

研究報告

生命医科学系の学生のための Python 言語と SymPy ライブラリを用いて科学法則を記述する数式を扱う方法

新谷正嶺

中部大学生命健康科学部 生命医科学科

要 旨

生命医科学系の分野では、数式で記述される科学法則の理解や扱いが苦手な学生さんが少なくない。しかし、科学の発展に伴い、生命の仕組みを理解するためにも、医療技術で用いられる装置の原理や計測データの意味を理解するためにも、数式で記述される科学法則と向き合うことは避けられなくなってきている。本研究報告では、紙と鉛筆を用いた数式計算の技術や計算力に自信が無くても、少数行のプログラミングを行うだけで、数式を比較的自由に操作する方法を教授する。インターネットに接続するパソコンさえあれば、高価な数学計算用ソフトを購入せずとも同様の操作が行えるよう、手法は Python 言語と SymPy ライブラリを用いる方法に絞って説明する。教授する数式操作法は、数式に数値を代入する方法、微分・積分・極限計算、方程式や微分方程式を解く方法、そして数式の可視化法である。これらの方法を扱えるようになることで、学生の皆さんが科学法則を記述する数式に慣れ親しみ理解を深められることを願っている。

1. はじめに

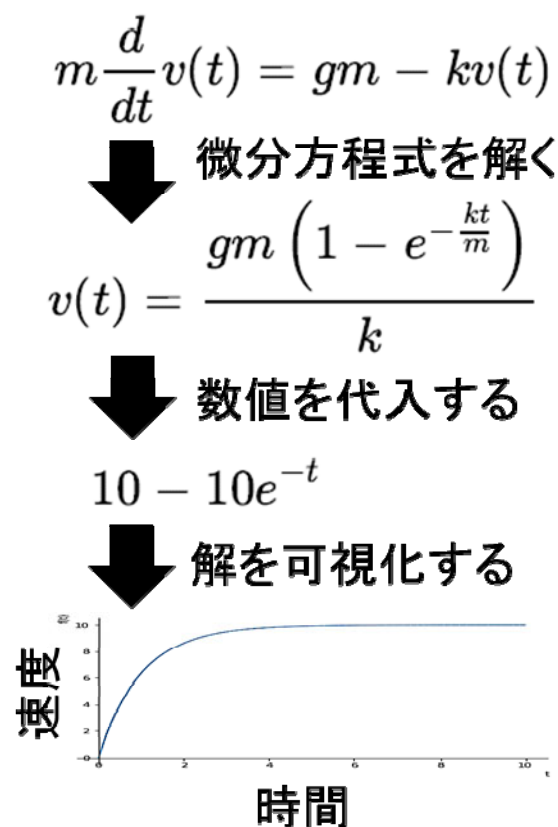


図 1. 本論文で紹介する数式(微分方程式)操作の概略

左図は著者が本研究報告で紹介する Python 言語および SymPy ライブラリを用いた方法で可能になる数式操作の具体例を紹介した概略図である。具体的には、雨滴が地表に落下する際の速度の時間変化を記述する運動方程式を速度に関して解き、その結果を可視化した。初めの運動方程式における記号を説明すると、雨滴の落下速度が時間 t の関数 $v(t)$ で、雨滴の質量が定数 m 、雨滴を地面に引き寄せる重力加速度が定数 g 、雨滴の落下速度 v に比例した空気抵抗力の比例定数が k となっている。この運動方程式を紙と鉛筆を使って解く場合、以下のような変数分離形の形に式を変形し、両辺を積分するという解き方で解を求めていく。

$$\frac{1}{-kv + mg} dv = \frac{1}{m} dt$$

左図の 2 段目の解は、初期条件を $t=0$ で $v(0)=0$ とした場合の解である。3 段目は、定数 m 、 g 、 k に数値を代入した $m=1$ 、 $g=10$ 、 $k=1$ という値を代入した後の解 $v(t)$ である。T に 0 から 10 までの値を代入した場合の $v(t)$ の値の変化を示したのが最後の 4 段目のグラフである。Python 言語と SymPy を用いることでこのような数式操作を非常

