*数学的記号処理システムを用いたソフトウェアの構成手法

脇田建　高野陸　武田一起

ソフトウェアの構築、継続的な保守、文書管理は煩雑な作業を伴うソフトウェア工学上のさまざまな問題を孕んでいる。本研究は特に数学的な理論を応用するソフトウェアシステムをとりあげ、簡素な数学的構成とそれを実装したソースコードの複雑さに根差す概念的なギャップの問題、このことに付随する文書執筆と保守の困難を研究対象とする。本稿では、この問題について数学的記号処理システムを系統的に応用する方法を提案する。ソフトウェアの核心部となる数学的概念は記号システムによって記述し、それからソフトウェア、文書、文書に付随する図等を自動合成することによって、さまざまな困難を解消できることを示す。事例としては、各種グラフ可視化アルゴリズムとその対話システム、3 次元グラフィックス用の空間変換ライブラリなどを取り上げ、従来の実装と実装の複雑さ、性能、システムの可読性、保守性などを論じる。

1　はじめに

ソフトウェアのドキュメント化は頭の痛い問題である．[4] [2] ソフトウェアのドキュメントはソフトウェアの保守に不可欠な情報を提供しており，ソフトウェアの進化に伴なって，適切に更新されることが期待されている．このことは，管理者にもエンジニアにも等しく理解されている．ところが，大規模なソフトウェア開発において膨大な書類の山の全体構成を十分に把握しきれない例や，中小規模のソフトウェア開発においてドキュメント化に資する余裕がないことを理由にドキュメント化を諦めてしまう例も報告されている．[6]

ひとくちにソフトウェアのドキュメントといっても，開発行程の管理のためのもの，開発行程の各段階ごとのもの，そしてソフトウェアに付随する API ドキュメント等のように外部に公開されるものが存在する．本稿では，特に API ドキュメントについて論じる．

ソフトウェアの API ドキュメントには，API 関数の役割，引数や返り値とそれらの役割，副作用や例外発生の有無などのような**インタフェイス記述**とともに，API 関数の呼び出し方を示した例題や実行例の記述が含まれる．90 年代後半から，プログラムコードの記述とそれに付随する定型書式のプログラムコメントからインタフェイス記述を自動生成する技術が一般化し，インタフェイス記述の負担は軽減され，インタフェイス記述とプログラムコードの一貫性の問題も緩和された．[11] [8] [1] [13] [15]

一方で，インタフェイス記述に含まれる API 関数の振舞いが，実際のソフトウェアの動作と一致しない場合が多いことが Zhong らによって指摘されている [20]．ソフトウェア開発者のなかには，ソフトウェアドキュメントの完全性を諦め，ドキュメントにもライフサイクルがあり，過去の古い記述も受容すべきだと主張するものもいる ([6], Section 3.3: Evolving Documentation Needs)．その反面，近年のテスト駆動開発の流れを汲んで，ソフトウェアの動作記述に自然言語を用いるよりも，テストケースを流用する傾向がある．[8] [16] [12]

これらインタフェイス記述は API を利用するプロ

脇田建、東京工業大学情報理工学院数理・計算科学系、CREST/JST、高野陸、武田一起、東京工業大学理学部情報科学科

グラマが参照する情報であるが，これとは別に API の実装方式を文書化した**内部記述**の存在は API を継続的に保守する目的で重要である．今日のようにオープンソースソフトウェア開発が定着した時代にあって，API の利用者と API の開発者の垣根は低くなり，プログラムには API の内部記述が記述することが望まれている．

本稿が扱うのは，このような API ドキュメントの内部記述，それも特に記述が困難な内部記述の問題である．本研究のひとつの目的は，API ドキュメントの内部記述を困難にしている要因を明らかにすることである．ソフトウェアの問題領域における論理がプログラムを記述するプログラミング言語の細かい形式的論理よりも遥かに高い抽象度を有していることがある．このため，問題領域における記述とそれをプログラミング言語に翻訳したものの間に記述面で大きな差を生んでしまう点である．

このような論理的な抽象度の違いは，ドキュメント記述においても大きな問題を産む．なぜなら，API ドキュメントの内部記述の役割のひとつは，これらの抽象度のギャップを埋めることにあるからだ．

本稿では，ここで指摘した問題一般に挑むかわりに，その特殊な問題領域として**数学的知識を背景とする問題領域**を取り上げ，問題の所在を分析し，そこで見極めた諸問題を数式処理システムを活用することで解決を提案する．

本研究の貢献は，二つある．ひとつは数学的知識を背景とする問題領域についての API ドキュメントの内部記述を困難にしている 3 つの要因を明らかにした点である．(1) 数学的知識の記述に用いられている数式とプログラミング言語の抽象度の違い，(2) 数学的知識の表現であるところの数式を効率的にプログラムコードに変換できないこと，そして (3) 数式を用いたドキュメント記述が困難であること．もうひとつの貢献は，これらの困難について数式処理システムを用いることによりエレガントに解決できることを示した点である．

