Python で実験研究する力学系

水谷 正大

2024 年

まえがき

時代を画した先端的研究が今では手近のパソコンで 1 分以内で計算し、再現できるようになりました。計算結果を matplotlib を使って視覚化することで語らせることができます。

本書は、非線形科学の扉を開いたいくつか有名な計算機実験をこの分野に関する予備知識を一切仮定せず再現するプログラムをどのように書くかを紹介し、読み物としても通読できるようにしたテキストです。本書の読者として、Python のやや進んだプログラミングを学ぼうとしている方々、数値計算や記号計算に関心のある高校生・大学生や大学院生または技術者を想定しています。Python 環境の利用・整備の仕方については「付録」にまとめました。パソコンだけで直ぐに実行できる Google Colaboratory の利用が手っ取り早い方法です。

事前知識を前提とはせずに必要なことを本書を通じて知ることができるように工夫しました（入門と称した難しいテキストは多数あることは知っています。本書で理解できない部分は筆者の理解不足のためです)。微分方程式についての知識は必要ではありませんが、線形代数の初歩やや若干の微積分の知識、またはプログラミングの知識のいずれかがあれば読み易いかもしれません。

紹介した Python プログラムはどれも高々 50 行程度の短いものです。短いプログラムでも科学的な実験や探求が可能であることを重視し、紹介した Python プログラムではクラス定義などは行っていません。一方、プログラムの可読性を高め正しく運用できるように関数定義などで型アノテーションを付けて静的型チェッカー `mypy` を実行し静的チェックをパスしたコードを掲載するように心がけました。

本書の構成

本書で紹介した話題は網羅的ではありません。プログラミングにおいても数理としても多くの紹介すべき事柄が筆者の浅学菲才のためにまとめきれずに漏れ落ちてしまいました。

第 1 章から第 2 章は、第 3 章以降で活用するプログラミング技巧の準備に当てらています。ライブラリ NumPy を使う数値計算が本書の Python プログラミングの中核になっていますが、同時に、本書で利用する範囲で SymPy を使う記号計算、及び両者の相互変換を具体的に紹介しています。

数学的な知識をプログラミング上の操作に関連付ける、逆にプログラミング上の技巧を数学的に整理するためのリファレンスとして利用できるように細かく目次をつけました。

線形代数における行列や固有値・固有ベクトルの計算、関数のテーラー展開やヤコビ行列など、本書だけでなく日常的な利用にも重宝する計算を取り上げました。

本編は第 3 章から始まります。微分方程式系の数値計算に先立って、第 3 章では与えた写像 F の反復 $x_{n+1} = F(x_n)$ によって得られる離散力学系の軌道列 $\{\boldsymbol{x}_n\}$ のプロットを手がかりとして複雑さがもたらす構造を紹介しています。Small[44] は写像研究の重要性について、微分方程式の流れの研究がそれを横切る横断面上の写像の研究に帰着されるだけでなく、微分方程式が呈する現象や課題がもっと簡単な写像の研究に表れていることを強調しています。実際、写像としては簡単であるとしてもその振る舞いには限りない微細構造を伴うことを紹介します。

第 4 章では微分方程式とは何であるかを導関数が満たす方程式としてではなく、ベクトル場が定める接線方程式であるという幾何学的立場で説明します。微分方程式の軌道を数値的に求めるためのいくつかの数値解アルゴリズムを紹介し、その精度を数値的に検証します。

微分方程式の数値解を計算してその解軌道を描画するだけでなく、その挙動の特徴をどう捉えるかが問題となります。第 5 章では非線形微分方程式として主にローレンツ方程式を取り上げその特徴を紹介した上で、その典型的軌道がなぜ非周期的なのかをポアンカレの切断面の方法およびロレンツプロットの方法によって示します。第 6 章では、古典力学の運動方程式を取り上げ自由度 2 のエノン=ハイレス系や Fermi-Pasta-Ulam(FPU) の実験、そして戸田格子を取り上げました。これらはカオス研究の嚆矢となりましたが、非線形性は必ずしも乱雑軌道をもたらすわけでなく、戸田格子など可積分系の発見を導きました。第 7 章で偏微分方程式である KdV 方程式がソリトン現象を示すことを Zabusky-Kruskal のアルゴリズムを使って再現しています。

取り上げた数値実験研究

本書で再現対象として取り上げた有名な数値実験研究は次のようです。

(1)E. Fermi, J. Pasta and S. Ulam, *Studies of nonlinear problems*[31, 1955].

(2)E. Lorenz, *Deterministic nonperiodic flow*[39, 1963].

(3)M. Hénon and C. Heiles, *The applicability of the third integral of motion: Some numerical experiments*[34, 1964].

(4)N. Zabusky and M.D. Kruskal, *Interaction of 'solitons' in a collisionless plasma and the recurrence of initial states*[48, 1965].

(5)M. Hénon, *Numerical study of quadratic area-preserving mappings*[35, 1969].

(6)FordJ, Stoddard and Turner,*On the Integrability of the Toda Lattice*[33, 1973].

(7)R. May, *Simple mathematical models with very complicated dynamics*[41, 1976].

(8)上田睨亮, 非線形性に基づく確率統計現象–Duffing 方程式で表わされる系の場合 [47, 1978].

(9)B.Mandelbrot, *Fractal Aspects of the iteration of* $z \to \Lambda z(1-z)$ *for complex* Λ *and* z[40, 1980].

(1) は非線形格子振動の FPU の実験として有名な開発されたばかりのコンピュータを使った非線形格子振動のエネルギー分配に関する研究です。FPU の実験は (3), (4), (6) に深い関わりがあることが後で判明し、戸田格子やソリトンなど可積分系の理論として大きな水脈の源流となりました。2) は 3 変数の簡単な散逸系、(3) は自由度 2 の古典力学の運動方程式の研究で、いずれも非線形微分方程式の数値解を巡う議論を行って、微分方程式のカオス的挙動が初めて視覚化されました。(4) は非線形偏微分方程式である KdV 方程式において初期境界状態がいくつかの孤立波分裂しその波の衝突挙動が粒子的に振る舞うソリトン現象を発見した数値研究です。(6) は戸田格子の積分可能性を数値的に示したたもので、翌年に Hénon[36] と Flaschka[32] による異なるアプローチによって数学的に可積分性が示されました。(8) は世界で最も早くから精密なカオス現象に取り組んでいた上田による強制ダッフィング方程式の研究である。

(5),(7),(9) は写像力学系の研究で、(5) は平面上の点を平面に写す面積保存写像, (7) は $[0,1)$ 区間上の点を区間に写す 1 次元写像、そして (8) は複素平面上の写像です。

なぜ Python プログラミングか

本書では汎用のプログラム言語 Python と NumPy を中核とする外部拡張ライブラリを使って数値計算と結果のプロットを行いました。もちろん、他のプログラム言語を使って本書で取り上げたことは実現できます。

Python はコードの可読性が高いために学びやすく活発な開発コミュニティから効率の良い多彩なライブラリの提供があり、機械学習などでも多く使われています（それぞれのプログラム言語の支持者は指示すべき明確な理由を持っています)。ここではプログラム言語の選択以外の課題に触れてみたいと思います。

本書で紹介したような数値実験（あるいはさらに進んだ研究）はいわゆる汎用プログラム言語以外に数式処理システム（計算機代数システム）でも達成可能です。数式処理システムは人が行う特定目的の計算を実現するために開発されたシステムで、今なお多くのプロジェクトがが進行中です。数式処理システムでは目的とする計算機構が組み込まれており、必要ならば自分で関数をプログラム定義することによって目標とする処理を達成することができます。実際、汎用的な数式処理システムとして商品の Mathematica、Maple や

オープンソースの SageMath を使って本書で取り上げた計算を行うことは可能です。

数式処理システムは主たる目的を記号処理においているため、数値計算としての計算速度は犠牲になります。本書でも SymPy を使ってハミルトニアンから正準方程式を導出して、その運動方程式を数値計算する方法を説明していますが、多くの計算資源（時間）が必要になります。また、数式処理システムでは背景で働いている処理アルゴリズムをユーザに見せないために（$\sin x$ を微分すると $\cos x$ となる過程を見せてもほとんど意味がありません)、数式処理システムでは数学的に正しい結果をもたらすブラックボックスであるかのように動作します。正しい計算方法を与えれば誤りのない結果を与えることが数式処理システムの目的である以上、(数値誤差など共通した課題があるにせよ）それ自体には問題がありません。

しかしながら、これからのコンピュータ利用では、与えられた式を評価し数値計算するだけでは不十分であるかもしれません。本書で紹介する挙動は、自然界や実験室の中だけでなく、社会現象においても同様な記述が可能な現象（たとえば同期現象）と考えることができます。そこおいては、過去に蓄積されてきた先行研究や試行錯誤に頼っていた計算対象をどのように設定し何を計算すべきか、また類似現象の発見や関連付けなど、より高度な数理および知能的処理が必要になると思われます。

このような観点に立った研究がどのような姿になるか筆者には想像もできません。Python プログラミングを活用して非線形研究の黎明期の研究に再度親しむことで、読者が現在の視点から新しい研究・開発の扉を拓くことを期待しています。

本書でのプログラミングは Python3 に準拠しています。本書のプログラムは macOS 上で Python 3.9.1

```
IPython 7.21.0
NumPy 1.20.1
Matplotlib2 3.3.4
SciPy 1.6.1
SymPy 1.7.1
```

で動作確認を行いました。Google Colaboratory でも問題なく動作します。プログラミングに際しては上記外部ライブラリ以外に特に高度な機構は使っていませんし、長大なプログラムもありません。ぜひ、自分で動かしてみて理解を深めてください。

本書を通じて Python による数値計算や記号計算、そして微分方程式の世界に興味を持っていただければ幸いです。

水谷正大

目次

第 1 章　NumPy 入門

NumPy スタイルでプログラムコードを書くことでプログラムの可読性を高め、計算を高速化することができます。NumPy は Python における洗練され高機能な数値計算手段を提供する重要な外部ライブラリで、Fortran や C 言語のために長年開発されてきた科学計算の方法を Python で利用できるようにしたパッケージが集められており、NumPy は Python における科学計算の中核を担っています。

この章では NumPy ファミリーにおけるデータと取り扱いや、簡単な使い方を紹介します。SciPy のように NumPy を包含するようなライブラリや、NumPy を必ずしも必要としないグラフィックスライブラリ matplotlib のようなライブラリであっても、Numpy と親和性があり NumPy を使うと様々な処理がより簡単に実行できたり、数値計算が必要な場合に NumPy（あるいは同等な機構）を呼び出すなどの連携を行っているようなライブラリをここでは NumPy ファミリと称しています。

なお、ライブラリ、パッケージ、モジュールはプログラミング言語において呼び出し可能な関数やクラス群の集合体の意味として用いられ、それらが提供する機能群は一般に

ライブラリ ⊃ パッケージ ⊃ モジュール

の包含関係があります。また、関数、モジュール、パッケージ自体を総称してライブラリということがあります。

1.1　本書で利用する拡張ライブラリ

このテキストでは以下の外部パッケージを使います。

- NumPy http://www.numpy.org
- matplotlib https://matplotlib.org
- SciPy https://www.scipy.org/
- pandas https://pandas.pydata.org

さらに、記号処理計算用の外部パッケージ

- SymPy https://www.sympy.org/

を積極的に併用して効率的な計算を試みます。

これらの外部パッケージのパッケージは次のように pip コマンドを使ってインストールすることができます（記号 '$' はターミナルのコマンドプロンプトです）[1)]。

```
$ pip install numpy, matplotlib, scipy, pandas
$ pip install sympy
```

ただし、Google Colaboratory ではこれらのライブラリは既にインストールされているので改めてインストールする必要はありません。

このテキストは Google Colaboratory を使って説明します。Colaboratory に予めインストールされている外部ライブラリを確認するには、コードセル (code Cell) 内で次のコマンドを実行します（行頭に記号 '!' があることに注意してください）。

```
!pip freeze
```

出力行に多くのインストール済みのライブラリが（390 行以上に渡って）表示されるはずです。また、Colaboratory で稼働している Python のバージョンを確認するコマンドは次のようになります。

```
!python --version
```

原稿執筆時点では Python 3.7.13 となっています（最新の Python 安定版は 3.10.6）。Colaboratory ではライブラリの開発状況や依存関係などを考慮して保守的な立場を取っています。

1.1.1 matplotlib を使ってみる

まず外部パッケージ matplotlib を使ってプロットするときのプロットデータの渡し方を紹介しながら、NumPy スタイルでのプログラミングに慣れていきましょう。

大量の数値データを視覚化してみると、その全体を俯瞰してその様子を理解・研究する手がかりを得られることがあります。目的とする計算を達成するには複数のプログラミング法があり得ますが、NumPy スタイルで実施することによってプログラムコードの簡素化と高速な処理を同時に実現することができます。

matplotlib のプロットモジュール matplotlib.pyplot を利用するときには、慣例にしたがっ

1) 外部ライブラリのインストールやインストール済みの拡張モジュールの確認の仕方については節 A.2.1 を参照してください。クラウドサービス Google Colaboratory を利用したり、Anaconda https://www.anaconda.com で Python 環境を整えた場合には本書で必要とする外部ライブラリのインストール作業は不要です。