に簡単に行うことが出来る。前置きはこれくらいに、早速この Python 言語と SymPy ライブラリの活用法を紹介する[1,2]。

2. 方法

2.1 Python 言語の環境構築

Python 言語は Python 公式サイトから無料でダウンロード出来る[2]。しかし、プログラム言語の環境構築に慣れていない場合、Anaconda をインストールすることを推奨する[3]。インストール後、Anaconda Navigator というプログラムを起動すると、マウスを用いた簡単な操作で Python 言語の開発環境(Jupyter Notebook など)の構築が完了する。Python 言語を使用する際に有用なライブラリのインストールも簡単に行える。Anaconda Navigator を起動した際に左側に表示される項目の上から 2 番目の Environments を選択すると、インストールしているライブラリや未だインストールしていない公開ライブラリの一覧が表示される。ここで sympy という名前のライブラリを探して選択し、Apply ボタンを押すことで SymPy のインストールが行われ、SymPy が使用可能になる。あるいは以下の URL のサイトの活用を推奨する。以下のサイトは、Google 社が提供するクラウドの Jupyter Notebook 環境で、細かい設定をすることなく、無料ですぐに使用出来る(<https://colab.research.google.com/notebooks/intro.ipynb>) [4]。利用するには Google 社の Google account を作成し、ログインしておく必要がある。このクラウド環境では最初から SymPy はインストールされている。

なお、Python 言語そのものの基本については著者が以前作成した入門用カリキュラム等を参考にして欲しい[5]。また、Python 言語を用いているわけではないが、「実験」とプログラミングによる「データ解析や数理モデルシミュレーション」を組み合わせるとどんな研究が可能になるのか、その具体的なイメージをつかみたい場合、最近の筆者の論文が一つの参考例になるので引用文献として紹介する[6]。

2.2 Python 言語と SymPy ライブラリのバージョン

本研究報告を作成した際の Python 言語と SymPy ライ

ブラリのバージョンを記載して置く。Python 言語のバージョンは 3.7.9、SymPy ライブラリのバージョンは 1.6.2 である。また、本研究報告で作成するプログラムはバージョン 6.1.3 の Jupyter Notebook 環境を用いる。

3. 結果

3.1 変数の生成と数式操作

ライブラリを使用するためには、import という操作を行う必要がある。そのためには、「import ライブラリ名」というコードを記述する。この際、続けて「as 略称」と記述することで、ライブラリを利用する際、その略称を利用出来るようになる。

```
In [1]: import sympy as smp # sympyのインポート。
```

次に、SymPy の関数 `init_printing()` を以下のように呼び出すことで、表示される数式の見た目を整える。`init_printing()` は SymPy の数式をどのように出力するのかを決める関数である。ここでは、引数無しの呼び出しで `init_printing()` のデフォルトの設定に整える。

```
In [2]: smp.init_printing()
```

SymPy では、以下のように、数式で用いる記号を定義することが出来る。一つの記号を定義する場合は SymPy の `Symbol()` 関数を用いる。() 中の文字が数式として表示される記号である。変数と表示される記号を別の表記とすることも出来るが、混乱の元なので、基本的に一致させることを勧める。

```
In [3]: x = smp.Symbol("x")
```

```
In [4]: x + x + 1
```

```
Out[4]: 2x + 1
```

上記の In[3] で `x` を記号の `x` として定義した。その結果、In[4] で記述した「`x+x+1`」は Out[4] に出力されているように「`2x+1`」と認識される。

複数の記号を定義する場合は、`Symbol()` 関数を複数回用いるのではなく、`symbols()` 関数を用いることも出来る。

```
In [5]: a, b, c = smp.symbols("a, b, c")
```

Symbol()関数や symbols()関数を用いて複数の記号を定義したら、以下のようにその記号を組み合わせた数式を定義することが出来る。

```
In [6]: y = a * x ** 3 + b * x ** 2 + c * x + 1
```

```
In [7]: y # 変数yの中身を出力する。
```

```
Out[7]:  $ax^3 + bx^2 + cx + 1$ 
```

作成した数式の記号に、subs()メソッドを用いて以下のように値を代入することが出来る。

```
In [8]: y_s = y.subs([a:1, b:3, c:3])
```

```
In [9]: y_s # 変数y_sの中身を出力する。
```

```
Out[9]:  $x^3 + 3x^2 + 3x + 1$ 
```