これ以後の本稿の構成は以下のとおりである．2 節では，上述の問題の具体的かつ典型的な例を掲げ，技術的な課題を示す．3 節では，数学的知識を背景とした領域の課題について，数式処理システムの数式変換，コード生成，そして数式記述生成の機能を上手に活用することで 2 節の例をきれいに解決できることを示し，さらにここで提案するアプローチがこの一例にとどまらず発展的な課題においても有効であることを示す．4 節では議論を深め，5 節で本稿をまとめるとともに今後の研究の方向性を示す．

2 問題の分析

本節では，小さな例をもとに本稿が対象としてる問題点を明らかにする．ここで扱う例題は「平方根を計算するプログラムとそのドキュメントを執筆すること」である．平方根の計算は，プログラミングの初学者が最初の数値計算の演習にでも扱うような簡単な内容ではある．しかし，数学的知識を背景とする問題領域のプログラミングの問題点のエッセンスを含んでおり，そのエレガントな解決は自明ではない．

正数 a の平方根は実変数 x の方程式 $x^2 - a = 0$ に対する非負根であり，この求解には普通 Newton-Raphson 法が用いられる．Python で記述した平方根を求めるプログラム例が図 1(a) であり，その内部ドキュメントが図 1(b) である．このドキュメントを作成するために，MathJax 拡張を施した Markdown 形式のドキュメントを記述したものが図 2 である．[†1]

さて，ここで図 1(a) と図 2 を見比べるといくつかの問題が見えてくる．

2.1 平方根の定義式はどこに？

すでに「正数 a の平方根は実数変数 x の方程式 $x^2 - a = 0$ に対する非負根」と述べたが，図 1(a) を眺めても $x^2 - a$ なる式は存在しない．本当にこのプログラムは平方根を計算するのだろうか．

一方，変数 x と定数 a を含む $(x + a/x)/2$ なる式がある．本当にこの定義は正しいのだろうか．

中等数学を学んだ辛抱強いエンジニアが図 1(a) を眺めれば，この式が確かに $x^2 - a = 0$ と関連した正

†1 この例は pLaTeX の記述にしか見えないかもしれないが，この研究では Jupyter Notebook と呼ばれる Python プログラムと Markdown を混在できる環境を用いている．[10]

```
def sqrt(a):
    x = 1
    for i in range(5):
        x = (x + a / x) / 2
    return x
```

(a) 平方根を計算するプログラム

正数 a の平方根を数値的に求める標準解法は方程式 $f(x) = x^2 - a = 0$ への Newton-Raphson 法の適用です．この手法では，解 x の近似解 x_n の近似度を漸次改善します．このために関数 f の $(x_n, f(x_n))$ 周辺における Taylor 展開の一次近似 $x_n - f(x_n)/f'(x_n)$ を用います．f の定義をこれにあてはめて

$$f(x) = x^2 - a$$
$$f'(x) = 2x$$
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{x + a/x}{2}$$

が得られます．

(b) 図 (a) の内部ドキュメント．図 2 のドキュメント記述から生成されたもの．

図 1: 数学的知識を背景とする領域における問題の例として，平方根の計算

しい式であることが理解できるかもしれない．しかし，このドキュメントや参考書，あるいは Wikipedia の類いの助けなしに，このプログラムを理解し，その正しさを把握することは困難だろう．

2.2 ドキュメント記述の複雑性

図 1(b) のドキュメントを書くのはかなりやっかいな作業である．図 1(b) の記述がこのドキュメントのもととなったドキュメント記述である．実のところ，このドキュメント記述の方が図 1(a) のプログラム記述よりも遥かに複雑で作業時間を要する．

本稿が問題にするような数学的知識を背景とする API 関数の内部ドキュメントの記述については，そもそもドキュメントの記述自体が困難なためにドキュメントには関連文献の参照情報を提示するのみに留めている例も多い．

```
正数$a$の平方根を数値的に求める標準解法は方程式
$f(x) = x^2 - a = 0$への Newton-Raphson 法の適用です．この手法では，解$x$の近似解$x_n$の近似度を漸次改善します．このために関数$f$の$(x_n, f(x_n))$周辺における Taylor 展開の一次近似$x_n - f(x_n) / f'(x_n)$を用います．$f$の定義をこれにあてはめて
\begin{align*}
f(x)    &= x^2 - a \\
f'(x)   &= 2x \\
x_{n+1} &= x_n - \frac {f(x_n)}{f'(x_n)}
         = x_n - \frac {x_n^2 - a} {2x_n}
         = \frac {x + a / x} 2
\end{align*}
が得られます．
```