てその短縮名を `plt` として次のようにインポートします。このテキストでは Colaboratory（あるいは Jupyter）を使うことを前提としているので、コードセルの先頭に記号 '@' で始まるマジックコマンドを次のように加えてます。

```
%matplotlib inline
import matplotlib.pyplot as plt
```

matplotlib を使うと 2 次元だけでなく 3 次元プロット、工夫すればアニメーション描画が可能になりますが、残念ながら Colaboratory では 3 次元描画はできません（ローカルにインストールした Python 環境では別ウィンドウに描画結果が開き、マウスドラッグで回転させることができる 3 次元プロットが可能になります）。

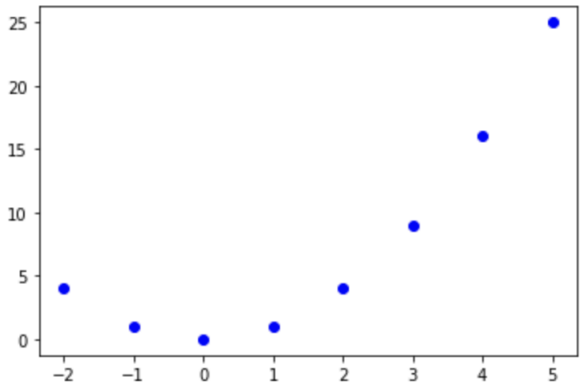


図 1.1　関数 $f(x) = x^2$ のグラフ：$y_i = x_i^2$ の関係にある平面上の点列 $\{(x_i, y_i)\}$ としてプロット

ごく簡単な例で NumPy スタイルのプログラミングを行ってみましょう。関数 $f(x) = x^2$ のグラフを図 1.1 のようにプロットするためには、x-軸上の n-個の点列 $x_0, x_2, \ldots, x_{n-1}$ を与えて各点 x_i に対応する関数値 $y_i = f(x_i)$ を求めて、平面上の点列 $(x_0, f(x_0)), (x_1, f(x_1), \ldots, (x_{n-1}, f(x_{n-1})$ を matplotlib のプロット関数 `plot()` を用いて描きます。計算アルゴリズムはこれに尽きます。

プロット図 1.1 を得るプログラムとして以下に `points_plot-1`, `points_plot-2` と `points_plot-3` の 3 種類のコードを考えました。それぞれ 1 つのコードセルに複数行にわたって完全な形で入力して、どれかのコードセルを実行するだけで描画されるようにしています（先頭のマジックコマンドがあれば、末尾の `plt.show()` は不要なのでコメントアウトしてあります）。そのために各コードセルでライブラリ matplotlib をインポートしています。

最初のコード 1.1-1 は、x-軸上の点列 $x_0, x_2, \ldots, x_{n-1}$ をリスト `xlist` $= [x_0, \ldots, x_{n-1}]$ で与えて、リストの各要素 `xlist[i]` ごとに関数値 `func(xlist[i])` を計算して平面上

に点を求めて plot(xlist[i], f(xlist[i])) 毎にプロット関数を呼ぶ素朴なプログラムです。リスト点列の数だけプロット関数 plot() の呼び出しが必要です。

コード 1.1-1　points_plot-1

```
%matplotlib inline
import matplotlib.pyplot as plt

def func(x):
    return(x ** 2)

xlist = [-2, -1, 0, 1, 2, 3, 4, 5]

fig, ax = plt.subplots()
for i in range(len(xlist)):
    ax.plot(xlist[i], func(xlist[i]),'bo')
#plt.show()
```

コード 1.1-2 は、平面上の x-成分リスト xlist $= [x_0, \ldots, x_{n-1}]$ に対応する関数値として y-成分リスト ylist $= [func(x_0), \ldots, func(x_{n-1})]$ を先に計算しておき、この x, y-成分双方のリストをプロット関数 plot() に引数として渡し、plot(xlist, ylist) と 1 回呼び出して描画しています。プロットすべき点の数によらずにプロット関数 plot() は 1 回だけ呼び出されます。

コード 1.1-2　points_plot-2

```
%matplotlib inline
import matplotlib.pyplot as plt

def func(x):
    return(x ** 2)

xlist = [-2, -1, 0, 1, 2, 3, 4, 5]
ylist = [func(x) for x in xlist]

fig, ax = plt.subplots()
ax.plot(xlist, ylist,'bo')
#plt.show()
```

改めて第 3 章で取り上げるように、プロット関数 plot() の呼び出し回数を減らすことは計算速度に大きく影響することがわかります。コード 1.1-2 では plot() による描画は効

率化されましたが、ylist を計算するために for 文を使っています。

1.1.2 NumPy のベクトル化計算

NumPy を利用するときには、慣例にしたがってその短縮名を np として次のようにインポートします。

```
import numpy as np
```

コード 1.1-3 は、数リスト $[x_0, \ldots, x_{n-1}]$ から NumPy の np.array() を使って生成した **NumPy 配列** (ndarray) として x-軸成分 xnlist を与え、func(xnlist) として対応する関数値である y-軸成分を一度に NumPy 配列として計算してプロットしています（コード 1.1-2 の要素ごとに関数値を求めるための繰り返し for が不要になりました)。こうした計算手法を NumPy の**ベクトル化計算**といいます。

matplotlib を使うプロットではこうした NumPy スタイルで実施すると、コード記述を簡素化できる（意図がわかり易くなる）だけでなく、実行速度の向上をもたらします。

コード 1.1-3　points_plot-3

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

def func(x):
    return(x ** 2)

xnlist = np.array([-2, -1, 0, 1, 2, 3, 4, 5])

fig, ax = plt.subplots()
ax.plot(xnlist, func(xnlist),'bo')
#plt.show()
```

NumPy 変数同士の加減乗除や組み込み関数の演算ではこうしたベクトル化計算が可能です。NumPy 配列（一般には多次元配列）の構造を利用して要素ごとの関数値を一度にベクトル化計算する関数を**ユニバーサル関数** (universal function) と呼びます。NumPy のベクトル化計算によってコードが簡素になり、プログラミングの意図（関数値をプロットする）が伝わりやすくなっています。

この例では関数 f の引数の 1 つだけが NmPy 配列のとき（他にも引数があるときでも 1 つの数や文字列をスカラーとして使うとき）は自明なベクトル化計算として配列要素 $\{x_{ijk\ldots}\}$ ごとの関数値 $\{f(x_{ijk\ldots})\}$ を返します。また、同じ型を持つ配列同士の加減乗除演

算+ − * /も要素ごとに実行できます。一般の 2 つ以上の異なる形状を持つ NumPy 配列を引数を与えた場合にどのようにしてベクトル化計算するかについて NumPy では大変巧妙で一貫した考え方が採用されています。この演算時に形状の異なる配列をどのように扱うかを NumPy の**ブロードキャスト規則** (broadcast rule) と言い、節 1.5 で改めて取り上げます。。

1.2 NumPy を使った計算の高速化と実行時間測定

Python では、前もって利用する変数の型宣言を必要としない動的な型付けが行われるため初心者でも使い易く、柔軟なプログラミングが可能です。標準パッケージ typing（**型アノテーション**：型ヒント） を使って、定義した関数引数や戻り値の型や途中で利用するデータ型について明示することができますが、実行時には型を強制しないため、型アノテーションだけで実行速度が早くなるわけではありません（コードの可読性の低下や意図を超えた誤用コードの混入を防ぐためにこのような型アノテーションの運用は望ましいと言われています)。

1.2.1 計算の高速化

プログラミングにおける実行速度の高速化には以下のような段階が考えられます。

(1)言語仕様と標準ライブラリの範疇内で計算アルゴリズムを検討する。
(2)実績のあるライブラリを活用する。
(3)既存のライブラリ内部を研究して新しく調整・開発する。
(4)実行されるバイトコードを最適化する。

やみくもに高速化を目指すのではなく、目的とする計算を実施するための取り掛かりの良さ（ハードルの低さ)、コードの可読性、情報交換コミュニティ、必要なコンピュータ資源などとのバランス関係の中で高速化を検討する必要があります。段階 (3), (4) は初心者の範囲を超えてしまいます。また、段階 (1) についても様々なアルゴリズムの一般論や計算評価に関する広範な知識が必要になります。

動的な型付けが可能な Python プログラムでは、変数に代入されたデータ（整数・浮動少数・複素数などの数や文字列など）を計算中に利用する際に、それらの値が格納されている Python オブジェクトにアクセスして初めてデータ型を検知するというオーバーヘッド（ある処理を行うためにかかる一定のコスト）が避けられません。扱うデータ量が多くなるとその影響は顕著になります。

NumPy 計算では、Python の動的型付を可能とするためのオーバヘッドをなくして各要

素に同一の固定されたデータ型を持たせて、NumPy 配列専用に最適にチューニングされた関数やメソッドを使うことによってデータ保持や計算操作の高速化を実現しています。多次元 NumPy 配列に対して実施できるベクトル化計算はその恩恵です。ただし、NumPy 配列の特定の位置にある数値を取り出したり、NumPy 配列に数値を代入するような処理は Python 側からオブジェクトを通した処理を伴うため、NumPy を使ったからといってどんな計算でも高速に実施されるわけではありません。

1.2.2 実行時間の計測

NumPy を使うと通常に考えた数値計算が劇的に改善されることを 2 つの方法で確かめてみましょう。1 つは標準モジュール time を使ってシステム時間を取得する方法、もう 1 つは Colaboratory（あるいは Jupyter）を使う場合に IPython カーネルで提供されるマジックコマンド`%timeit`（または`%%timeit`）を使う方法です。

1.2.2.1 システム時間の取得を使う `time.time()`

標準モジュール time に用意されている `time.time()` は、エポックタイム（1970 年 1 月 1 日 0 時 0 分 0 秒）を起点とする経過時間を秒 ([sec]) を単位とする不動小数で返します。これを利用して時間計測するステートメントの前に `start = time.time()` と計測終了後のステートメントとして `elapsed_time = time.time() - start` を挟んだコードを書いて実行時間を計測することが考えられます。しかし、システム時間取得のシステムコールはシステム依存し、すべてのシステムが 1 秒より高い精度で時刻を提供するとは限らず、高い精度で実行時刻を提供するわけではありません。

コード 1.2-1 は、NumPy のパッケージ numpy.random を使って標準 10 万個からなる実数乱数を生成し各要素を 2 乗する計算を、まずリスト内包表記で for を使って生成した乱数をリスト `rndlist` に格納しておき、(1) Python の標準的方法としてリスト `rndlist` から 1 つずつ要素を取り出して 2 乗計算した結果をリスト `sqlist` に追加する方法と、(2) リスト `rndlist` を NumPy 配列化して `nplist` としベクトル化計算で一度に 2 乗計算して得られる NumPy 配列 `npsqlist` をリスト化計算するというう 2 つの計算方法の実行時間を `time.time()` で比較測定しています。コードセルを実行した結果も示しました。

今の場合、乱数の並びを 2 乗した数の並びを得たいのであれば、わざわざ Python リストから NumPy 配列化してベクトル化計算した結果を再びリスト化する必要はなく、NumPy を使えば次のように乱数配列を直接与えて一気に計算することができ計算時間もうんと短縮することができます。

```
rndlist = np.random.random(100000)
npsqlist = func(nplist)
```

それでも、for とリスト要素の追加メソッド.append() を使って要素毎の計算をするよりも NumPy 配列化して一度に計算する方法の方が計算時間は大幅に短縮されるばかりか、コードの可読性（計算意図の理解）は大幅に向上することに注意してください。

コード 1.2-1　compare_time

```
import time
import numpy as np

def func(x):
    return(x ** 2)

rndlist = [np.random.random() for i in range(100000)]

print('Without NumPy by time.time():')
start = time.time()
sqlist = []
for x in rndlist:
    sqlist.append(func(x))
print('\telapsed time[sec] = ', time.time() - start)

print('Using NumPy by time.time():')
start = time.time()
nplist = np.array(rndlist)
npsqlist = func(nplist)
res = npsqlist.tolist()
print('\telapsed time[sec] = ', time.time() - start)
```

```
Without NumPy by time.time():
        elapsed time[sec] =  0.04780459403991699
Using NumPy by time.time():
        elapsed time[sec] =  0.01944708824157715
```

1.2.2.2　マジックコマンド　%timeit %%timeit

Colaboratory（あるいは Jupyter）を使う場合、スクリプトの実行時間の計測には IPython カーネルが提供するマジックコマンド%timeit（または%%timeit）を使うとステートメントの実行時間をさらに精密に測定し、しかも手軽に行うことができます。

1 行のステートメントの実行時間計測をするには、コードセルに%timeit に続けて 1 つのステートメントを 1 行に書いて実行します。複数行（1 行でもよい）からなるステー

トメントの実行時間を計測するには、コードセル内で 2 つの%が付いた%%timeit で改行して次行以降に 1 つ以上のステートメントを書くと複数ステートメントの実行時間を計測します。具体的には、先のコード 1.2-1 に引き続いてコードセルを次のように書いて実行します。

```
%timeit npsqlist = func(nplist)
```

```
126 μs ± 2.18 μs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%%timeit
sqlist = []
for x in rndlist:
    sqlist.append(func(x))
```