作成した数式に対して、factor()関数を用いることで因数分解を行える。

```
In [10]: y_f = smp.factor(y_s)
```

```
In [11]: y_f
```

```
Out[11]:  $(x + 1)^3$ 
```

expand()関数を用いることで数式の展開も行える。

```
In [12]: smp.expand(y_f)
```

```
Out[12]:  $x^3 + 3x^2 + 3x + 1$ 
```

simplify()関数は、数式を SymPy が扱える簡素な表現に変換してくれる関数である。計算過程の重複部分のある数式の整理などに使うことが出来る。

```
In [13]: y_c = (x ** 2 + 12 * x + 35) / (x + 5) + 3
```

```
In [14]: y_c
```

```
Out[14]:  $3 + \frac{x^2 + 12x + 35}{x + 5}$ 
```

```
In [15]: smp.simplify(y_c)
```

```
Out[15]:  $x + 10$ 
```

3.2 数列と極限操作

数列を扱うには、sequence()関数を用いる。ここでは、

極限が自然対数の底であるネイピア数になる等差数列を求める。

```
In [16]: n = smp.Symbol("n")
napier_sequence = smp.sequence((1 + 1 / n) ** n)
```

```
In [17]: napier_sequence # 中身の確認。
```

```
Out[17]:  $\left[1, 2, \frac{9}{4}, \frac{64}{27}, \dots\right]$ 
```

```
In [18]: napier_sequence.formula # 数列の数式表記。
```

```
Out[18]:  $\left(1 + \frac{1}{n}\right)^n$ 
```

limit()メソッドで数列や関数の極限を求めることが出来る。limit(計算する式, 変化させる変数, 変数が向かう極限)。下式の smp.oo は無限(∞)を表現している。

```
In [19]: smp.limit(napier_sequence.formula, n, smp.oo)
```

```
Out[19]:  $e$ 
```

極限が自然対数の底であるネイピア数「e」になることが確認できた。なお、極限を求める計算をすぐに実行するのではなく、極限の計算を数式で表現したうえで、その計算を後で実行する場合、以下のように Limit()メソッドと doit()メソッドを用いる。

```
In [20]: napier_number = smp.Limit(napier_sequence.formula, n, smp.oo)
```

```
In [21]: napier_number # 中身の確認。
```

```
Out[21]:  $\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n$ 
```

```
In [22]: napier_number.doit() # 計算の実行。
```

```
Out[22]:  $e$ 
```

summation()関数で、数列のある区間の総和を求めることが出来る。summation(用いる数列の式, (変数, 始めの数, 最後の数))。

```
In [23]: n_sequence = smp.sequence(1 / n ** 2)
n_sequence.formula
```

```
Out[23]:  $\frac{1}{n^2}$ 
```

```
In [24]: smp.summation(n_sequence.formula, (n, 1, 5))
```

```
Out[24]:  $\frac{5269}{3600}$ 
```

範囲の指定に無限(∞)を表す記号「oo」を用いることで、総和の極限を求めることも出来る。

```
In [25]: smp.summation(n_sequence.formula, (n, 1, smp.oo))
```

```
Out[25]:  $\frac{\pi^2}{6}$ 
```

product()関数で、数列のある区間の総乗を求めることが出来る。product(用いる数列の式, (変数, 始めの数, 最後の数))。

```
In [26]: wallis_sequence = smp.sequence((2 * n / (2 * n - 1)) * (2 * n / (2 * n + 1)))
wallis_sequence.formula
```

```
Out[26]:  $\frac{4n^2}{(2n-1)(2n+1)}$ 
```

```
In [27]: smp.product(wallis_sequence.formula, (n, 1, 5))
```

```
Out[27]:  $\frac{65536}{43659}$ 
```

```
In [28]: smp.product(wallis_sequence.formula, (n, 1, smp.oo))
```

```
Out[28]:  $\prod_{n=1}^{\infty} \frac{4n^2}{(2n-1)(2n+1)}$ 
```

