3.33564095198152e-9*I*s‘
```

時空ベクトルを mLght‘, m‘ の単位の数値で表記するとき光速度は無次元の 1 になります。$-(cta)^2 + xa^2 == 0 == -(ctb)^2 + xb^2$ は $-ta^2 + xa^2 == 0 == -tb^2 + xb^2$ になります。

4.2.3 Lorentz 変換

Lorentz 変換 L を $[tb, xb] = L[ta, xa]$ となる線形変換だとしましょう。b 点は光の半分の速度で xa 方向に走っているとしましょう。時間が 1mLght だけ進んだとき b 点は 0.5 m‘ だけ進んでいることになります。この 1mLght‘, と 0.5m‘ が作る三角形の原点側の角度は次のように虚数値になります。

```
Python sf one-liner
ts();arctan( 0.5m‘/(1mLght‘) )
================================
-0.549306144334j
```

この虚数回転角度で Lorentz 変換を行うことは、虚数回転角だけ座標系を回転する変換になります。回転変換ですから $-ta^2 + xa^2 == -tb^2 + xb^2$ になります。

```
Python sf one-liner
θ = arctan(0.5m‘/(1mLght‘));
    L2:=~[[cos(θ), -sin(θ)],
                [sin(θ),cos(θ)]];
    L=L2.real;
    L[0,1]=L[1,0]=L2[0,1].imag;
    L2R:=L
================================
[[ 1.15470054  0.57735027]
 [ 0.57735027  1.15470054]]
---- ClTensor ----

L=:L2R;N=10;mt = kzrs(N,N,object);
for idx in mrng(N,N):mt[idx]=L idx;
LorentzCoordinate:=mt
データがおきすぎるので略

mt =:LorentzCoordinate;
for k in range(10):plotTrajectory(mt[k,:])
and plotTrajectory(mt[:,k],color = yellow)
```

4.3 圏論

圏論は abstract nonsence と言われるほど抽象化されている理論です。そんな理論を理解するために教科書を何回も写経を繰り返すのは効率が悪すぎます。

抽象的な理論を理解する手っ取り早い方法は、その具体例を構成してみることです。圏の定義が与えられたら、それを満たす具体例を作ってみることです。圏の定義を満たす／満たさないの境界のような実例を作れたら、圏の定義を深く理解したと言えます。

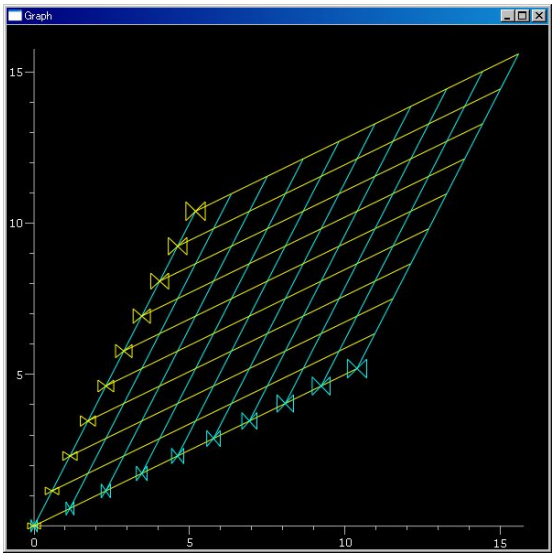

図 4 Lorentz 座標変換

Python sf を使えば、この具体例を作る作業が簡単になります。ワン・ライナーで具体例を記述できます。

4.3.1 圏の定義

「Basic Category Theory for Computer Scientists - B. Pierce」の圏の定義を抜き出します。英語ですが、これぐらい抽象的な定義だと英語の方が分かりやすいと思います。冠詞などで微妙なニュアンスを裏付けられるからです。

4.3.1.1 Definition A category C comprises

1. a collection of objects;
2. a collection of arrows (often called morphisms);
3. operations assigning to each arrow f an object dom, its domain, and an object cod f, its codomain (we write $f : A \to B$ to show that dom f = A and cod f = B; the collection of all arrows with domain A and codomain B is written C(A, B));
4. a composition operator assigning to each pair of arrows f and g, with cod = dom g, a composite arrow $gof : dom\, f \to cod\, g$, satisfying the following associative law:

for any arrows $f : A \to B$, $g : B \to C$,
and $h : C \to D$ (with A, B, C, and D
not necessarily distinct),
h o (g o f) = (h o g) o f;

5. for each object A, an identity arrow
$id_A : A \to A$ satisfying the following identity law:
for any arrow $f : A \to B$,
$id_A \, o \, f = f and f \, o \, id_A = f$

このようなカテゴリーの定義を objects を型と読み替え arrow を副作用を持たない関数と読み替えてやれば、カテゴリーの概念をプログラミングに適用できます。

上のカテゴリーの定義で一番分かりにくいのは 4 番目と 5 番目だと思います。4 番目の h o (g o f) = (h o g) o f は関数合成の意味では当然です。そう思うと、公理にする必然性が理解できなくなります。5 番目は恒等関数の存在は自明です。$id_A \, o \, f = f and f \, o \, id_A = f$ も関数合成を仮定すれば当然です。これの疑問を解消できたのは八元数で monoid category を作ってみたときでした。

結論から言うと「h o (g o f) = (h o g) o f」の意味が関数合成の意味だと限っていないのでした。もちろん、「h o (g o f) = (h o g) o f」を関数合成の意味としてカテゴリーを構成してもかまいません。そのときは、四番目と五番目の公理は自明です。

でも「h o (g o f) = (h o g) o f」の意味が関数合成の意味ではないことがありえます。カテゴリの公理が要求しているのは arrow f の cod と arrow g の dom が一致したとき f o g の arrow が作れることだけです。これを関数合成の意味で作れとは言っていないのです。

八元数の monoid カテゴリーを考えて見ましょう。八元数の要素 a,b に対して morphing fa, fb が次のように作れます。ここで a,b を八元数の部分集合の四元数からとってきます。oc.Oc(1,2,3,4), oc.Oc(5,6,7,8) とでもしておきましょう。

Python sf one-liner