図 2: 図 1(a) に示されたプログラムコードの内部ドキュメント記述．この記述から図 1b のドキュメントが生成される．

2.3 冗長性

プログラムコードの冗長性は，プログラムコード中に同一，あるいは類似した字句が繰り返し出現することを指す．冗長性はソフトウェアの品質，可読性，保守性を左右する重要な指標である．ここで検討している課題に関して，冗長性について着目すると深刻な問題があることに気づかされる．

まず，プログラムとドキュメントのあいだの冗長性として，プログラム中の `(x + a / x) / 2` という記述と，それに類似したドキュメント記述として `\frac {x + a / x} 2` が見つかる．プログラムコードの冗長性に比べると，ここで論じる冗長性は質が悪い．なぜなら，ここで指摘した例は，意味的には同じものを Python コードと LaTeX 風の数式記述として表現しており，通常の冗長性の範疇から逸脱したものだからだ．

プログラムとドキュメントのあいだに冗長性があることはやっかいな問題である．なぜなら，その冗長な記述の存在こそが，プログラムとドキュメントの一貫性を示唆しているからだ．つまり，プログラムの更新に伴なって適切にドキュメントを更新しなくてはならない．

ドキュメント記述のなかにもいくつかの冗長な記述を見ることができる（表 1）．この種の冗長性は，ド

表 1: ドキュメント記述中の冗長性

冗長な数式の記述	出現回数
`x^2 - a`	3
`x_n - f(x_n) / f'(x_n)`	2

キュメントの記述に要する労力を増加している．

3 数式処理を用いたコードとドキュメントの自動生成

本節は数学的知識を背景にした領域の記述について，数式処理システムを利用することが有効であること，特に，前節において挙げた平方根の問題がエレガントに記述できることを示す．

3.1 Jupyter/Python プログラマのための数式処理システム SymPy

SymPy は数式処理を実施するための Python ライブラリである．数式処理システムとしては，既存の Reduce, Maple, Mathematica に比べると不十分な点は多いが，Python を用いたソフトウェア開発の面からは面白い興味深い点がある．

既存の数式処理システムとは異なり SymPy はライブラリであり，プログラミング言語ではない．すなわち，SymPy を利用することによって，数式を表現するためのデータ構造と数式上の演算を表現する各種関数や演算子が利用できるようになる．SymPy を用いることで数学記号と重要な定数，関数，等式や不等式，行列やテンソル，方程式系などが表現できる．これらは Python のデータ構造として提供されるため，標準的な Python のデータ構造と同様に Python の変数に束縛したり，Python の関数に実引数として渡すことができる．

数式に対しては一般的な代数演算や微積分を施すことができる．また，方程式系のソルバーは与えられた方程式系の解析的な解を与える．たとえば，以下の例では二つの数学の変数 a と x を用意し，それぞれ Python の変数 `a` と `x` に束縛した上で，簡単な数式 $(x^2 - a)$ を作成したものを（数式）微分した結果として $2x$ を得ている．

```
def md(*args):
    s = ''
    for x in args:
        if (isinstance(x, sp.Basic)
            or isinstance(x, sp.MutableDenseMatrix)
            or isinstance(x, tuple)):
            s += sp.latex(x)
        elif isinstance(x, np.ndarray):
            s += sp.latex(sp.Matrix(x))
        elif (isinstance(x, str)): s += x
        elif (isinstance(x, int)
            or isinstance(x, float)): s += str(x)
        else: print(type(x))
    markdown(s, raw=True)
```

図 3: SymPy 形式の数式が混ざったテキスト列から Markdown 文書を生成するユーティリティの定義

```
$ python -q
>>> import sympy
>>> a, x = sympy.var('a x')
>>> e = x**2 - a
>>> e.diff(x)
2*x
>>> f = sympy.lambdify([x], e.diff(x))
>>> f(5)
10
```

Python プログラマにとって有り難い存在として，SymPy で表現した数式に相当する Python のラムダ式を生成する機能 (`lambdify`) がある．このように生成されたラムダ式を用いれば，SymPy の数式に該当する数値計算ができる．上の例では `lambdify` を用いて `e.diff(x)` を変換して Python の関数 `f` を得，`f(x)` が確かに 10 を返すことを確認している．

さらにドキュメントを書くときに有用なのは SymPy で表現した数式を LaTeX 形式の文字列に変換する機能である．

```
>>> sympy.latex(e)
'- a + x^{2}'
```

この機能を利用することで，ドキュメント記述の文字列と SymPy 形式の数式が混在するデータ列から，MathJax 拡張を施した Markdown 書式のドキュメントを生成するためのユーティリティ関数 `md` を定義し

```
def NewtonRaphson(関数式, x, *args):
    F = sp.Function('F')
    NR = x - F(x) / sp.diff(F(x), x)

    差分計算式 = NR.subs(F(x), 関数式).doit()
    差分計算   = sp.lambdify((x, *args),
                             差分計算式)

    def newton_raphson(*args, x = 1, ループ回数 = 5):
        for i in range(ループ回数):
            x = 差分計算 (x, *args)
        return x

    return newton_raphson
```