```
42.6 ms ± 645 μs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

%timeit は指定したステートメントを複数回実行してその統計平均値を返します。[ms] はミリ秒（10^{-3}sec）、[μs] はマイクロ秒（10^{-6}sec）ですから、関数値の計算は要素毎に計算するよりも NumPy を上手く使うと 100 倍以上（今の場合は 400 倍程度）早くなることがわかりました。

1.3 NumPy の多次元配列

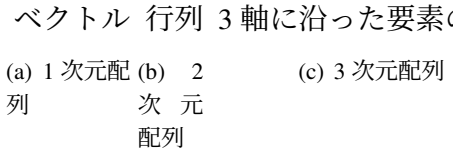
NumPy 配列は複数の要素を格納しておくための多次元的に配置した容れ物で、異なる型を並び要素とできるネストされたリストとは異なり、NumPy 配列要素はすべて同じデータ型です。NumPy ではこのような多次元配列を化膿したデータクラスを NumPy 配列または **ndarray**(N-dimennsional array) と呼んでいます。NumPy の公式ドキュメントにある NumPy: the absolute basics for beginners に NumPy の基本的な使い方と考え方が要領よく紹介されており、是非一読してください。

NumPy 配列の要素となるデータ型は整数（int/uint)、浮動少数 (float)、複素数 (complex)、ブール値 (bool)、文字列 (unicode) あるいは Python オブジェクト (object) のどれか 1 つのデータ型を持つと理解しておけば十分です。

1.3.1 NumPy 配列の構造

NumPy 多次元配列には要素を配置するために複数の**軸** (axis)（座標軸のようなもので第 0 軸、第 1 軸、.... といいます）が用意され、配列要素は第 0 軸にそって i 番目、第 1 軸に沿って j 番目の要素を、第 2 軸に沿って k 番目の要素をというようにして配列要素が配置されます（図 1.2 参照)。この NumPy 配列が持つ軸の数を NumPy の**配列次元** (dimension)

と呼びます[2)]。同様に 4 次元、5 次元と高次元配列を考えることができます。一方、0 次元 NumPy 配列は軸をもたず 1 つの値だけを持ち、スカラーと考えます（節 1.3.2 の最後を参照）。



(a) 1 次元配列　(b) 2 次元配列　(c) 3 次元配列

図 1.2　NumPy 配列の構造。(a) ベクトルは第 0 軸にだけ要素が並んいる 1 次元配列、(b) 行列は第 0 軸と第 1 軸に要素が配置されている 2 次元配列、(c) 3 次元配列は 3 つの軸に沿って要素が並んでいる。

特に 1 次元 NumPy 配列を**ベクトル** (vector) と呼び、第 0 軸に n 個の要素が並んでいます（n-次元ベクトル）。2 次元 NumPy 配列は第 0 軸と第 1 軸に沿って平面的に要素が配置されており、各軸に沿った要素の数がそれぞれ m, n 個のとき $m \times n$-行列と呼ぶことがあります（第 0 軸方向を行、第 1 軸方向を列とみなします）[3)]。標準 Python と同じく NumPy 配列では要素位置インデックスは 0 から始まります。

n 次元 NumPy 配列 a の要素へのアクセスは、配列次元 $n \geqq 1$（軸の数）個からなる添字の並び $i_0, i_1, \ldots, i_{n-1}$ を使って

$$\mathrm{a}[\underbrace{i_0, i_1, \ldots, i_{n-1}}_{\text{配列次元の個数分}}] \Leftarrow \text{第 0 軸の } i_0 \text{ 番目、第 1 軸の } i_1 \text{ 番目,...、第 } (n-1) \text{ 軸の } i_{n-1} \text{ 番目}$$

で行います。

数学ではベクトルやテンソルのような多次元配列の要素を a^i, a_j, a^i_j、$a^{i_0,\ldots i_{k-1}}{}_{i_k,\ldots,i_{n-1}}$ のように記すことがあります[4)]。しかしながら NumPy では、多次元配列をこのような添字の上下に関わりなく各軸に沿って配列要素の位置を $[i_0, i_1, \ldots, i_{n-1}]$ で指定するという形で値を格納するための容器であると考えます実際、2.2.1.1 で改めて紹介するように、ベクトル（1 次元 NumPy 配列）では列ベクトルや行ベクトルの区別はありません（それでも目的とする計算がきちんと達成されます）。

2） NumPy では NumPy 配列に割り当てられた軸の数を配列次元といい、NumPy 配列の各軸それぞれに並んでいる要素の総数はサイズ（size）として定義され次元ではありません。たとえば、一つの軸の沿って n 個の要素が並んだ配列は 1 次元 NumPy 配列であり、n 次元 NumPy 配列ではありません。このような 1 軸だけを持つ NumPy 配列の形状を特にベクトルと称し、n 次元ベクトルと言うことがあります。

3） NumPy: the absolute basics for beginners https://numpy.org/doc/stable/user/absolute_beginners.html には NumPy 配列の様子を図示してわかりやすく説明しています。

4） 空間論の立場では、成分表記は座標系の採り方に取り方に依存し、それら成分が変換 U によって互いにどのように関係するかを問題にします（たとえば $a^i = \sum_j U^i{}_j b^j$ のように、下付き添字で表す成分を共変性 (covariant)、上付き成分で表す成分を反変性（contravariant）と区別します。

1.3.2 NumPy 配列の基本属性 ndarray.ndim ndarray.shape ndarray.size

numpy.array(plist) は、ndarray のコンストラクタ array を使って長さとネストの深さが揃ったリスト plist からを NumPy 配列を生成します。以下の説明では array の適用に当たってネストの深さが揃っていないリスト [[1, 2, 3], [4, 5]] や [[1, 2, 3], [[4, 5], 6]] の NumPy 化は考えないとします。たとえば

```
import numpy as np
x = np.array([[1,2,3], [4,5]])
```

を実行しようとしても、要素の補填は行われず破綻したネスト列であるという VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences の警告が出て、意図した計算ができなくなります。

NumPy 配列からその属性が取得できます。メソッド .shape が返す整数並びのタプルは**配列形状**（または shape）といい、 各軸に並べた要素数の並び (tuple of array dimensions) を表します。メソッド .ndim が返す非負整数は **ndim**(次元: number of array dimension) といい、NumPy 配列軸の数を表します（数の並びである shape の要素数)。また、メソッド .size が返すに非負整数は、NumPy 配列をなす全要素数で、 **size**（サイズ: 要素数 number of elements in the array）といい、数の並びである shape の各数の積に一致します。

今後の計算では NumPy 配列の形状 shape の各値と軸の数（次元）に注意を払ってください。得られた計算結果が意図するものであるかチェックするために最初に確認すべき事項です。NumPy 配列を与えて、その shape、ndim、size を確かめてみましょう。次の例で、3 つの数 1,2,3 を並べただけのように見える NumPy 配列 x と x1 の違いに注目してください。

```
import numpy as np
x1 = np.array([1,2,3])
x1.shape # 1次元配列=3次元ベクトル
```

```
(3,)
```

```
x1.ndim
```

```
1
```

```
x1.size
```

```
3
```

```
x2 = np.array([[1, 2, 3]])
x2.shape # 1行3列の配列
```

```
(1, 3)
```

```
x2.ndim
```

```
2
```

```
x2.size
```

```
3
```

`x1 = np.array([1, 2, 3])` のように、n 個の数が並んだリストから得られる NumPy 配列は第 0 軸方向（垂直）に n 段に 1 つず数が並んだ 1 次元配列で、その形状 `x1.shape` は $(n,)$ であることがわかりました。形状 $(n,)$ のタプル長（数の並びの長さ）は 1 であり、その配列次元 `x1.ndim` が 1 であることの辻褄が合います。一方、`x2 = np.array([[1, 2, 3]])` のように、n 個の数が並んだ 1 つのリスト `[1,2,...,n]` を要素とするリストから得られた NumPy 配列の形状は $(1,n)$ となって長さ 2 のタプル持つことから 2 次元配列であることがわかりました。`x2` は第 0 軸方（垂直）に 1 段、第 1 軸（水平）に 3 列並んでいる点が 1 次元配列 `x1` とは違います。

同じように考えると、次のように `a = np.array([[1, 2, 3], [4, 5, 6]])` の形状が `(2,3)` であることが理解できます。

```
a = np.array([[1, 2, 3], [4, 5, 6]])
a
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
a.shape # 2行3列配列
```

```
(2, 3)
```

```
a.ndim
```

```
2
```

```
a.size
```

```
6
```

1 次元 NumPy 配列（ベクトル）は線形代数で言う列ベクトルや行ベクトルの区別はありませんが、今の場合、演算子 `@` を使って 2 次元 NumPy 配列 `a` と 1 次元 `x1` との行列積 `a @ x1` を正しく計算します（節 2.2.1.1 参照）。

```
a @ x1
```

```
array([14, 32])
```

```
(a @ x1).shape # 2次元ベクトル
```

```
(2,)
```

改めて節 2.2.1 で説明しますが、NumPy では NumPy の計算手順にしたがって計算する限り、列ベクトルや行ベクトルとを区別する必要はありません。

もう少しだけ踏み込んでみましょう。`x3 = np.array([[1], [2], [3]])` は、配列 `x1` や `x2` と要素の並びは同じですが 2 次元 NumPy 配列です。

```
x3 = np.array([[1], [2], [3]])
x3
```

```
array([[1],
       [2],
       [3]])
```

```
x3.shape # NumPyでは3行1列の2次元配列
```

```
(3, 1)
```

```
x3.size
```

```
3
```

この 3×1 の 2 次元配列 `x3` は、2×3 の 2 次元配列 `a` との行列積計算 `a @ x1` から 2×1 の 2 次元 NumPy 配列が得られます。線形代数としては $m \times n$ 行列と n-次列ベクトルとの行列積としてら m-次列ベクトル（1 次元配列）を得たとしたいのですが、ここでは `x3` がベクトルでなく 2 次元配であることに注意してください。こうした課題については節 2.2.1.1 で改めて再考します。

```
a @ x3
```

```
array([[14],
       [32]])
```

```
(a @ x3).shape
```

```
(2, 1)
```

大きさだけを持ち方向に依らない量（ただの数）である数 (スカラー: scalar) の NumPy での取り扱いについて触れておきます。方向に依らないことは NumPy では軸数が 0 であると考えます。実際、以下の例で NumPy 配列を生成するコンストラクタ `np.array` にリストでなく、ただの数（たとえば 5）を渡してみます。

```
s = np.array(5)
a
```

```
array(5)
```

```
s.shape
```

```
()
```

```
s.ndim
```

```
0
```

```
s.size
```

```
1
```

```
type(s)
```

```
numpy.ndarray
```

スカラーから生成した NumPy 配列はその shape や ndim から 0 次元配列であることがわかりました。また、空リストから NumPy 配列を生成することも可能です。

```
null = np.array([])
null.shape
```

```
(0,)
```

```
null.ndim
```

```
1
```

```
null.size
```

```
0
```

1.3.2.1 複素数の取り扱い

2 行 2 列の Pauli 行列の 1 つ σ_Y

$$A = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

は複素数 i を要素に持ちます。Python には複素数型が complex が用意されており、NumPy でも同様に利用できます。Python では**虚数単位**として ‘j’（小文字） を使います（数学記法の ‘i’ とは異なります)。しかも、純虚数 i を `1j` と記す必要があり、`j` の前の 1 は省略できません。

Pauli 行列 σ_Y を NumPy では次のように与えることができます。

```
pauliY = np.array([[0, -1j], [1j, 0]])
pauliY
```

```
array([[ 0.+0.j, -0.-1.j],
       [ 0.+1.j,  0.+0.j]])
```

1.3.3 NumPy 配列を生成する便利な関数

プログラム上の動作確認のために NumPy 要素を直接タイプしたりファイルからデータを読み込む場合などを除けば、しばしば以下のような NumPy 配列を与える関数を便利に使ってコードを明快に簡素化することができます。

1.3.3.1 区間内を等間隔で並ぶ配列 `numpy.arange`

`numpy.arange([start,] stop, [step])` は指定した区間内 $[\mathrm{start}, \mathrm{stop})$ に等間隔 step に並んだ値を 1 次元 NumPy 配列として返します。右端の終端値 stop は生成される配列要素には含まれません。区間左端 start が指定されると配列開始値は start から始まり、省略されたときは 0 から始まります。刻み間隔 step が省略されたときのデフォルト間隔は 1 です。

```
import numpy as np


np.arange(-3, 5)
```

```
array([-3, -2, -1,  0,  1,  2,  3,  4])
```

```
np.arange(-3.3, 5.1)
```

```
array([-3.3, -2.3, -1.3, -0.3,  0.7,  1.7,  2.7,  3.7,  4.7])
```

```
np.arange(-3.3, 5.1, 1.5)
```

```
array([-3.3, -1.8, -0.3,  1.2,  2.7,  4.2])
```