上記の無限積はウォリス積と呼ばれている。表記は正しいが、その総乗の極限值「 $\pi/2$ 」は求められていない。

3.3 方程式

solve()関数を用いることで、方程式を解くことが出来る。solve(解きたい式, 解く対象の変数)。この関数は入力した式=0と仮定して計算を行う。

```
In [29]: y = a * x ** 2 + b * x + c
```

```
In [30]: y
```

```
Out[30]:  $ax^2 + bx + c$ 
```

```
In [31]: smp.solve(y, x)
```

```
Out[31]:  $\left[ \frac{-b + \sqrt{-4ac + b^2}}{2a}, -\frac{b + \sqrt{-4ac + b^2}}{2a} \right]$ 
```

solve()関数は、入力の式として、方程式のリストやタプルを渡すことで、連立方程式を解くことも出来る。

```
In [32]: x, y, z = smp.symbols("x, y, z")
y1 = x + y - z - 5
y2 = 2 * x - y + z + 1
y3 = -x + 3 * y + 2 * z
```

```
In [33]: y1
```

```
Out[33]:  $x + y - z - 5$ 
```

```
In [34]: y2
```

```
Out[34]:  $2x - y + z + 1$ 
```

```
In [35]: y3
```

```
Out[35]:  $-x + 3y + 2z$ 
```

```
In [36]: smp.solve((y1, y2, y3))
```

```
Out[36]:  $\left\{ x: \frac{4}{3}, y: \frac{26}{15}, z: -\frac{29}{15} \right\}$ 
```

y1=0, y2=0, y3=0 の連立方程式の解となる x, y, z の値が求められた。Eq()関数を用いることで、左辺も右辺も0ではない等式の方程式を定義することが出来る。

```
In [37]: left = x ** 2 + x
right = 2 * x + 4
smp.Eq(left, right)
```

```
Out[37]:  $x^2 + x = 2x + 4$ 
```

```
In [38]: smp.solve(smp.Eq(left, right), x)
```

```
Out[38]:  $\left[ \frac{1}{2} - \frac{\sqrt{17}}{2}, \frac{1}{2} + \frac{\sqrt{17}}{2} \right]$ 
```

3.4 微分と積分

diff()関数を用いることで、微分の計算が行える。diff(微分する式, 微分する変数)。

```
In [39]: x, y = smp.symbols("x, y")
f = smp.sin(x) * smp.cos(2*y)
```

```
In [40]: f
```

```
Out[40]:  $\sin(x) \cos(2y)$ 
```

```
In [41]: smp.diff(f, x)
```

```
Out[41]:  $\cos(x) \cos(2y)$ 
```

diff()関数の第2引数以降に、複数の微分したい変数を並べることで、順に微分を行える。

```
In [42]: smp.diff(f, x, x) # f式をxで2階微分する。
```

```
Out[42]:  $-\sin(x) \cos(2y)$ 
```

```
In [43]: smp.diff(f, x, y, y) # xで微分後, yで2回微分。
```

```
Out[43]:  $-4 \cos(x) \cos(2y)$ 
```

微分をすぐに実行するのではなく、微分を数式で表現したうえで、その計算を後で実行する場合、以下のように `Derivative()` メソッドと `doit()` メソッドを用いる。

```
In [44]: f_d = smp.Derivative(f, x, y)
```

```
In [45]: f_d
```

```
Out[45]:  $\frac{\partial^2}{\partial y \partial x} \sin(x) \cos(2y)$ 
```

```
In [46]: f_d.doit()
```

```
Out[46]:  $-2 \sin(2y) \cos(x)$ 
```

`Out[45]` の微分表記は、2 変数で微分しているのので、微分記号が偏微分を意味する ∂ になっている。

`subs()` メソッドを使うことで、結果に数値を代入することも可能である。

```
In [47]: f_d.doit().subs([x:0, y:smp.pi/4])
```

```
Out[47]:  $-2$ 
```

`integrate()` 関数を用いることで、積分の計算も可能である。`integrate(積分したい式, 変数)`。なお、この関数の計算では、計算結果に積分定数はつかない。

```
In [48]: x = smp.Symbol("x")
f = (3 * x + 1) ** 2 - 4
```

```
In [49]: f
```

```
Out[49]:  $(3x + 1)^2 - 4$ 
```

```
In [50]: smp.integrate(f, x)
```

```
Out[50]:  $3x^3 + 3x^2 - 3x$ 
```

与える変数を(積分する変数、積分範囲の最初の値、積分範囲の最後の値)のタプルにすることで、定積分を実行できる。

```
In [51]: smp.integrate(f, (x, 0, 2))
```

```
Out[51]:  $30$ 
```