図 4: Newton-Raphson 法の一般的な記述

```
md('# `NewtonRaphson`関数を用いた平方根の計算')

平方根 = NewtonRaphson(x**2 - a, x, a)
md(r'$\sqrt 2 = ', str(平方根(2)), '$')
```

NewtonRaphson関数を用いた平方根の計算

NewtonRaphson関数は、方程式 $F(x) = -a + x^2 = 0$ の根をNewton-Raphson法を用いて求める関数を返します。 この手法は、方程式の根の近似値 x_n を漸次改善します。 このために、$(x_n, f(x_n))$ におけるTaylor展開の一次近似を用います：

$$x - \frac{F(x)}{\frac{d}{dx}F(x)}$$

関数式の定義 $-a + x^2$ をこれにあてはめて，以下の差分計算が得られます：

$$x_{n+1} = x_n - \frac{-a + x_n^2}{2x_n}$$

$$\sqrt{2} = 1.4142135623730951$$

図 5: 図 4 を用いて，平方根を計算する関数を生成し，$\sqrt{2}$ を計算させた様子．平方根が正しく計算されているだけでなく，平方根を計算する内部ドキュメントは 6 の記述を平方根の計算に特殊化させたものが生成されている．

た．(図 3)

前節においては Newton-Raphson 法を用いて平方根を計算するプログラムを紹介した．これに対して図 4 は，平方根に限定せず Newton-Raphson 法を一般的に記述したプログラムである．この関数は，引数として根を求めたい方程式（関数式）とその方程式

```
xn, xn1 = sp.var('x_n x_{n+1}')
md('`NewtonRaphson`関数は、方程式 $', F(x), '=',
    関数式, '`= 0$ の根を Newton-Raphson 法を用いて求める関数を返します。この手法は、方程式の根の近似値 $x_n$ を漸次改善します。このために、$', (xn, F(xn)), '$ における Taylor 展開の一次近似を用います：\n\n
    $$', NEWTON_RAPHSON, '$$\n\n',
    '関数式の内容 $(', 関数式, ')$ をこれにあてはめて，以下の差分計算が得られます：\n\n$$',
    xn1, '=', 差分計算式.subs(x, xn), '$$')
```

図 6: 図 4 の内部ドキュメント記述

における変数 (x) を受け取り，与えられた方程式を x で受け取った（数学の）変数について求解する関数を返す．

たとえば，関数式として x**2 - a を与えると，方程式 $x^2 - a = 0$ を数値的に求解する Python 関数，すなわち平方根を計算する関数を返す．図 5 では，こうして得られた Python 関数を「平方根」という名前の Python 変数に束縛した上で，引数に 2 を渡すことで，$\sqrt{2}$ の近似解として 1.41421356... を得ている．

では，図 4 の記述を見ていこう．引数に与えられた関数式と変数 x はそれぞれ SymPy 表現の数式と数学記号である．3 行目で定義された Python 変数には，Newton-Raphson 法の差分式 $(x - F(x)/F'(x))$ が SymPy の数式として与えられ，Python 変数の NR[†2] で束縛している．ここまでは関数の形を特定せずに表現してきたが，実際には NewtonRaphson 関数の引数「関数式」として，たとえば $x^2 - a$ のような数式が与えられる．そのことを反映するために，一般的な差分式の $F(x)$ を具体的な関数式の内容で置き換えている操作を経て，**差分計算式**を定義している．差分計算式を方程式の変数 x によって抽象化したものが差分を数値計算する**差分計算**関数である．

newton_raphson 関数のなかでは，このように得た**差分計算**関数を繰り返し呼び出すことで Newton-Raphson 法の計算を実施している．最終的に NewtonRaphson 関数は newton_raphson 関数を返

†2 NR は Newton-Raphson を短縮した名前．

り値として与える．ここで返される `newton_raphson` 関数は，一般的な NewtonRaphson 法の計算を引数に与えた関数式に関して特殊化したものと見做すことができる．

ここでの実装は，Newton-Raphson 法の（特定の数式の例の依らない）一般的な記述（図 4）と平方根を求めるために Newton-Raphson 法を方程式 $x^2 - a = 0$ 適用する記述（図 5）にきれいに分離されている．前者は，Newton-Raphson 法の方式を素直な数式処理として記述したものである．後者は `NewtonRaphson` 関数に `x**2 - a` というまさに平方根を定義する式を受け渡すのみである．以上より 2.1 節で指摘した平方根の定義式が消え失せる問題が解消されるだけでなく，元々の記述のなかで Newton-Raphson 法の一般的な性質と平方根という特定の応用問題の記述が混在していた問題も解消できた．

さて，図 6 は，図 4 に対応したドキュメントを生成するコードである．このコードを図 4 の **return** 文の直前に挿入することによって，`NewtonRaphson` を**関数式**の内容に特化したドキュメントを生成できる．