1.3.3.2 区間内に指定個数分だけ等間隔に並ぶ配列 `numpy.linspace`

`numpy.linspace(start, stop, num=Numb[, endpoint=False])` は与えられた区間内 $[\mathrm{start}, \mathrm{stop})$ に等間隔に Numb 個並んだ 1 次元 NumPy 配列を返します。オプション指定 endpoint = False がなければ（デフォルト）配列は区間の端点を含み、点列の公差は $(\mathrm{stop} - \mathrm{start})/(\mathrm{Numb} - 1)$ となります。一方、オプション endpoint = False をつけたときは、点列は区間の右端を含まず、点列の公差は $(\mathrm{stop} - \mathrm{start})/\mathrm{Numb}$ となります。

```
np.linspace(-3, 5, num=9)
```

```
array([-3., -2., -1.,  0.,  1.,  2.,  3.,  4.,  5.])
```

```
np.linspace(-3, 5, num=8, endpoint=False)
```

```
array([-3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

1.3.3.3 配列形状の変更 numpy.reshape ndarray.reshape

numpy.reshape(a, newshape) は NumPy 配列 a を、または メソッド ndarray.reshape(newshape) は NumPy 配列 ndarray を、タプルで指定した新たな配列形状 newshape を持つ NumPy 配列へと作り替えます。ただし、元の配列形状と新たな newshape とで配列の size は一致しなければなりません。また、ndarray.reshape(m, -1) とすると、$m \times k$ の 2 次元 NumPy 配列が得られますただし、k は mk が元の配列 size に一致するように選ばれるため、m は元の配列 size の約数でなければいけません。

```
np.arange(12).reshape(3,4)
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
np.arange(12).reshape(2,3,2)
```

```
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
        [10, 11]]])
```

```
np.arange(12).reshape(3,4).reshape(2, -1)
```

```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

NumPy 配列 a の shape が $(s_0, s_1, \ldots, s_{n-1})$ のとき、各軸の shape 値の積 $s_0 s_1 \cdots s_{n-1}$ が配列要素の総数 a.size となっています。配列を reshape したとき size を保つ整合性が保証されるときは、配列次元 ndim の変更も可能です。たとえば、shape (2,3) を持つ 2 次元配列 a は a.reshape(2,1,3,1) と配列形状を変えて 4 次元配列にすることができます。

```
np.arange(6).reshape(2,1,3,1).ndim
```

```
4
```

1.3.3.4 未初期化の配列 numpy.empty

numpy.empty(shape) は、指定した配列形状 shape を持つ NumPy 配列でその要素を特定の値で初期化しない未初期化 NumPy 配列を返します。

```
np.empty((2,3))
```

```
array([[0.0e+000, 4.9e-324, 9.9e-324],
       [1.5e-323, 2.0e-323, 2.5e-323]])
```

numpy.empty で生成される NumPy 配列の要素値は不定であるだけで（実行しても同じ値になるとは限りません）、空配列を返すわけではなく NumPy 配列を使う計算のための配列形状が確保されるだけです。意味ある計算を実施するためには、その後に各配列要素に必要な値を代入（初期化）する必要があります。

numpy.empty の実行速度は numpy.ones や numpy.zeros などに比べてうんと高速であるため、コード進行上で配列要素を特定の値にセットする必要がない場合に np.empty が利用されます。たとえば、コード 2.3-1 や節 3.2.2 で紹介するように、大きさがあらかじめわかっている NumPy 配列要素を逐次的に計算する場合に有用です。

1.3.3.5 すべての配列要素が 1 または 0 の配列 numpy.ones numpy.zeros

numpy.ones(shape) は、指定した配列形状 shape でその配列要素がすべて 1 の NumPy 配列を返します。配列形状を (n,) または単に数 n を渡したときは n 個の要素からなる 1 次元 NumPy 配列となります。オプションでデータ型を'int64' などと指定しない限り、配列要素は浮動小数で与えられます。

```
np.ones((2,3,4))
```

```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

```
np.ones((5,), dtype='int64')
```

```
rray([1, 1, 1, 1, 1])
```

同様に、numpy.zeros(shape) 指定した指定した配列形状 shape でその配列要素がすべて 0 の配列を返します。

```
np.zeros((2,3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

1.3.3.6 単位配列 numpy.eye numpy.identity

numpy.eye(N[, M=None, k=0]) は、M だけを指定ししたとき（デフォルト値 $\mathrm{M} = \mathrm{N}, \mathrm{k} = 0$）$N \times N$ の対角要素が 1 の 2 次元 NumPy 配列（単位行列）を返します。非負整数の M が指定されると（k は指定しない）、$N \times M$ の対角要素に 1 が並んだ $N \times M$ 行列が返ります。$\mathrm{k} \neq 0$ に指定すると、行列を $(a_{i,j})$ と記したとき、$k > 0$ のときは「1 が並ぶ対角線」位置が上方に（$a_{i,i+k} = 1$）、また $k < 0$ のときは下方に（$a_{i-k,i=1}$）にシフトします。

```
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
np.eye(5,4)
```

```
rray([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.]])
```

```
np.eye(4, k=1)
```

```
array([[0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.]])
```

```
np.eye(4,k=-1)
```

```
array([[0., 0., 0., 0.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

numpy.identity(N) は $N \times N$ の対角要素が 1 の単位行列を返します。

```
np.identity(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

1.3.3.7 対角配列 numpy.diag

numpy.diag(v[, k=0]) は、Numpy 配列あるいはリスト v で指定された対角配列を返します。非負整数 k が指定されたときは、numpy.eye（節 1.3.3.6）の場合と同じように「対角線」が上下します。

v が 1 次元配列のときは、それら要素を対角成分とする 2 次元配列（対角行列）、v が 2 次元配列のときは、その対角成分をだけを含む 1 次元配列を返します。

```
np.diag(np.array([1,2,3]))
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

```
v = np.arange(9).reshape(3,3)
v
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
np.diag(v)
```

```
array([0, 4, 8])
```

```
np.diag(v,k=1)
```

```
array([1, 5])
```

1.3.4 疑似乱数配列

コードの検証などで適当な値が欲しい場合、疑似乱数 (pseudo random number) の利用はたいへん重宝します。NumPy の乱数パッケージ numpy.random のごく簡単な使い方を紹介します。

1.3.4.1 一様乱数 numpy.random.rand

numpy.random.rand(d0,...,dn) は、配列形状 $(\mathrm{d0},\ldots,\mathrm{dn})$ を持つ配列要素として区間 $[0,1)$ 上の一様分布に従う乱数からなる NumPy 配列を返します。

```
np.random.rand(2,3)
```

```
array([[0.74513857, 0.99891941, 0.08471666],
       [0.2586736 , 0.02496878, 0.83462983]])
```

1.3.4.2 正規分布 numpy.random.randn　numpy.random.normal

numpy.random.randn(d0,...,dn) は、配列形状 $(\mathrm{d0},\ldots,\mathrm{dn})$ を持つ配列要素が平均 0、分散 1 の標準正規分布に従う乱数からなる NumPy 配列を返します。

```
np.random.randn(5)
```

```
array([ 0.13796983,  1.6437732 ,  0.21351329, -2.39785189,  0.11233496])
```

numpy.random.normal(mu, sigma, num) は、平均 mu、標準偏差 sigma の正規分布 $N(\mu,\sigma^2)$ [5] にしたがう num 個の乱数を返します。

変数 X が 平均 μ、分散 σ^2 を持つ正規分布 $N(\mu,\sigma^2)$ に従っているとき、変換 $Y=aX+b$ の分布は $N(a\mu+b,a^2\sigma^2)$ に従い、とくに、標準化変数 $Z=\frac{X-\mu}{\sigma}$ は標準正規分布 $N(0,1)$ に従います。

これより、平均 0、分散 1 の標準正規分布 $N(0,1)$ にたいして正規分布 $N(\mu,\sigma^2)$ は次の変換として表されます。

$$\mu+\sigma N(0,1)$$

たとえば、次のスクリプトは $N(50,10^2)$ に従う 10000 個の疑似乱数配列のヒストグラムを標準正規乱数を使って描きます（図 1.3(a)）。

```
import numpy as np
import matplotlib.pyplot as plt

plt.hist(50 + 10 * np.random.randn(10000), bins=20)
#plt.show()
```

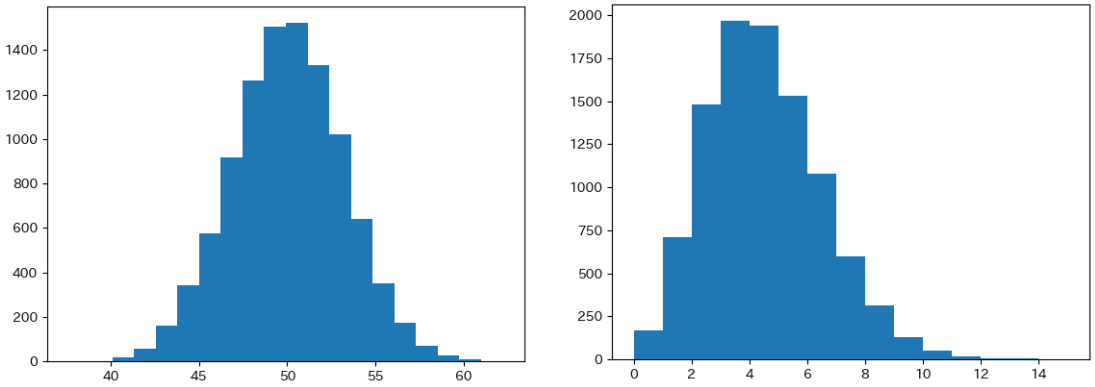
5) 統計学では正規分布を分散 σ^2 を使って $N(\mu,\sigma^2)$ と表記しますが、NumPy ではこれを $N(\mu,\sigma)$ とパラメータを標準偏差を使って渡します。実際の利用ではこうしたパラメータの与え方はマニュアルで確認してください。

1.3.4.3 Poisson 分布 `numpy.random.poisson`

`numpy.random.poisson(lam, k)` は、平均値 lam を持つ Poisson 分布に従 k 個の乱数からなる 1 次元 NumPy 配列を生成します。次の Colaboratory（または Jupyter）コードは平均 4 の個数 10000 個の Poisson 分布のヒストグラムを描きます（図 1.3(b)）。

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

plt.hist(np.random.poisson(4, 10000), bins=15)
#plt.show()
```



(a) 正規分布　　(b) Poisson 分布

図 1.3　(a) 正規分布 $N(50, 10^2)$ に従う 10000 個の疑似乱数を `50 + 10 * numpy.random.randn(10000)` で生成して、Matplotlib でヒストグラム表示。階級数 (bins) は 20 としている。(b) Poisson 分布 $P(\lambda, k)$ の $k = 10000$ 個の疑似乱数を `numpy.random.poisson(4, 10000)` で生成して、Matplotlib でヒストグラム表示（階級数 (bins) は 15 としている）。

1.3.5 NumPy 配列要素のアクセスとスライス

節 1.3.1 で紹介したように、NumPy 配列の要素へのアクセスは配列次元 $n \geqq 1$ の数（軸数）だけの添字を使って $\mathrm{a}[i_0, i_1, \ldots, i_{n-1}]$ のように各軸に沿った要素並びの場所（位置インデックス）をカンマ (',') で区切って指定します。この操作を**スライス** (slice) といいます。Python 標準のネストされたリスト `plist` に対する要素指定 $\mathrm{plist}[i_0][i_1]\cdots[i_{n-1}]$ よりも簡素化されるだけでなく、さらに軸をまたいで指定した範囲の要素をまとめて取り出した NumPy 配列（たとえば、行列の指定した列の取り出し）を得ることができます。

軸に沿って要素取り出しの**範囲指定**をするためにコロン (':') を使います。位置インデックスを `start:stop[:step]` で範囲指定したスライスによって、インデックス start から step ごとに要素を拾いながら stop 未満のインデックスの配列要素を取りだします。スライス範囲を省略した際のデフォルト値は、start = 0, stop = その軸の shape 値, step = 1 として扱われます（ただし、スライス時にはコロンは省略できません)。

1.3.5.1 1 次元 NumPy 配列のスライス

1 次元配列のスライスは単リストに対しても同様です。

```
import numpy as np

a = np.arange(10)
a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
a[1:5]
```

```
array([1, 2, 3, 4])
```

```
a[:5]
```

```
array([0, 1, 2, 3, 4])
```

```
a[5:]
```

```
array([5, 6, 7, 8, 9])
```

step を負にすると、start と stop のデフォルト値の値が入れ替わります。この性質を使って配列要素を反転することができるので Python にはリスト反転のメソッドは用意されていません。

```
a[::-1]
```

```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

```
a[4::-2]
```

```
array([4, 2, 0])
```

1.3.5.2 多次元 NumPy 配列のスライス

多次元 NumPy 配列におけるスライス機能は強力です。標準 Python の範囲では、行列から行の取り出しは容易ですが、列を取り出すためにはわざわざコードを書く必要があります。

Shape (3,4) を持つ 2 次元 NumPy 配列 a から、直ちに a[:,1:3] とスライスして shape (3,2) の配列を得ることができます。

```
a = np.arange(12).reshape(3,4)
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
a[:,1]
```

```
array([1, 5, 9]
```

```
a[:,1:3] # shape (3,2)
```

```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

実際、shape (3,4) を持つ 2 次元 NumPy 配列 a と同じ深さを持つリスト plist から、第 1 軸（列）方向でスライスすることは面倒です（plist[:,1:3] とは実行できません）。plist を得るには、NumPy 配列を Python リストに変換するメソッド ndarray.tolist()（節 2.4.3 参照）を使って、次のように実施できます。

```
plist = a.tolist()
plist
```

```
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

スライス a[:,1:3] に相当することをリスト plist から計算するには、たとえば、次のようにまず転置配列リスト clist を求めた上で、スライス clist[1:3] して得たリスト dlist をさらに再び転置するというわかりにくい処理が必要です。