積分をすぐに実行するのではなく、積分を数式で表現したうえで、その計算を後で実行する場合、以下のように `Integral()` メソッドと `doit()` メソッドを用いる。

```
In [52]: f_i = smp.Integral(f, (x, 0, 2))
```

```
In [53]: f_i
```

```
Out[53]:  $\int_0^2 ((3x + 1)^2 - 4) dx$ 
```

```
In [54]: f_i.doit()
```

```
Out[54]:  $30$ 
```

ガウス分布(正規分布)に対する $-\infty$ から ∞ までの範囲の積分をガウス積分と呼ぶ。このガウス積分は物理学や統計学で良く出てくる数式である。このガウス積分も以下のように計算出来る。

```
In [55]: f_g = smp.Integral(smp.exp(- x ** 2), (x, -smp.oo, smp.oo))
```

```
In [56]: f_g
```

```
Out[56]:  $\int_{-\infty}^{\infty} e^{-x^2} dx$ 
```

```
In [57]: f_g.doit()
```

```
Out[57]:  $\sqrt{\pi}$ 
```

3.5 微分方程式

微分方程式を扱う場合、以下のように `Function()` と `diff()` を使用して方程式を設定し、`dsolve()` 関数を用いて解を求める。ここでは、速度 v に比例した空気抵抗 k を受けて落下する質量 m の雨粒の運動方程式を解いて、雨粒の速度の時間変化を表す数式を求める。なお、重力加速度を g とする。

```
In [58]: v = smp.Function("v")
t, m, g, k = smp.symbols("t, m, g, k")
```

```
In [59]: vt = v(t)
vt
```

```
Out[59]:  $v(t)$ 
```

```
In [60]: left = m * smp.diff(vt, t)
right = m * g - k * vt
smp.Eq(left, right)
```

```
Out[60]:  $m \frac{d}{dt} v(t) = gm - kv(t)$ 
```

`In[58]` の 1 行目で速度 v を関数として定義している。そして、2 行目で時間を t 、雨粒の質量を m 、重力加速度を

g、速度に比例する空気抵抗の値を k として用いるべく、記号として定義している。そして In[59]で雨粒の速度 v を時間 t の関数 $v(t)$ と定義した。In[60]で、これらの関数と記号を用いて、運動方程式の左辺である「質量と加速度(速度の時間微分)の積」が、運動方程式の右辺である「質量と重力加速度の積から空気抵抗と速度の積の空気抵抗力を引いたもの」に等しいとして運動方程式を作成した。

以下のように `dsolve()` 関数を用いて解を求める。

```
In [61]: ans = smp.dsolve(smp.Eq(left, right), vt)
          smp.simplify(ans) # 数式を簡素化する。
```

```
Out[61]:
```

$$v(t) = \frac{gm + e^{\frac{k(C_1 - t)}}{m}}{k}$$

初期条件を与えるには、`dsolve()` 関数に以下のように引数を渡す。ここでは、 $t=0$ で $v=0$ という初期条件を与える。

```
In [62]: ans_i = smp.dsolve(smp.Eq(left, right), vt, ics = [v(0):0])
          smp.simplify(ans_i) # 数式を簡素化する。
```

```
Out[62]:
```

$$v(t) = \frac{gm(1 - e^{-\frac{kt}{m}})}{k}$$

雨粒の速度は、重力に引っ張られる力と、速度依存の空気抵抗力が釣り合うところで速度増加が無くなり、終端速度に至る。上記の式から、 t が大きくなると指数関数「 $\exp(-kt/m)$ 」が 0 に近づいていき、雨粒の速度 v が終端速度「 gm/k 」となることが確認できる。

3.6 数式のグラフ表示

SymPy にはグラフ表示で可視化したい方程式を引数に与えることで、そのグラフを描画してくれる関数群が備わっている。関数群の中で最もシンプルな `plot()` 関数を紹介する。この `plot()` 関数は、 x 軸と y 軸の 2 軸で、 x 軸に対応させた 1 変数の値を変えていった際の y 軸に対応させた変数の値の変化を描画する。表示させる数式は、先ほどの 3.5 節の最後に紹介した、雨粒の速度の時間変化の数式とする。時間 t を x 軸、速度 v を y 軸に対応させて描画する。