この記述において，注目すべき点は図 2 と比較して数式の記述量が大幅に少ない点である．これは，`NewtonRaphson` 関数の定義に出現する SymPy 表現の 4 つの数式（`F(x)`，関数式，`NEWTON_RAPHSON`，差分計算式）を埋め込むことによって，それらを具体的に記述する手間が除去されているからである．この結果，2.2 で指摘されたドキュメント記述の複雑性の問題を避けることができた．

このことは単なる記述の簡素化と理解してはいけない．もうひとつの重要な点は，プログラム記述とドキュメント記述の間の冗長性が除去されていることである．以前のドキュメント記述（図 2）にはさまざまな数式が具体的な形で出現していた．一方，今度のドキュメント記述では，具体的な数式の構成は SymPy 数式を扱う `NewtonRaphshon` 関数の記述や，図 5 に現れる SymPy 表現の数式を活用している．

このことはプログラムコード記述に出現する数式を表現したデータ構造を活用し，ドキュメントに出現する数式の生成に用いることによって，データ構造として表現された数式の再利用性を高めて，プログラムコード記述とドキュメント記述の双方で利用したと見做せる．この結果 2.3 節で指摘した数式記述の冗長性の問題を解決できた．

4 議論

4.1 平方根だけなの

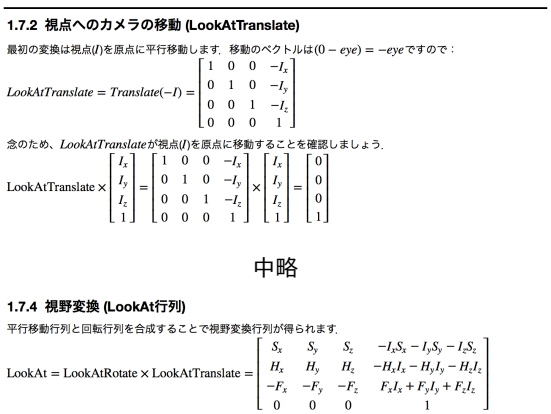
本稿では議論を簡素化するために，ごく単純な例題として平方根を計算する例を用いた．この例の直接の発展として，`NewtonRaphson` に $x^k - a$ を与える k-乗根の計算や，条件次第ではより収束の速いと言われる式についての適用を行った．

この研究のきっかけは C++で記述されていた SNS の可視的解析システムの Python への移植にある．Python は統計処理，グラフ処理，機械学習の豊富なライブラリを備えており，高速グラフィックスのための OpenGL のライブラリも提供していたのだが，3 次元グラフィックスに必要な空間変換のライブラリで適切なものがなかったため自作することとした．

この作業において，Riccio の OpenGL Mathematics (GLM, C++, [5])，Gohlke の transformations.py (Python, [7])，Brett の transforms3d (Python, [3]) を参考にした．ここで transforms3d は Golke の協力のもの transformations.py を発展させたものである．興味深いことに Brett が書いた README には以下の記述がある．

> We have tried to document the algorithms carefully and write clear code in the hope that this code can be a teaching reference. We document the math behind some of the algorithms using ‘sympy’ in “transforms3d/derivations”.

たしかに Brett のコードは読み易く，内部で用いられているアルゴリズムについては引用元となる書籍やウェブサイトへの参照が提供されている点で普通の API に比べて優れている．しかし，ここから生成される API ドキュメントをひとつの書物として見る場合は，その完全性が乏しいと言わざるを得ない．問題は Brett の手になる transforms3d の API ドキュメントの欠陥ではなく，おそらく世の中にあるほとんどの 3D グラフィックスライブラリの状況がこのレベル



1.7.2 視点へのカメラの移動 (LookAtTranslate)

最初の変換は視点(I)を原点に平行移動します．移動のベクトルは$(0 - eye) = -eye$ですので：

$$LookAtTranslate = Translate(-I) = \begin{bmatrix} 1 & 0 & 0 & -I_x \\ 0 & 1 & 0 & -I_y \\ 0 & 0 & 1 & -I_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

念のため、$LookAtTranslate$が視点(I)を原点に移動することを確認しましょう．

$$\text{LookAtTranslate} \times \begin{bmatrix} I_x \\ I_y \\ I_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -I_x \\ 0 & 1 & 0 & -I_y \\ 0 & 0 & 1 & -I_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} I_x \\ I_y \\ I_z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

中略

1.7.4 視野変換 (LookAt行列)