```
clist = [list(x) for x in zip(*plist)]# リスト plist の転置
clist
```

```
[[0, 4, 8], [1, 5, 9], [2, 6, 10], [3, 7, 11]]
```

```
dlist = clist[1:3] # 第1,2要素のスライス
[list(y) for y in zip(*dlist)] # リスト dlist の転置
```

```
 [(1, 2), (5, 6), (9, 10)]
```

1.3.6 NumPy 配列の連結、軸の追加

NumPy 配列同士を結合させて新たな NumPy 配列を構成したい場合があります。配列同士を結合する方法として、まず垂直（第 0 軸方向）に積み重ねるやり方と水平（第 1 軸方向）に連結するやり方の 2 つを紹介します。

1.3.6.1 垂直連結 numpy.vstack

numpy.vstack((v0,...,vk−1)) は、同じ shape を持つ NumPy 配列（またはリスト）$\{v_i\}$ の並びをタプル $(v_0,\ldots,v_{n-1})$ で指定して axis=0 の第 0 方向（行方向）に積み上げて結合した配列を返します。得られる配列の shape の第 0 方向の値が増えることになります。

```
import numpy as np

a0 = [ 0,  1,  2,  3]
a1 = [ 4,  5,  6,  7]
a2 = [ 8,  9, 10, 11]
np.vstack((a0, a1, a2)) # shape (3, 4)
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

積み上げる配列は、第 0 軸以降の shape が揃っていれば同じ shape の配列である必要はありません。第 0 軸の shape 値が積み上げた分だけ増えていることがわかります。

```
b1 = np.arange(12).reshape(3,2,2)
b2 = np.arange(4).reshape(1,2,2)
np.vstack((c1,c2)).shape
```

```
(4, 2, 2)
```

1.3.6.2 水平連結 numpy.hstack

numpy.hstack((v0,...,vk−1)) は NumPy 配列（またはリスト）$\{v_i\}$ の並びをタプル $(v_0,\ldots,v_{n-1})$ で指定して水平に（第 1 方向 axis=1 の方向である列方向に）結合した配列を返します。得られる配列の shape の第 1 方向の値が増えることになります。

```
np.hstack((a0, a1, a2))
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

ただし、$n \times m$ 列からなる 2 次元 NumPy 配列 a を第 1 軸（列）方向で各列ごとにスライス a[:,i] して取り出した 1 次元 NumPy 配列は第 0 軸（行）情報のみの shape (n,) を持ち、そのまま np.hstack で水平方向に並べても元の 2 次配列 a は復元できません。次のように、列ごとにスライスする際に右側に np.newaxis で新しく軸を追加して shape が (3,1) になるようにするか、shape 値が (3,1) になるように reshape した 2 次元配列を水平に並べると目的の配列を再構成できます。

```
a = np.arange(12).reshape(3,4)
h0 = a[:,0, np.newaxis] # shape(3,1)
h0
```

```
array([[0],
       [4],
       [8]])
```

```
h1 = a[:,1].reshape(3,1) # shape(3,1)
h2 = a[:,2].reshape(3,1)
h3 = a[:,3].reshape(3,1)
np.hstack((h0, h1, h2, h3))
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

1.3.6.3 連結 numpy.stack

numpy.stack((v0,...,vk−1)[,axis=0]) は、同じ shape を持つ NumPy 配列（またはリスト）$\{v_i\}$ の並びをタプル $(v_0,\ldots,v_{n-1})$ で指定して、配列次元を 1 つ上げて指定した軸（デフォルトは axis=0 の第 0 軸）に沿って積み上げます。

```
b0 = [0, 1, 2]
b1 = [5, 6, 7]
np.stack((b0, b1))
```

```
array([[0, 1, 2],
       [5, 6, 7]])
```

numpy.stack は numpy.vstack や numpy.hstack とは違った使い方をすることができます。

```
np.stack((b0, b1), axis=1)
```

```
array([[0, 5],
```

```
       [1, 6],
       [2, 7]])
```

興味深いのは 2 次元以上の配列の場合です。配列要素を積み重ねる軸方向による違いをその shape に注目して観察してみてください。

```
X = np.arange(12).reshape(3,4)
Y = np.arange(12,24).reshape(3,4)
np.stack((X, Y)) # shape (2, 3, 4)
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

```
np.stack((X, Y), axis=1) # shape (3, 2, 4)
```

```
array([[[ 0,  1,  2,  3],
        [12, 13, 14, 15]],

       [[ 4,  5,  6,  7],
        [16, 17, 18, 19]],

       [[ 8,  9, 10, 11],
        [20, 21, 22, 23]]])
```

今の場合、2 次元配列同士を `np.stack` しているので、第 0 軸（デフォルト）と第 1 軸方向以外に、最後の新しい軸として第 2 軸方向に連結することができます（`axis=-1` として最後の軸を指定できます)。

```
np.stack((X, Y), axis=2) # (3, 4, 2)
```

```
array([[[ 0, 12],
        [ 1, 13],
        [ 2, 14],
        [ 3, 15]],

       [[ 4, 16],
        [ 5, 17],
        [ 6, 18],
```

```
        [ 7, 19]],

       [[ 8, 20],
        [ 9, 21],
        [10, 22],
        [11, 23]]])
```

この例でわかるように、同じ shape を持つ配列同士を最後の軸で `np.stack` を取ると、対応する配列要素がペアとなって並ぶことになります。

1.3.6.4 軸の追加と配列次元の拡大 `newaxis`

これまで見てきたように、NumPy 配列の次元を拡大して ndim を 1 つ増やすには次の 2 つの方法がありました。1 つは目的の配列形状になるように明示的に reshape を使う方法（節 1.3.3.3 参照）、もう 1 つが `newaxis` を使って目的とする軸順位に軸を追加する方法（節 1.4.3）です。どちらの方法を使っても配置される配列の要素総数 size や要素順は保たれますが、その配置形状が変わります。

新たに追加する軸はどこでもよく、次のように記号 : で軸の全要素をスライスしながら、ある軸の直前に追加するときは 'np.newaxis, ' を、ある軸の直後に追加するときは ', np.newaxis' を挿入します。

```
x = np.array(5) # shape ()
x[np.newaxis] # shape (1,) <- ()
```

```
array([5])
```

```
a = np.arange(12).reshape(3,4)
an0 = a[np.newaxis, :] # shape (1, 3, 4) <- (3,4)
np.vstack((an0, an0)) # shape (2, 3, 4)
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]])
```

```
an1 = a[:, np.newaxis] # shape (3, 1, 4)
np.hstack((an1, an1)) # shape (3, 2, 4)
```

```
array([[[ 0,  1,  2,  3],
        [ 0,  1,  2,  3]],

       [[ 4,  5,  6,  7],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [ 8,  9, 10, 11]]])
```

```
an2 = a[:,:, np.newaxis] # shape (3, 4, 1)
np.vstack((an2, an2)).shape
```

```
(6, 4, 1)
```

```
np.hstack((an2, an2)).shape
```

```
(3, 8, 1)
```

NumPy 配列を reshape あるいはこのように軸を追加して配列形状を変化させても、配列の要素並びの順は

```
a.reshape(2,1,3,1,2).flatten()
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

のように元の配列の要素並びの順と同じです。

1.4 NumPy 配列同士のブロードキャスト

節 1.1.1 で紹介したように、NumPy ユニバーサル関数の機構を使うベクトル化演算によって、NumPy 配列の対応する要素ごとに関数値を高速に実行することができました。そこでは NumPy 配列同士の基本算術演算 +, *, −, /, //（整数商）, **（べき）, %（剰余）などが

- 同じ shape を持つ NumPy 配列の間で
- 対応する要素ごとに演算

として実行されました。例えば、次のような場合です。

```
import numpy as np

x = np.array([1, 2, 3]) # shape(3,)
y = np.array([3, 4, 5]) # shape(3,)
```

```
x * y
```

```
array([ 3,  8, 15])
```

しかしながら、NumPy では異なる配列形状を持つ配列同士で形状情報が公告されることによって計算可能な ‘互換性ある形状’ として計算可能となる場合があります。この NumPy の機構を**ブロードキャスト**といいます。次の計算は、ベクトル（1 次元配列）$(v_0, \ldots, v_{n-1})$ にスカラー（ただの数）s を掛けると、結果は各要素のスカラー倍 $(sv_0, \ldots, sv_{n-1})$ となることを示しています。

```
v = np.array([1, 2, 3]) # shape (3,)
s = np.array(4) # shape ()
s * v # shape (3,)
```

```
array([ 4,  8, 12])
```

同じ shape を持たずしかも異なる size の NumPy 配列同士の計算においても、対応する配列要素が補完され（**ブロードキャスティング**: broadcasting）、その結果として同じ shape を持つ配列要素ごとの計算として実行可能になるのです（今の場合、軸を持たないスカラーをかけると新たの軸が生成されて値が補完されると考えます）。

ブロードキャストを活用した効率の良い計算例については節 1.4.3 以降で紹介していきます。

1.4.1 ブロードキャスト規則

異なる shape を持つ NumPy 配列同士の計算において、どんなときにブロードキャスト可能であり、どのようにして NumPy 配列の shape を補完して計算が実行さるのでしょう。図を交えた説明が Numpy の User Guide の Broadcasting [6] にあります。

2 つの NumPy 配列の計算において、それらの shape 値を要素ごとに末尾（右端）から左端に向かってと比較していき条件

(1)すべて等しい
(2)それらの中に値 1 がある

を満たすとき**ブロードキャスティング可能**といいます。このため、NumPy では配列同士がブロードキャスト可能となるように、まず配列次元を揃えようとします。

(0)配列次元の小さい方の配列形状の shape 値の ‘左端に’1 を追加して軸を新設する（可能ならこれを繰り返す）

6) https://numpy.org/doc/stable/user/basics.broadcasting.html

これら (0), (1), (2) を**ブロードキャスト規則**と呼びます。これらに合致しないとき、エラー ValueError: operands could not be broadcast が通知されます。

以下で見るように、ブロードキャスティングによって同じ次元を持つ配列同士において（配列形状を右端から調べて）、ある配列の配列形状の shape 値が 1 で他の配列の対応する shape 値は 1 よりも大きいときには、shape 値 1 を持つ軸に沿った配列の並びを保ったまま shape 値が一致するまで（コピーされるように）引き延ばされます。その結果、ブロードキャスト規則が適用される配列同士では配列形状が一致し（shape 値の並びが同じになって）、その結果として対応する配列の要素毎の計算が可能になります。

具体的に見てみましょう。shape(3,4) を持つ 2 次元配列 a に、shape (4,) を持つ 1 次元配列 b を加えてみると次のように計算します。

```
import numpy as np

a = np.arange(12).reshape(3,4)
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
b = np.arange(4)
b
```

```
array([0, 1, 2, 3])
```

```
a + b
```

```
array([[ 0,  2,  4,  6],
       [ 4,  6,  8, 10],
       [ 8, 10, 12, 14]])
```

この計算は配列間でブロードキャスティングすることによって次のような手順で実施されたと考えます：まず次元の小さい方の配列 b に対して規則 (0) が適用されて shape (1,4) となって規則 (2) が適合し、新たに第 0 軸となって shape 値 1 を持つ軸方向に（各行が同じ 0,1,2,3 を持つよう）コピーされるように ‘引き伸ばされ’ て shape(3,4) となることで、同じ配列形状を持つことによって、要素毎に和が計算されたわけです。このブロードキャスト規則の適用を次のように示します。

	初期形状	次元一致	引き延ばし
a	3×4	3×4	3×4
b	4	1×4	3×4

ここで、shape (3,) を持つ 1 次元配列 `b0` を加算 `a + b0` しようとしても失敗することに注意してください。配列次元を揃えるために仮に shape (1,3) となたとしても、`a` の shape (3,4) と一致させることができずに ValueError を生じるからです。

この節の始めに示したスカラー（0 次元配列）`s` と 1 次元 NumPy 配列 `v` との積計算では、shape () を持つスカラーと 1 次元以上の任意の NumPy 配列 `A` との演算においては、規則 (0) が適用され次元を合わせるために shape 左軸に shape 値 1 を持つ軸が追加されて shape は (1,) となり次元が一致するまで繰り返し適用されて shape (1,...,1) となってから、各軸について比較対象の配列と shape 値が一致するまで引き伸ばされて同じ配列形状同士での演算が実行されるというブロードキャスト経過として計算されたと考えればよいのです。

すなわち、一般の n 次元配列 A とスカラー s との掛け算では次のようなブロードキャスト規則の適用となります。次元を一致させた配列において、配列形状を一致させるために形状を shape 値の右端から調べていきます。

	初期形状	次元拡大	...	次元一致	引き延ばし	...	引き延ばし
A	$k_1 \times \cdots \times k_n$	$k_1 \times \cdots \times k_n$	...	$k_1 \times \cdots \times k_n$	$k_1 \times \cdots \times k_n$	$k_1 \times \cdots \times k_n$	$k_1 \times \cdots \times k_n$
s	none	1	...	$1 \times \cdots \times 1 \times 1$	$1 \times \cdots \times 1 \times k_n$	...	$k_1 \times \cdots \times k_n$