Out[62]の $v(t)$ の解(方程式の右辺(rhs))を変数 $v_$ に代入する。

```
In [63]: v_ = smp.simplify(ans_i).rhs
          v_
```

```
Out[63]:
```

$$\frac{gm(1 - e^{-\frac{kt}{m}})}{k}$$

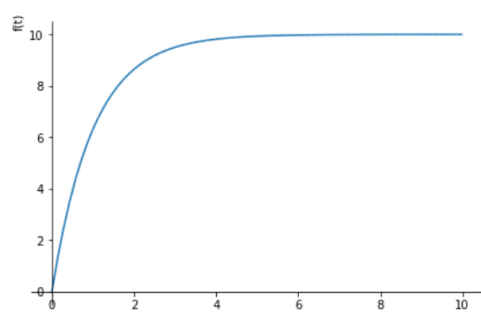
グラフ表示の為、値を未だ定めていない定数 g 、 m 、 k に適当な数値を代入する。

```
In [64]: v_ = v_.subs([g:10, m:1, k:1])
          v_
```

```
Out[64]: 10 - 10e-t
```

`plot()` 関数は、第 1 引数にグラフ化したい数式、第 2 引数に(x 軸の変数、表示範囲の最初の値、表示範囲の最後の値)を与えることが出来る。第 2 引数で表示の範囲を指定しないと、 x 軸の値が -10 から 10 までの範囲を描画する。

```
In [65]: smp.plot(v_, (t, 0, 10))
```



```
Out[65]: <sympy.plotting.plot.Plot at 0x26b9f4b9288>
```

これで、初速度 0 の雨粒が重力によって落下速度を上げていき、しかし速度依存の空気抵抗力の影響を受けて落下速度が終端速度に落ち着いていく様子の可視化が出来た。

次に復元力 k のばねを引っ張ったときに生じる単振動を記述する運動方程式を解き、その結果をグラフに描画する。

```
In [66]: x = smp.Function("x")
          t, k = smp.symbols("t, k", positive="True")
```

```
In [67]: xt = x(t)
          left = smp.diff(xt, t, 2)
          right = - k * xt
          eq_sp = smp.Eq(left, right)
          eq_sp
```

```
Out[67]:
```

$$\frac{d^2}{dt^2}x(t) = -kx(t)$$

第2引数に `positive="True"` と記述することで、記号 t , k は正の値であると指定した。まず、`dsolve()`関数で一般解を求める。

```
In [68]: smp.dsolve(eq_sp, xt)
```

```
Out[68]:  $x(t) = C_1 \sin(\sqrt{k}t) + C_2 \cos(\sqrt{k}t)$ 
```

次に、時刻 $t=0$ で自然長からの変位量 $x(0)=1$ に伸ばされており、初速は0という初期条件を代入し、その場合の解を求める。

```
In [69]: x_ = smp.dsolve(eq_sp, xt, ics = [x(0):1, smp.diff(xt,t).subs([t:0]):0])
x_
```

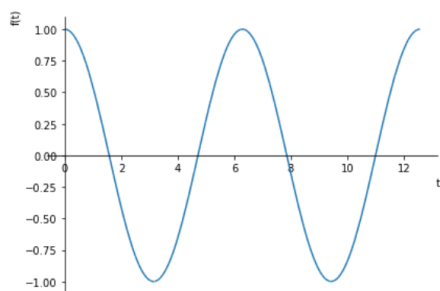
```
Out[69]:  $x(t) = \cos(\sqrt{k}t)$ 
```

グラフ表示の為、値を未だ定めていない定数 k に適当な数値を代入し、グラフを表示する。

```
In [70]: x_1 = x_.subs([k:1]).rhs
x_1
```

```
Out[70]:  $\cos(t)$ 
```

```
In [71]: smp.plot(x_1, (t, 0, 4 * smp.pi))
```



```
Out[71]: <sympy.plotting.plot.Plot at 0x26ba0c60848>
```

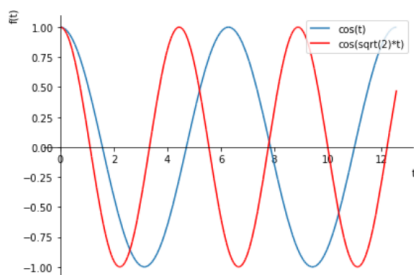
このように、復元力 k のばねを引っ張ったときに生じる単振動のグラフ描画が出来た。