平行移動行列と回転行列を合成することで視野変換行列が得られます．

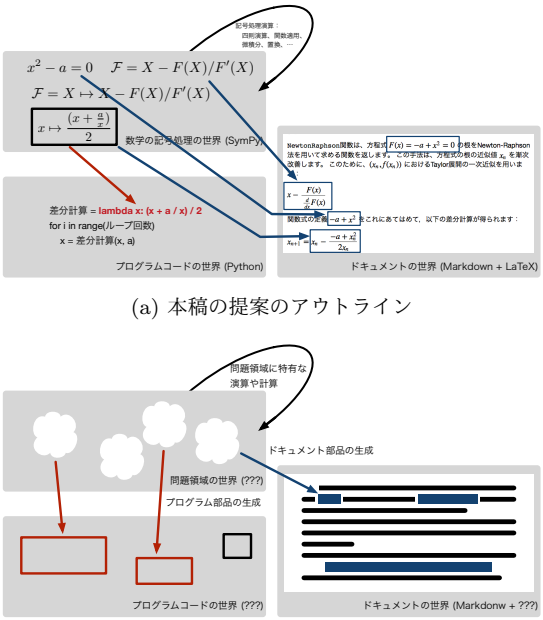
$$\text{LookAt} = \text{LookAtRotate} \times \text{LookAtTranslate} = \begin{bmatrix} S_x & S_y & S_z & -I_xS_x - I_yS_y - I_zS_z \\ H_x & H_y & H_z & -H_xI_x - H_yI_y - H_zI_z \\ -F_x & -F_y & -F_z & F_xI_x + F_yI_y + F_zI_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

図 7: 本稿の技術を応用して生成された 3 次元グラフィックスライブラリの API ドキュメントより，LookAt 変換のドキュメント．

にすら達していないことである．

われわれは今回の提案を 3 次元グラフィックスのための空間変換ライブラリ API（四元数，拡大縮小変換，回転変換，平行移動変換，モデルビュー変換，LookAt 変換，正射影，透視投影変換など）の構築に応用した．詳細は省くが 7 図は，そのうち LookAt 変換行列を生成する機能について生成したドキュメントの一部である．このプロジェクトにおいては，線形代数における方程式系の解と線形代数演算を利用することでプログラムコードとドキュメントコードの簡素化に成功している．

本稿で提案した技術は，いくつかの対話的なグラフ構造の可視化システムの構築に応用されている．武田は無向グラフを力学的モデルによって可視化する基本的なアルゴリズムの記述に SymPy を用いた [18]．このアルゴリズムでは，グラフの頂点間に仮想的にバネを張り巡らし，それらのバネ群が構成するポテンシャルエネルギーを最小化することでグラフを定める．[9] この研究では，ポテンシャル関数を SymPy で記述した上で，その一次微分と二次微分を数式として得たものを lambdify によって Python の関数を得ている．二次微分で得られるヘッセ行列は巨大であり，人手でこれを正確に計算し，そのとおりのプログラムを作成することには困難を伴なう．



(a) 本稿の提案のアウトライン

(b) 数学的知識を問題領域とした本稿の提案を抽象化し，他の問題領域への適用を検討すると…

図 8: より広い問題領域への応用をめざして

高野は対話的な高次元グラフ可視化システムである AGI3D のアルゴリズム [19] の記述を試みた [17]．AGI3D のアルゴリズムでは，マウスドラッグ操作を高次元回転操作に翻訳する過程で制約充足問題を解いている．この研究では，比較的簡単な制約式から思いがけず複雑な制約設定系を半自動生成することで，200 行ほどの難解なプログラムコードを数十行の数式処理に変換している．これらの数式処理の記述は，このアルゴリズムの背景となっている統計理論と対応しており，理解も容易になった．

これらの記述例は限定的ではあるものの，微積分，線形代数，物理シミュレーション，大規模行列などという比較的広い数学領域を覆っている．以上の結果より，本稿で提案しているアプローチが数学や物理の広い分野の問題の記述に利用できるという感触が得られた．

4.2 ほかの領域への応用

本稿では数学的知識を背景とするソフトウェアにおけるプログラム作成とドキュメント執筆について問題を明らかにするとともに数式処理システムを利用したエレガントな解決策を提示した．この仕組みを模式的に表現したものが図 8(a) である．

ここで扱っている問題は 3 種の世界，すなわち問題領域を自然に記述可能な**数学の世界**，計算として実行するための**プログラムの世界**，さらにプログラムを理解するための用意する**ドキュメントの世界**があること，それらの間に実は密接な関係があるのだが，これまでほとんどの場合，それを人力に頼って記述していたことである．数学の世界の記述は主に数式が用いられ，プログラミングの世界ではプログラミング言語が利用され，ドキュメントの世界ではプログラムコメント，ワープロ，表計算ソフト，マークアップ言語などが利用されてきた．ソフトウェア技術者たちはこれらのツールを場当たり的に活用してドキュメントを用意してきた．

これに対して，本稿の提案では数式処理システム SymPy を用いることで，数学における記号処理，プログラムコード断片の生成，ドキュメントに埋め込むべき数式表現の生成を効率的に実施できた．

ところで，世の中の諸問題のなかで数学的知識を背景とするものはわずかである．そのほかの問題についてはどのようにアプローチすべきだろう．

それぞれの領域ごとに問題となっている概念を記述する記述系というものがある．たとえば，音楽でいえば楽譜であったり，音楽演奏でいえば Midi データである．法律でいえば，法律の条文や判例がそれにあたるだろう．