1.4.2 形状を合わせてブロードキャストする

計算対象の配列同士がブロードキャスト可能でない場合には、ブロードキャスト可能なように配列形状を変更して計算できるような工夫を考えることができます。たとえば、shape(4,) を持つ 1 次元配列 `x` と shape (3,) を持つ 1 次元配列 `y` の積を考えてみます（数学では 2 つのベクトルの外積として計算できます[7)]）。しかし、このまま `x * y` としては先のブロードキャスト規則が適用できずエラーとなります。配列の shape の左端に新たに shape 値 1 として新たな軸を追加して引き延ばしたとしても両者の配列形状を一致させることができないからです。しかしながら、次のように `np.newaxis`（節 1.3.6 参照）を使って `x` の右端に新しく軸を追加して shape (4,1) と配列形状を変更すると積計算が可能になります（あるいは `reshape` によってブロードキャスティング可能な配列形状 (4,1) に変換しても構いません）。

```
import numpy as np

x = np.array([1,2,3,4])
y = np.array([5,6,7])
```

7) ここで考えている 1 次元 NumPy 配列（ベクトル）`a`,`b` との外積は `numpy.outer(a,b)` で計算できます。

```
x[:,np.newaxis] * y
```

```
array([[ 5,  6,  7],
       [10, 12, 14],
       [15, 18, 21],
       [20, 24, 28]])
```

この計算でのブロードキャスティングの経緯は次のようになっています。

	初期形状	次元一致	引き延ばし
x′	4×1	4×1	4×3
y	3	1×3	4×3

`x′ = x[:,np.newaxis]` と `y` との積において、まず次元の小さな `y` の形状が規則 (0) により shape (1,3) として `x′` と次元を一致させ、次いでそれぞれの配列形状を `x′` では第 1 軸方向に、`y` では第 0 軸方向に引き延ばすことによって同一配列形状を得た後、要素毎の掛け算が実施されます。

1.4.3 ブロードキャストの応用

2 つ以上の引数を持つ関数値の計算をブロードキャスト機構を使って効率的に計算することができます。たとえば、平面上の実数値関数 $f : \mathbb{R}^2 \to \mathbb{R}$ を使って矩形格子点領域 $\{(x_j, y_k)\} \,|\, x_j \in \text{xrange}, y_k \in \text{yrange}$ における各点 (x_j, y_k) に対応する関数値 $f(x_j, y_k)$ を計算することを考えてみます。NumPy では、矩形上の関数値全体 $\{f(x_j, y_k)\}$ を x-成分リスト `xrange` と y-成分リスト `yrange` から 2 重の for 文を使って要素を取り出すようなコードを書かずとも、一気の計算することができます。

ここでは具体的に、図 1.4 のような x, y-平面上に高さ $V(x, y)$ を持つ関数 V（ポテンシャル関数）

$$V(x, y) = \frac{1}{2}(x^2 + y^2) + x^2 y - \frac{1}{3}y^3 \tag{1.1}$$

について計算してみましょう。

x-成分リスト `xrange` と y-成分リスト `yrange` をそれぞれ shape (N,) および shape (M,) を持つ第 0 軸だけからなる 1 次元 NumPy 配列として用意しておき、`reyrange = yrange.reshape(M,1)`（または `yrange[:, np.newaxis]`）で右端に新しく軸を追加して、関数値 potential(xrange, reyrange) を計算すると各矩形要素に対応する関数値からなる shape (N,M) を有する 2 次元配列 `zrange` が計算できます。次のコード 1.4-1 は、関数 potential に 2 つの引数 `xrange,reyrange` を渡すとき、節 1.4.2 で説明したブロードキャスティングが実施されて potential に shape (N,M) の配列を渡したことにとなって、その関

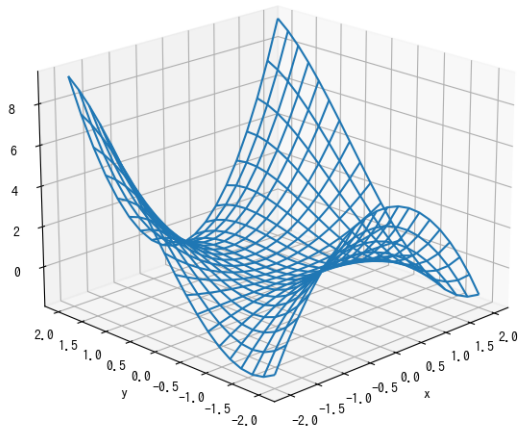


図 1.4　関数 $V(x,y)$（式 (1.1)）の領域 $[-2,2]\times[-2,2]$ での様子。水を注ぎ込むと原点付近でわずかに貯めることができるが、やがてそれ以上は三方からこぼれ無限遠に流れ落ちてしまう（図 7.1 参照）。

数値を要素毎に計算します。このような NumPy 計算ができる関数をユニバーサル関数と呼び、改めて節 1.5 で詳しく説明します。

コード 1.4-1　imshow_potential-1

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

def potential(x: float, y: float) -> float:
    return((x ** 2 + y ** 2 + 2 * x ** 2 * y -2/3 * y ** 3) / 2)

xmin, xmax = -3, 3
ymin, ymax = -2, 2
xrange = np.linspace(xmin, xmax, num = 30) # shape (30,)
yrange = np.linspace(ymin, ymax, num = 20) # shape (20,)
reyrange = yrange.reshape(20, 1) # 軸の追加 yrange[:, np.newaxis]  (num,1)
zmesh = potential(xrange, reyrange)
```

さらに次のコード 1.4-2 を実行して、2 次元配列 `zrange` の値の大小を Matplotlib の `pyplot.imshow` を使ってグレースケールの濃淡で描いてみます。小さな値（ポテンシャル値が低いと）白に、大きな値（高い値）では黒く描かれています。

コード 1.4-2　imshow_potential-2

```
fig, ax = plt.subplots()
ax.imshow(zmesh, origin='lower', extent=[xmin,xmax, ymin,ymax], cmap='gray')
```

```
ax.set(xlabel='x', ylabel='y')
#plt.show()
```

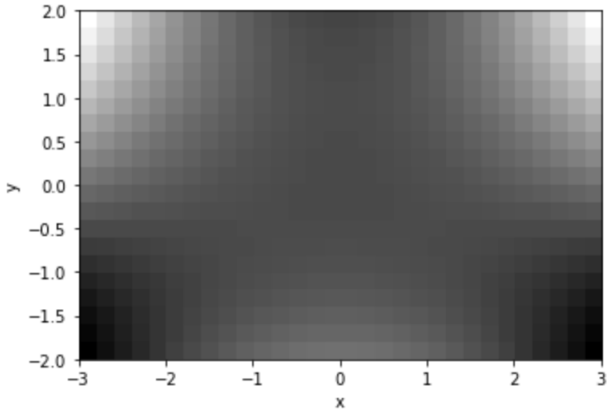


図 1.5　式 (1.1) で定義される平面上の高さを要素とする 2 次元 NumPy 配列値の大小を濃淡をつけてプロット。小さな値（高さが低いと）白に、大きな値（高い高さ）では黒くなる。

配列プロット `pyplot.imshow` は、描画原点を左下にし数学での座標位置の習慣に合わせるためにオプション `origin='lower'`、およびその描画範囲を x と y-方向でそれぞれ区間端までの目盛りを付けるように `extent=[xmin,xmax,ymin,ymax]` を指定しています。

なお、図 1.4 は Matplotlib の 3 次元プロット機能を使ってコード**??**で描いています。ここでが、節 2.2.3 で紹介する、1 次元次元 NumPy`xrange` と `yrange` を渡して `numpy.meshgrid` が返す同じ shape (N,M) を持つ x, y-平面上の矩形格子点配列の x-成分配列 `xg` と y-成分配列 `yg` をつかって矩形療育での関数値全体 `zg = potential(xg, yg)` を計算して `plot_wireframe` をつかって 3 次元ワイヤーフレームを描きました。Colaboratory（または Jupyter）では Matplotlib の 3D 描画は射影されたままですが、コマンドライン IPython をつかった場合にはプロット結果は別ウィンドウに表示され、マウスで回転させながらさまざまな視点から眺めることができます。

コード 1.4-3　potential_well

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def potential(x, y: float) -> float:
    return((x ** 2 + y ** 2 + 2 * x ** 2 * y -2/3 * y ** 3) / 2)
```

```
xmin, xmax = -2, 2
ymin, ymax = -2, 2
xrange = np.linspace(xmin, xmax, num = 20) # shape (20,)
yrange = np.linspace(ymin, ymax, num = 20) # shape (20,)
xg, yg = np.meshgrid(xrange, yrange) # グリッド格子配列
zg = potential(xg, yg)

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_wireframe(xg, yg, zg)
ax.set(xlabel = 'x', ylabel = 'y')
#fig.show()
```

1.4.4 配列値の条件処理 numpy.where

NumPy 関数 numpy.where(condition, Texp, Fexp) は、NumPy 配列に関する条件 condition の真偽に応じて True のとき値 Texp を、False のとき値 Fexp を返します。このとき、Texp, Fexp と condition はブロードキャスティング可能でなければなりません。また、条件 condition だけが指定された場合は、要素が非ゼロとなる配列要素のインデックスを返します。

```
a = np.arange(3*4).reshape(3,4)
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
np.where(a % 2, a, -1) # 偶数ならそのまま、奇数なら-1
```

```
array([[-1,  1, -1,  3],
       [-1,  5, -1,  7],
       [-1,  9, -1, 11]])
```

オプション Texp, Fexp が省略されたときには、条件を満たす要素のインデックス（場所）が各軸ごとに配列として返ります。条件を満たす要素は、各軸からそれぞれ取り出して得られた数の並びで指定された配列要素です。

```
np.where(a % 2) # 奇数 a % 2 == 1 である要素のインデックスの軸ごとの並び
```

```
(array([0, 0, 1, 1, 2, 2]), array([1, 3, 1, 3, 1, 3]))
```

たとえば、a[0,1],a[0,3],a[1,1],...,a[2,1],a[2,3] の 6 つの要素が条件 a % 2（2 で割った余りが 1）のある要素の行と列の場所の並びを返しています。

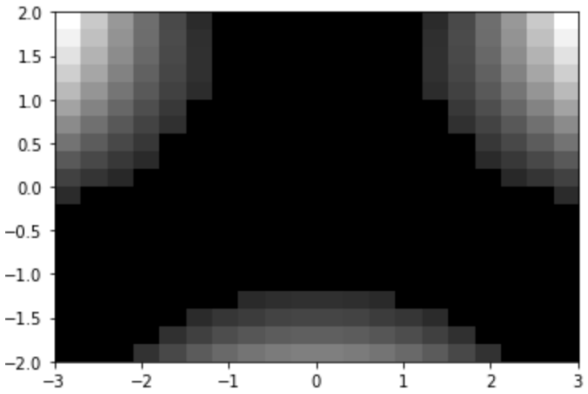


図 1.6　コード 1.4-1（図 1.5）で得られる 2 次元配列 zrange を値 1.5 以下なら値 0 としそれ以外は同じ値の配列とする条件処理 np.where(zrange < 1.5, 0, zrange) を施してプロット。1.5 よりも小さい要素を持つ領域が黒で描かれる。

コード 1.4-1 で得た配列 zrange（図 1.5）に対して numpy.where を conditioned = np.where(zrange < 1.5, 0, zrange) のように使って、配列 zrange を値 1.5 以下なら値 0 としそれ以外は同じ値の配列となるように変更して濃淡プロットプロットすると図 1.6 のようになります。1.5 よりも小さい要素を 0 とした領域はグレイスケールでは黒で描かれます。

コード 1.4-4　potential_well_using_where

```
zrange = potential(xg, yg)
conditioned = np.where(zrange < 1.5, 0, zrange) # 配列の条件処理

fig, ax = plt.subplots()
ax.imshow(conditioned, origin='lower', extent=[xmin,xmax, ymin,ymax], cmap='
    gray')
#fig.show()
```

1.5　ユニバーサル関数

NumPy 計算では、複数の引数を持つ関数に NumPy 配列を渡したとき、それらの入力配列の組をブロードキャスティングして得られる配列形状で要素毎に関数値を計算することができました。このように振る舞う関数を**ユニバーサル関数** (ufunc: universal function) といい、配列形状の要素毎に計算する様子を計算の**ベクトル化計算**と呼びます。

しかしながら、ユニバーサル関数はどんな引数配列の組に対してもうまく計算できるわけではありません。まず、与えられた配列引数の組がブロードキャスティング規則に合致するかをがチェックされ（節 1.4.1 ブロードキャスト規則参照)、要素毎に関数値が計算される配列形状が決定されねばなりません。NumPy 配列同士の加減乗除が要素ごとに計算できるために配列形状の一致が必要だったように、ユニバーサル関数に与えられる引数配列同士でブロードキャスチング可能という配列形状整合性が必要です。ベクトル化計算のごく簡単な例として、節 1.1.2 では 1 変数関数 $f : \mathbb{R} \to \mathbb{R}$ の引数に shape (N,) の 1 次元 NumPy 配列を与えて (N,) の 1 次元配列の関数値（ベクトル）$\{f(x_i)\}$ を一度に計算すること、節 1.4.3 では 2 変数関数 $f : \mathbb{R}^2 \to \mathbb{R}$ に shape (N,) と (M,1) を持つ配列を与えて Shape (N,M) の 2 次元配列の関数値 $\{f(x_i, y_j)\}$ を一度に計算することを紹介しました。