ばねの復元力を変えた場合も加えた、複数の結果を描画したい場合は以下のように記述する。

```
In [72]: x_2 = x_.subs([k:2]).rhs
x_2
```

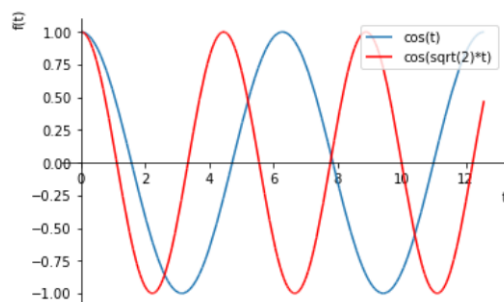
```
Out[72]:  $\cos(\sqrt{2}t)$ 
```

```
In [73]: result = smp.plot(x_1, x_2, (t, 0, 4 * smp.pi), legend=True, show = False)
result[1].line_color = "red"
result.show()
```



In[73]の1行目で、 k の値を2にした、2つの目の数式のグラフの色を変えるため、「`show = False`」でグラフの表示を一旦止めている。2行目で2つ目の数式のグラフの色を変えたのち、3行目でグラフ表示を実行している。描画したグラフを保存したい場合は、以下のように `save()`メソッドを使う。

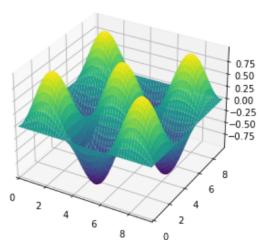
```
In [74]: result.save("result")
```



`plotting.plot3d()`メソッドを用いることで、3次元グラフの描写も可能である。

```
In [75]: t, x = smp.symbols("t, x")
```

```
In [76]: smp.plotting.plot3d(smp.sin(t) * smp.sin(x), (t, 0, 3 * smp.pi), (x, 0, 3 * smp.pi))
```



```
Out[76]: <sympy.plotting.plot.Plot at 0x26ba0d95648>
```

4. 結論

Python 言語と SymPy ライブラリを活用すれば、少数行のプログラムコードを記述するだけで、方程式や数列などの数式の作成、その数式への数値の代入、数式への微分・積分・極限計算、方程式や微分方程式の計算、数式の可視化が行えるようになることを示した。微分方程式の計算とその結果の可視化に関しては、空気抵抗を受けて落下する雨滴の運動方程式と、ばねに繋がれて単振動をする物体の運動方程式を具体例として取り上げた。数式を変えるだけで、他の様々な物理法則、化学方程式等の取り扱いも可能になる。本方法が、学生の皆さんが数式で記述される科学法則を理解する際の足掛かりとなれば幸いである。

5. 謝辞

本研究は 2018, 2019, 2020 年度の中部大学特別研究費(CP)、JSPS 科研費(S.A.S., 17K15102, 20K15762)の助成を受けて行った。

引用文献

- [1] Python (<https://www.python.org>)
- [2] SymPy (<https://www.sympy.org>)
- [3] Anaconda (<https://www.anaconda.org>)
- [4] Google Colaboratory
(<https://colab.research.google.com/notebooks/intro.ipynb>)
- [5] Shintani, S. A., Curriculum for Programming and Artificial Intelligence Education for Biomedical Science Students -From the Basics of the Python Language to Artificial Intelligence Algorithms for Generation-. *Annual report of Research Institute of Life and Health Sciences*. (2019).
- [6] Shintani, S. A., Washio, T. & Higuchi, H. Mechanism of contraction rhythm homeostasis for hyperthermal sarcomeric oscillations of neonatal cardiomyocytes. *Scientific reports*. 10, 20468 (2020).

Title: How to work with mathematical formulas that describe scientific law using the Python language and SymPy library for biomedical science students

Author(s): Seine A. Shintani

Address(es): Department of Biomedical Sciences,
College of Life and Health Sciences, Chubu University,
1200 Matsumoto-cho, Kasugai, Aichi 487-8501, Japan

Keywords: Python, SymPy, Mathematical calculations,
Scientific calculations