今回の数式処理システムは幸運な例であり，数式という領域を形式的に記述するための枠組みが SymPy ライブラリの実装として備わっていた．また，SymPy が提供するコード生成系と LaTeX 風の数式生成系を利用することにより，数学の世界からプログラミングの世界，そしてドキュメントの世界への橋渡しの用意があった．実際，本稿のプロジェクトを実施する上で著者らが追加した仕掛けは，図 3 に掲げたもの程度にすぎない．

```
# 視野錐台が単位立方体に射影されること
Points = Perspective * PointsCamera
for c in range(4):
    assert(Cartesian(Points[:,c])
           == Cartesian(PointsPerspective[:,c]))
```

図 9: 数式的テストケースの例

一方，このほかの領域において，これだけの仕掛けが完備している例は少ないかもしれない．そもそも，数学ほど形式的な理解が進んでいない領域も多いだろうし，Midi データのようにデジタル化はされているものの，それを問題領域における音楽の理解として把握することにはまだもう少し準備が足りないように思えるものもある．ほとんどの領域では不定形，あるいは場当たり的で煩雑な知識の記述が用いられているのではないだろうか．

本稿での成果を他の領域へ展開するにあたって，最初の障害がその領域の知識を形式的に記述することである．幸運なことにわずかな努力でそれが達成できる領域が存在した場合は，形式化されたその領域の知識から，望ましい計算を取り出すための `lambdify` のような関数を提供するとともに，一般的な文書に組込むための変換器を用意すればよいように思う．図 8(b) は，この議論を模式的に表したものである．言うは易き行うは難しではあるが，この図を念頭にすこしでも応用範囲を広げられないか考察を深めたい．

4.3 数式的テストケース

SymPy は数式を簡約するための `simplify` 関数を提供している．これを利用することで二つの数式の等価性を判定できる場合がある．たとえば，ふたつの実数値をとる数式 `e1` と `e2` に対して，`e1 == e2` によって，等価性について判断できる場合がある．†3

図 9 はこの性質を利用し，視野錐台が透視投影変換によって単位立方体に射影されることを確認している．ふたつの数式の恒等性は，数式に出現する記号への任意の値の割り当てについての恒等性を意味する．

†3 もちろん任意の数式の組について等価性を判定できるわけではない．

このため，変数への割り当てについての有限のサンプルについて等価性を確認する通常の単体テストよりも強力な道具立てとなる．

4.4 なぜ SymPy，なぜ Python

数式処理システムとしての SymPy は一定の有用性はあるものの，まだまだ基本的な機能で欠けているものもあり，バグも多い．数式処理システムとしての能力としては Maple や Mathematica の方がはるかに高機能で安定しているだろう．

本研究で SymPy を用いた理由は，たまたま Python を用いた開発のなかで問題を発見をしたことにすぎない．しかし，Python のための Python ライブラリとして提供されている数式システムという SymPy は面白い位置付けにある．まず，SymPy は言語ではなく拡張ライブラリであるため，Python のアプリケーションと統合して利用することが用意である．また，数式を実装する計算を Python が提供する無名関数 (`lambda`) として提供しているため，アプリケーションの実行時に数式処理を施して，その場で処理された数式に関する数値計算ができることも魅力的である．もしも，SymPy がライブラリでなく（Maple や Mathematica のような）独自の数式処理言語だったら，あるいは，数式から計算への変換に無名関数を用いていなかったら，オフラインでコード生成とリンク作業が必要になっていたことであろう．

ところで，数式から実行可能なコードを出力する機能は SymPy の専売特許ではない．Mathematica は C 言語のコードを生成することができ，Maple は C, Fortran, JavaScript, Python を始めとする多くのプログラミング言語のコードを生成できる．一方，SymPy は本稿で紹介した Python lambda の生成だけでなく，C, Fortran 95, JavaScript といった汎用プログラミング言語と OpenCL や CUDA のような汎用 GPU 計算もサポートしている．

いくつかの数式処理システムとそれらが提供するコード生成器を利用すれば，本稿と同様の試みに近いことはできるかもしれない．ただ，本研究の成果が比較的容易に得られた背景として，Jupyter Notebook の存在は無視できない．[†4]HTML, Markdown, MathJax を併用できるノートブックの存在がなかったならば，本稿のアイデアもさまざまな変換器を組み合わせた，実用性が疑わしいものとなっていただろう．そもそも，著者らも，このような研究を始める気にすらならなかったかもしれない．Jupyter Notebook の存在は本研究の成果において重要であった．

本稿のアイデアに基づいて，類似した試みをするために，いくつかの数式処理システムとさまざまなプログラミング言語を利用することが考えられるが，数式処理，プログラムの作成，ドキュメントの執筆という 3 つの作業を滑らかに実施するための，もっとも素直な環境は Jupyter Notebook for Python 基盤の上での SymPy の利用であろうと思われる．