ただし、ベクトル化計算を実行するためには、関数に与える引数の整合性だけでなく、関数自身にも依存します。たとえば次のような条件式を含むような関数 funcif ではベクトル化計算に失敗します。

```
xlist = np.arange(-5,5)

def funcif(x: float) -> float:
    if x ** 2 > 10:
        return(-x ** 2)
    else:
        return(x ** 2)

funcif(xlist)
```

```
ValueError: The truth value of an array with more than one element is
    ambiguous. Use a.any() or a.all()
```

関数 `funcif` は引数に NumPy 配列を渡したときに要素毎に条件が吟味されて計算できるよう想定されていないために ValueError となりました。

1.5.1 関数のユニバーサル化 `numpy.frompyfunc`

ユニバーサル化関数 `numpy.frompyfunc(func, nin, nout)` は、Python 関数 `func` とその引数の数 `nin`、および出力されるオブジェクト数 `nout` を与えてユニバーサル関数を生成します。生成されたユニバーサル関数は Python Object からなる配列を返します。

先に if 文を使って定義した関数 `funcif` が浮動小数の引数 1 つに対して返値に 1 つの浮動小数を持つことから、`numpy.frompyfunc` でユニバーサル関数化される `ufuncif` は NumPy 配列 `xlist = np.arange(-5,5)` に対して正しく動作します。

```
ufuncif = np.frompyfunc(funcif,1,1)

ufuncif(xlist)
```

```
array([-25, -16, 9, 4, 1, 0, 1, 4, 9, -16], dtype=object)
```

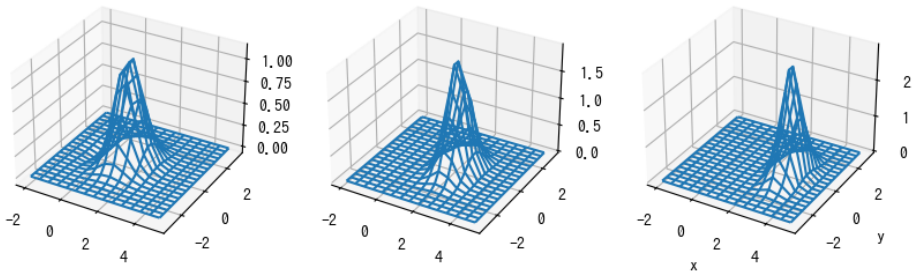


図 1.7　関数 $f_a(x,y) = xe^{-a(x-a)^2-y^2}$ のパラメータ $a = 1,2,3$ 変化。パラメータ a_k ごとに格子点 $\{x_i, y_i\}$ 上の関数値 $\{f_{a_k}(x_i,y_j)\}$ を計算してプロットする方法と、ユニバーサル関数化して値 `uf` $= \{f_{a_k}(x_i,y_j)\}$ をパラメータ変化ごと一度に計算して、パラメータに関してスライス `uf[k]` しながらプロットする方法がある。

平面上の関数 $f_a(x,y)$

$$f_a(x,y) = xe^{-a(x-a)^2-y^2}$$

をパラメータ a による変化を、たとえば `arange[k]` $= 1,2,3$ の 3 つで観察してみましょう。関数 `f(x,y,a)` を定義してプログラム 1.4-4 に倣って、$a =$ `arange[k]` ごとにワイヤーフレームをプロットすると図 1.7 を得ることができます。shape (xn,) および (yn,) の 1 次元配列 `xpoint, ypoint` から `numpy.meshgrid` を使って `xnum` × `ynum` の格子点 `xg,yg` を求め、各格子点上の高さ `zrange = f(xg, yg, arange[k])` を繰り返し計算します。

```
xpoint = np.linspace(xmin,xmax, num=xn) # shape (xn,)
ypoint = np.linspace(ymin,ymax, num=yn) # shape (yn,)
xg, yg = np.meshgrid(xpoint, ypoint)
an = 3
arange = np.linspace(1,3, num=an) # shape (an,)

fig = plt.figure(figsize = (10, 3))
ax: List[Any] = [x for x in range(an+1)]
for k in range(an):
```

```
    zrange = f(xg, yg, arange[k])
    ax[k] = fig.add_subplot(1, an, k+1, projection = '3d')
    ax[k].plot_wireframe(xg, yg, zrange)
ax[k].set(xlabel = 'x', ylabel = 'y')
plt.show()
```

引数候補	shape	broadcast 可能な shape	関数内の引数 shape
xpoint	(xn,)	(xn,) または (xn,1)	(1,1,xn) または (1,xn,1)
xpoint	(yn,)	(yn,1) または (yn,)	(1,yn,1) または (1,1,yn)
arange	(an,)	(an,1,1)	(an,1,1)

表 1.1　ユニバーサル関数 `uf` に渡すための配列 shape の調整と関数内で使われる引数配列の shape。ユニバーサル関数に渡された配列はすべてブロードキャストされて同じ次元にされる。規則 B1 を使ってそれよりも少ない次元を持つ配列はその shape の先頭に 1 を追加して次元が追加され、たとえば `(xn,)` は `(1,1,xn)` に、`(yn,1)` は `(1,yn,1)` に形状が変わる。規則 B2 からユニバーサル関数の出力 shape はいずれの場合でも `(an, yn, xn)` になる。

関数 `f` に以下のように正しく引数配列を渡すと‘強引に’求めるベクトル計算できます（Python では型を強要しないために正しく実行しますが mypy の型チェックではエラーとなります)。ここでは関数 `f` を次のように 3 つの引数として浮動小数 x, y, a を与え 1 つの NumPy 浮動小数を返す関数と型アノテート宣言しています。

```
def f(x: float, y: float, a: float) -> np.float_:
    return(x * np.exp(-a*(x - a)**2 - y**2))
```

mypy 型チェックを成功させるためには、配列引数を取れるように `numpy.frompyfunc` を使って `f` をユニバーサル化して `uf` とします。型アノテーションを厳格に履行するとこのような手間がかかりますが、コード上の副作用に気づくことができるので型アノテーションが有用です。

```
uf = np.frompyfunc(f, 3, 1)
```

念のために触れておくと、次のように型アノテーションしておくと、`numpy.frompyfunc` でユニバーサル化して `uf` とする必要はありません。

```
    return(x * np.exp(-a*(x - a)**2 - y**2))
```

それでもなお、`uf(xpoint, ypoint, arange)`、`uf(xpoint, ypoint, arange[k])` などとすると、異なる shape の引数配列のブロードキャストに失敗して実行エラー‘`ValueError` になります。引数配列の shape を手当しない限り、このままでは `uf(xpoint, ypoint[j], arange[k])`（shape (xn,) の配列）, `uf(xpoint[i],`

ypoint, arange[k])（shape (yn,) の配列）、uf(xpoint[i], ypoint[k], arange)（shape (anu,0) の配列）しか計算できません。

問題は配列の組 xpoint,ypoint,arange がユニバーサル関数 uf においてブロードキャスト可能でないことにあります。ユニバーサル関数への配列引数はブロードキャスト規則 B1∼B4 が適用され関数内で同じ次元を持つ配列となるように与えねばなりません。小さな次元を持つ配列は shape の先頭に 1 を追加して次元拡大されるという制約のもとで、計算目的に合致するように引数配列を調整します。

現在の目的は、shape (xn,yn）を持つ x, y-格子上にパラメータ変化によって an 個の高さを有する shape(xn,yn,an) の配列結果 uz を計算し、各パラメータ a_k での格子上の高さを uz[k] でスライスして取り出すことです。

表 1.1 に示したように、ユニバーサル関数 $\mathtt{uf}(x, y, a)$ を使って目的の計算を行うためにブロードキャスト可能な引数配列の shape の組 xshape, yshape, ashape は

$$\mathtt{xshape, yshape, ashape} = \mathtt{(xn,),(yn,1),(an,1,1)} \quad \text{または}$$
$$\mathtt{(xn,1),(yn,),(an,1,1)}$$

のどちらかになります。ユニバーサル関数に渡された配列はすべてブロードキャストされて同じ次元になります。規則 B1 を使ってそれよりも少ない次元を持つ配列はその shape の先頭に 1 を追加して次元が追加され、たとえば (xn,) は (1,1,xn) に、(yn,1) は (1,yn,1) に形状が変わります。規則 B2 からユニバーサル関数の出力 shape は目的の (an, yn, xn) になります。

プログラム 1.5-1 ufunction_1.py は、関数 f(x,y,a) をユニバーサル関数 uf にした上で、x, y-格子上にパラメータ変化ごとの高さを一度に計算しておき、パラメータに関してスライスしながら図 1.7 をプロットします。関数 uf に渡す配列の shape を確認してください。

コード 1.5-1　ufunction_1.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from typing import List, Any, Union

def f(x: float, y: float, a: float) -> np.float_:
    return(x * np.exp(-a*(x - a)**2 - y**2))

uf = np.frompyfunc(f, 3, 1) # ユニバーサル化
```

```
xmin, xmax = -2, 5
ymin, ymax = -3, 3
xn, yn = 15, 20
xpoint = np.linspace(xmin,xmax, num=xn) # shape (xn,)
ypoint = np.linspace(ymin,ymax, num=yn) # shape (yn,)
xg, yg = np.meshgrid(xpoint, ypoint)
an = 3
arange = np.linspace(1,3, num=an) # shape(an,)
# 格子点上のパラメータごとの高さを一度に計算
uz: np.ndarray = uf(xpoint, ypoint.reshape(yn,1), arange.reshape(an,1,1))
uz = np.array(uz, dtype='float') # オブジェクト配列を浮動小数配列で上書き