```
O=oc.Oc; a,b=O(1,2,3,4),O(5,6,7,8);
    fa, fb = λ x:a x, λ x:b x
```

```
==============================
(<function <lambda> at 0x00BB0730>,
 <function <lambda> at 0x00BB0230>)
```

ここで faob ≡ fa o fb を関数合成の意味で定義すれば、前にも述べたように、公理の 4 番目と 5 番目は自動的に満たされます。関数合成の性質が、それを保障します。

Python sf one-liner

```
O=oc.Oc; a,b=O(1,2,3,4),O(5,6,7,8);
    fa, fb = λ x:a x, λ x:b x
    faob = λ x: fa( fb(x) )
==============================
```

でも fa(fb(x)) == fa(b x) == a b x ですから、q= a b としたとき faob = λ x: q x (==λ x: (a b) x) で定義することができます。プログラミングで言えば、monoid の性質を利用して、関数合成で最適化してやったことになります。この最適化を使った関数合成でも fc の c を四元数の範囲から持ってくれば、4 番目の公理を満たします。四元数は推移律を満たすから、最適化しても問題ないからです。

でも、c を推移律を満たさない 八元数の要素から持ってくると、最適化によって作った合成関数は 4 番目の公理を満たさなくなります。例えば c=O(9,8,7,6,5,4,3,2) とすると、下のように最適化した合成関数は下のように推移律を満たしません。

Python sf one-liner

```
O=oc.Oc; a,b=O(1,2,3,4),O(5,6,7,8);
    c=O(9,8,7,6,5,4,3,2);
    print a (b c); (a b) c
(-990, -360, -30, -300, -486, -72, -198, 36)
==============================
(-990, -360, -30, -300, -486, -168, -102, 84)
```

八元数であっても「h o (g o f) = (h o g) o f」の意味を関数合成の意味にしておけば、八元数のオブジェクトの上で monoid カテゴリになります。

八元数の monoid カテゴリは g o f の定義で最適化を施すか否かで、「圏の定義を満たす／満たさない」のどちらにも転びます。これでカテゴリの境界のような実例を作れたと思います。これによりカテゴリの公理を、より深く理解できるようになると思います。如何でしょうか。

4.3.2 Functor の定義

「Basic Category Theory for Computer Scientists - B. Pierce」に書いてある Functor の定義を抜き出します。

4.3.2.1 Definition

Let C and D be categories. A functor $F : C \to D$ is a map taking each C-object A to a D-object F(A) and each C-arrow $f : A \to B$ to a D-arrow $F(f) : F(A) \to F(B)$, such that for all C-objects A and composable C-arrows f and g

1. $F(id_A) = id_{F(A)}$
2. F(g o f) = F(g) o F(f).

分かり難い定義です。こんな定義で分かるのは人間ではないでしょう。プログラマーならばファンクタの具体例として STL の list¡T¿, vector¡T¿ などを思い浮かべると分かり易くなると思います。要はカテゴリ C とは直交した機能を持つ list, vector などを組み合わせることを主張しています。

一般的には Functor を考える場合、カテゴリ C に対し、カテゴリ D は upper world になっています。C==int だとしたと D は list¡int¿ や vector¡int¿ などの upper world になっています。定義の 2 番目 F(g o f) = F(g) o F(f) が int と list,vector が独立していることを主張しています。

Python sf で関数の加減乗除べき乗算を可能にした ClAF クラスもファンクタだと思っていました。ClAF を被せて加減乗除べき乗算を可能にすることは関数の dom,cod が float であるからであり、関数の他の性質とは無関係だからです。この ClAF を被せる操作は何重にも可能であり Monad の Functor T に対応させられると思っていました。Monad の自然変換 μ は ClAF の被せを一段落とす自然変換に対応させられると思っていました。

でも Python sf で ClAF がファンクタである構造関数を作ろうとして行き詰まりました。どうも ClAF を被せることはファンクタではないようです。