5 まとめと今後の課題

本稿では数学的知識を背景とする領域のソフトウェア記述とそのドキュメントの内部記述について扱い，その困難が問題領域であるところの数学の世界とプログラミングをする計算の世界の間の大きなギャップにあること，そして，このギャップを埋めるべき内部記述を助けるツールが存在しないことを指摘した．

この問題について，数学の世界を数式処理システムによって記述し，数式から自動生成したプログラム断片や LaTeX 書式の文書断片をプログラムや内部記述に埋め込むことによって，問題をきれいに解消できることを示した．

今後は，この技術の適用範囲をさらに拡大するとともに，ソフトウェアドキュメントの記述上の問題をさらに深く検討したい．

謝辞 この論文の執筆にあたって，時間をさいて議論に応じて下さった池谷のぞみ先生と権藤克彦先生に感謝します．本稿のアイデアについて萌芽的な段階でご議論をいただいた SIGPX 参加者と、特に参考文献をご教示下さった，加藤淳先生，増原英彦先生，渡部

†4 Jupyter Notebook の前身である iPython 環境 [14] は，Mathematica Notebook に触発されて開発された Python 用のノートブック環境であった．iPython のノートブック機能を多言語化して多くのインタプリタ言語に対応させたものが Jupyter Notebook である．[10]

卓雄先生に感謝します.

参 考 文 献

[1] Aguiar, A. and David, G.: WikiWiki Weaving Heterogeneous Software Artifacts, *Proceedings of the 2005 International Symposium on Wikis*, New York, NY, USA, ACM, 2005, pp. 67–74.

[2] Boswell, D. and Foucher, T.: *The art of readable code: Simple and practical techniques for writing better code*, Theory In Practice, O'Reilly & Associates Inc, November 2011.

[3] Brett, M.: Transforms3d.

[4] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J.: *Documenting software architectures: Views and beyond, portable documents*, Addison-Wesley Professional, 2nd edition, October 2010.

[5] Creation, G.-T.: OpenGL Mathematics.

[6] Forward, A. and Lethbridge, T. C.: The Relevance of Software Documentation, Tools and Technologies: A Survey, *Proceedings of the 2002 ACM Symposium on Document Engineering*, New York, NY, USA, ACM, 2002, pp. 26–33.

[7] Gohlke, C.: transformations.py.

[8] Hoffman, D. and Strooper, P.: API documentation with executable examples, *Journal of Systems and Software*, Vol. 66, No. 2(2003), pp. 143 – 156.

[9] Kamada, T. and Kawai, S.: A general framework for visualizing abstract objects and relations, *ACM Transactions on Graphics*, Vol. 10, No. 1(1991), pp. 1–39.

[10] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C., and Team, J. D.: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, IOS press, 2016, chapter Jupyter Notebooks – a publishing format for reproducible computational workflows, pp. 89–90.

[11] Kramer, D.: API Documentation from Source Code Comments: A Case Study of Javadoc, *Proceedings of the 17th Annual International Conference on Computer Documentation*, SIGDOC '99, New York, NY, USA, ACM, 1999, pp. 147–153.

[12] Kr amer, J.-P., Brandt, J., and Borchers, J.: Using Runtime Traces to Improve Documentation and Unit Test Authoring for Dynamic Languages, 2016. to be presented at CHI 2016.

[13] Nørmark, K.: The vicinity of program documentation tools, Technical Report 09-004, Department of Computer Science, Aalborg University, December 2009.

[14] Pérez, F. and Granger, B. E.: IPython: a System for Interactive Scientific Computing, *Computing in Science and Engineering*, Vol. 9, No. 3(2007), pp. 21–29.

[15] rust lang.org: *The Rust Programming Language*, available online, 2016, chapter 5.4 Documentation.

[16] Stylos, J., Myers, B. A., and Yang, Z.: Jadeite: Improving API Documentation Using Usage Information, *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '09, New York, NY, USA, ACM, 2009, pp. 4429–4434.

[17] 高野陸, 脇田建: 数式処理システムを用いた対話的高次元グラフ可視化方式の実装, September 2016. 日本ソフトウェア科学会第 33 回大会ポスター発表.

[18] 武田一起, 脇田建: 数式処理システムを用いた力学モデルに基づく対話的グラフ可視化方式の実装, September 2016. 日本ソフトウェア科学会第 33 回大会ポスター発表.

[19] Wakita, K., Takami, M., and Hosobe, H.: Interactive high-dimensional visualization of social graphs, *Visualization Symposium (PacificVis), 2015 IEEE Pacific*, IEEE, 2015, pp. 303–310.

[20] Zhong, H. and Su, Z.: Detecting API Documentation Errors, *SIGPLAN Not.*, Vol. 48, No. 10(2013), pp. 803–816.