fig = plt.figure(figsize = (10, 3))
ax: List[Any] = [x for x in range(an+1)]
for k in range(an):
    ax[k] = fig.add_subplot(1, an, k+1, projection = '3d')
    ax[k].plot_wireframe(xg, yg, uz[k]) # パラメータ次元でスライスした高さで描
        く
ax[k].set(xlabel = 'x', ylabel = 'y')
plt.show()
```

第 2 章　行列計算

NumPy を利用する行列計算の取り扱いを紹介します。n-次元 NumPy 配列（クラス array）は行列だけでなくテンソル計算も同じ枠組で行うことができます（1 階テンソルがベクトル、2 階テンソルが行列という具合です)。計算に関与する NumPy 配列の shape 同士が正しい整合関係にあるように留意すれば、自由に NumPy 配列同士の計算を行うことができます。

行列の計算、たとえば連立方程式や固有値と固有ベクトルの計算は科学計算における中心的テーマの一つです。

NumPy では行列計算として開発洗練されてきた多くの手法を使うことができます。そのわずかだけを使っても有用な計算を行うことが可能になります。Numpy を使い始めたときには是非 NumPy user guide https://numpy.org/doc/stable/user/ を眺めてみてください。また、計算目的や計算規模に応じて NumPy と記号計算パッケージ SymPy とを併用するとさらに強力な計算が可能になります。

2.1　行列積と内積

2.1.1　行列の積

m 行 n 列行列（$m \times n$ 行列）A の要素を ${A^i}_j (i \in \{0, \ldots, m-1\}, j \in \{0, \ldots, n-1\})$ とします。行列 A の要素をあえて ${A^\mu}_\nu$ を記したのは、A には要素が並ぶ軸が 2 つあるここと を明示するためです。NumPy では m 行 n 列行列 A は shape (m,n) を持ちます。また、A の行と列の要素を入れ替えた $n \times m$ 列の行列を転置行列 A^T と記し、NumPy では shape (n,m) を持ちます。

$$\boldsymbol{A} = {A^i}_j = \begin{bmatrix} {A^0}_0 & {A^0}_1 & \ldots & {A^0}_{n-1} \\ {A^1}_0 & {A^1}_1 & \ldots & {A^1}_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ {A^{m-1}}_0 & {A^{m-1}}_1 & \ldots & {A^{m-1}}_{n-1} \end{bmatrix}, \quad \boldsymbol{A}^T = ({A^j}_i) = \begin{bmatrix} {A^0}_0 & {A^1}_0 & \ldots & {A^{m-1}}_0 \\ {A^0}_1 & {A^1}_1 & \ldots & {A^{m-1}}_1 \\ \vdots & \vdots & \ddots & \vdots \\ {A^0}_{n-1} & {A^1}_{n-1} & \ldots & {A^{m-1}}_{n-1} \end{bmatrix}$$

とくに、$1 \times n$ 行列を**行ベクトル** $\boldsymbol{v} = (v_i)$、$n \times 1$ 行列を**列ベクトル**といい行ベクトルの

転置 $\boldsymbol{v}^T = (v^i)$ で表されます。

$$(v_i) = \begin{bmatrix} v^0 & v^1 & \dots v^{n-1} \end{bmatrix}, \quad (v^i) = \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{bmatrix}, \quad ((\boldsymbol{v}^i)^T)^T = v^i.$$

ただし、NumPy 的では行ベクトルと列ベクトルとを区別して利用することはなく、shape (n,) を持つ 1 次元 NumPy 配列として賄ういます。一方、SymPy の行列を表す Matrix クラスでは行ベクトルは (1,n)、列ベクトルでは (n,1) と区別されます（節 2.4.2.1 参照)。

$m \times n$ 行列 $A = (A^i{}_j$ と $n \times \ell$ 行列 $B = B^p{}_q$ との行列積 $AB \equiv A \times B$ は $m \times \ell$ 行列となり、その行列要素 $(AB)^i{}_k$ は次のように A の第 1 軸と B の第 0 軸の添字 ρ に関する総和積によって定義されます。このような操作をテンソルの**縮約**ということがります（節 2.1.3 参照)。

$$(AB)^i{}_k \equiv \sum_{j=1}^{n} A^i{}_j B^j{}_k, \quad 0 \leqq i \leqq m-1, 0 \leqq k \leqq \ell - 1.$$

行列積 BA が定義できる場合でも、一般に $AB \neq BA$ であって行列は積演算に関して可換ではありません。

たとえば次のように、2×3 行列 A と 3×2 行列 B

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 2 \\ 1 & 0 \\ 5 & 6 \end{bmatrix}$$

および、列ベクトル $\boldsymbol{u}$ と行ベクトル $\boldsymbol{v}$ を考えます。

$$\boldsymbol{u} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}, \quad \boldsymbol{v} = \begin{bmatrix} 2 & i \end{bmatrix}$$

このとき、次の行列積が計算できます。行列同士や行列とベクトルとの縮約操作と得られる配列の shape を確認してください。

$$AB = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 1 & 0 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 16 & 22 \\ 34 & 49 \end{bmatrix}, \quad BA = \begin{bmatrix} 9 & 12 & 15 \\ 4 & 5 & 6 \\ 29 & 40 & 51 \end{bmatrix},$$

$$A\boldsymbol{u} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 10 \\ 28 \end{bmatrix}, \quad \boldsymbol{v}A = \begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 2+4i & 4+5i & 6+6i \end{bmatrix}.$$

行数と列数が等しい行列を**正方行列**と呼び、その行数（= 列数）を行列の次数といいます。行列 A の要素 ${A^k}_k$ を対角要素と呼びます。対角要素以外の行列要素がすべて 0 である行列を**対角行列** (diagonal matrix) といいます。特に対角成分がすべて 1 の対角行列を**単位行列**といい I で記します。任意の正方行列 A に対して $IA = AI = A$ となります。

正方行列では**行列式** (determinant) $\det A$ が定義されます。正方行列 $\boldsymbol{A}$ との行列積が**単位行列** (identity matrix) になる行列を A の逆行列といい A^{-1} と記します。逆行列を有する行列を**正則行列** (regular matrix) といいう、その必要十分条件は $\det A \neq 0$ であることが知られています。

大規模行列の逆行列の計算は計算量的に困難です。しかしながら、連立方程式の解法を含む実用上の重要な問題において逆行列や関連する行列を求めることが必要になる場合が多く、計算機科学における重要な研究テーマの 1 つとなっています。

2.1.2 スカラー積

行ベクトル $\boldsymbol{v}$ と列ベクトル $\boldsymbol{u}$ との添字の総和積（縮約）としてスカラーを返す操作を**スカラー積**といい（scalar product）

$$v_k u^k \equiv \sum_k v_k v^k$$

で定義します。スカラー積を $\boldsymbol{v} \cdot \boldsymbol{u}$ と記すことがあるため**ドット積**ということもあります[1)]。また、スカラー積をしばしは**内積**ということがあります。計量ベクトル空間を V としたときには、任意のベクトル $\boldsymbol{x} \in V$ について正値性 $(\boldsymbol{x}, \boldsymbol{x}) \geqq 0$ が必要なため、複素ベクトルにおいて内積 $(\boldsymbol{x}, \boldsymbol{x})$ を

$$(\boldsymbol{x}^{T*}, \boldsymbol{x}) \equiv \sum_k x_k^* x^k$$

のように複素共役 * をとった総和積で定義します。

$$(\boldsymbol{v}, \boldsymbol{v}) = 2 \times 2 + (-i) \times i = 5$$

1） NumPy には同様な関数が複数用意されており、目的に応じて使い分けしてください（節 2.2.1.2）。

2.1.3 テンソル

要素が各要素が ${T^{p_0,\ldots,p_{k-1}}}_{q_0,\ldots,q_{\ell-1}}$ のように $n = k + \ell$ 個の添字で指定される多次元配列 T を n-階**テンソル** (tensor) と呼びます（次元を 2 次元配列とか n 次元ベクトルなどというように使って混乱することがあるので、ここでは階数と呼びます)。配列次元をその shape 長でると考えると混乱しません。すると、テンソル T の shape は n-本の軸に沿って $(s_0, \ldots, s_{n-1})$ となり、配列 T に $prod_j s_j$ 個の要素が並んでいます。スカラーは 0 階テンソル（shape ()）、ベクトルは 1 階テンソル（shape (n,)）、行列は 2 階テンソル（shape (m,n)）などと考えることができます。

添字が多くあるテンソル計算を手計算で確かめることは大変煩雑ですが、NumPy の多次元配列の計算をより良く理解することができます。配列の shape に着目した見通しのよい計算ができるようになります。

2.1.3.1 テンソル積

shape $(p_0, \ldots, p_{k-1})$ を持つテンソル A（軸は k 本）と shape $(q_0, \ldots, q_{\ell-1})$ を持つテンソル B（軸は ℓ 本）のテンソル積 (tensor product) $A \otimes B$ は、$(k + \ell)$ 本の軸に沿って $\prod_i p_i \prod_j q_j$ 個の要素を shape $(p_0, \ldots, p_{k-1}, q_0, \ldots, q_{\ell-1})$ を持つような配置したテンソルです。たとえば、1 次元配列 x, y や 2 次元配列のテンソル積は次のようになります

$$(x_0, x_1) \otimes (y_0, y_1, y_2) = \begin{bmatrix} x_0y_0 & x_0y_1 & x_0y_2 \\ x_1y_0 & x_1y_1 & x_1y_2 \end{bmatrix}, \qquad \text{shape (2,3)}$$

$$(y_0, y_1, y_2) \otimes (x_0, x_1) = \begin{bmatrix} x_0y_0 & x_1y_0 \\ x_0y_1 & x_1y_1 \\ x_0y_2 & x_1y_2 \end{bmatrix}, \qquad \text{shape (3,2)}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes (x_0, x_1) = \left[, \begin{bmatrix} ax_0 & ax_1 \\ bx_0 & bx_1 \end{bmatrix} \quad \begin{bmatrix} cx_0 & cx_1 \\ dx_0 & dx_1 \end{bmatrix} \right], \qquad \text{shape (2,2,2)}$$

2.1.3.2 テンソルの縮約

shape $(s_0, s_1, \ldots, s_{n-1})$ を持つ n 次元テンソル $T = {T^{\mu_0,\ldots,\mu_{k-1}}}_{\nu_0,\ldots,\nu_{\ell-1}}$ において、s 本の第 $(j_1, \ldots, j_s)$-軸に関する総和積

$$\sum_{j_1,\ldots,j_s} {T^{p_0,\ldots,p_{k-1}}}_{q_0,\ldots,q_{\ell-1}}$$

を取ることがあります。この操作を**縮約** (summation convention) といいいます。縮約した添字の数だけテンソルの階数が下がります。

数学や物理学では表記上の煩雑さを避けるために、1 つのテンソル表記において同じ添字（擬標: dummy index といいます）が現れるとその擬標について縮約するという習慣を**アインシュタイン記法**と呼んでいます[2)]。

テンソル積 $A \otimes B$ において、k-階テンソル $A^{\mu_0,\ldots,\mu_{k-1}}$ が shape $(p_0,\ldots,p_{k-3},d_0,d_1)$、$\ell$-階テンソル $B_{\nu_0,\ldots,\nu_{\ell-1}}$ が shape $(d_0,d_1,q_2,\ldots,q_{\ell-1})$ を持つとします。このとき、A の最後の 1 軸と B の先頭の 1 軸の対応する shape 値が等しい、さらに A の最後の 2 軸と B の先頭の 2 軸の対応する shape 値が等しいために、添字 d_2 あるいは添字 d_2,d_1 について総和積を取って縮約することができます。たとえば 2 組 d_1,d_2 について縮約するとテンソル C が次のように与えられます。

$$C^{i_0,\ldots,i_{m-3}}{}_{j_2\ldots,j_{n-1}} = \sum_{d_1,d_2} A^{i_0,\ldots,i_{m-3},d_1,d_2} B_{d_1,d_2,j_2\ldots,j_{n-1}}$$

このとき縮約に関わった軸数は A,B の双方で 4 本あり、縮約して得られた C はの階数は 4 つ減り $(k+\ell-4)$-階のテンソルになります。

2.2 NumPy を使う行列計算

NumPy は既に見てきたように多次元配列の計算をたいへん効率的に実行するう多数のパッケージを用意しています。配列計算に関わる僅かな例に限りますが、NumPy 計算の強力を詳しく見ていきます。

何度も繰り返すことになりますが（節 2.1.1）、NumPy ベクトルを使う計算では計算にあたって関与する配列 shape の整合性に留意位する限り、列ベクトルと行ベクトルを区別する必要はありません。

2.2.0.1 転置 `transpose`

`numpy.transpose(a)` あるいはメソッド `a.transpose()`、または略記した `a.T` は NumPy 配列の shape $(\mathrm{s}_0,\ldots,\mathrm{s}_{n-1})$ を反転して $(\mathrm{s}_{n-1},\ldots,\mathrm{s}_0)$ とします。この操作を転置と呼びます。ただし、1 次元配列 x については転置による配列形状の変化はありません（shape 値が 1 つなので反転しても同じです）。

```
In [1]: x = np.arange(6)

In [2]: x.shape == (x.T).shape
Out[2]: True
```

2) たとえば、Riemann の曲率テンソル $R^{\alpha}_{\beta\mu\nu}$ から縮約によって Ricci テンソル $R_{\mu\nu} = R^{\alpha}_{\mu\nu\alpha} (= \sum_{\alpha} R^{\alpha}_{\mu\nu\alpha})$ を定義するという具合です。

```
In [3]: y = np.ones((1, 2, 3))

In [4]: (y.T).shape
Out[4]: (3, 2, 1)
```

2.2.1 配列の積演算

配列積を含むドット積に関わる基本計算でも表 2.1 にその一部を掲載したように NumPy には多数の手段が用意されています。これらの計算方法には差異があります。利用する際には目的とする計算を正しく行うことができるか、リファレンスマニュアルで使い方とその利用例を確かめてください。

表 2.1　NumPy 配列同士の掛け算（ほんの一部）

`numpy.matmul(a, b)`	2 つの配列の行列積（2 項演算子 `@` も可）
`numpy.vdot(a, b)`	2 つのベクトルのドット積（内積）
`numpy.linalg.matrix_power(a, n)`	正方行列のべき乗
`numpy.outer(a, b)`	2 つのベクトルの外積
`numpy.kron(a, b)`	2 つの配列の Kronecker 積
`numpy.tensordot(a, b[, axes])`	指定して軸に沿ったテンソル積
`numpy.dot(a, b)`	2 つの配列のドット積
`numpy.inner(a, b)`	2 つの配列の複素共役なしの内積

2.2.1.1 配列積 `matmul` (`@`)

`numpy.matmul(a, b)` または 2 項演算子 `@` を使った `a @ b` は、NumPy 配列 `a` と `b` の shape が整合的であるときその行列積を NumPy 配列で返します。スカラー倍する計算はできません。

表 2.2 に、1 次元および 2 次元 NumPy 配列同士の行列計算が可能な shape の整合関係を示しました。1 次元 NumPy 配列同士では、その shape 値が一致するときにだけ対応する要素ごとの積の総和されて配列次元 0 のスカラー積が返ります。一方、2 次元 NumPy 配列に 1 次元 NumPy 配列との積においても、計算に関与する配列の shape を並べたときに ‘隣り合う shape 値’ が一致している場合に総和積を計算（縮約）して、1 次元または 2 次元配列が返ります。この様子は線形代数の行列積の規則と同じです。

```
In [1]: import numpy as np
```

表 2.2　a, b は 1 次元または 2 次元の NumPy 配列の a @ b が計算可能な shape 整合性。下線のある shape 値について総和積が実行される。

a の shape	b の shape	a @ b の shape	返り値の次元
$(\underline{\mathrm{n}},)$	$(\underline{\mathrm{n}},)$	$()$	0 次元（スカラー）
$(\mathrm{n},\underline{\mathrm{m}})$	$(\underline{\mathrm{m}},)$	$(\mathrm{n},)$	1 次元
$(\underline{\mathrm{n}},)$	$(\underline{\mathrm{n}},\mathrm{m})$	$(\mathrm{m},)$	1 次元
$(\mathrm{n},\underline{\mathrm{k}})$	$(\underline{\mathrm{k}},\mathrm{m})$	(n,m)	2 次元
$(\mathrm{n},\underline{\mathrm{m}})$	$(\underline{\mathrm{m}},1)$	$(\mathrm{n},1)$	2 次元
$(1,\underline{\mathrm{n}})$	$(\underline{\mathrm{n}},\mathrm{m})$	$(1,\mathrm{m})$	2 次元

```
In [2]: x = np.array([3,2,1]) # shape (3,)

In [3]: x @ x # shape ()
Out[3]: 14

In [4]: a = np.arange(6).reshape(2,3) # shape (2,3)

In [5]: a @ x # shape (2,)
Out[5]: array([ 4, 22])

In [6]: y = np.array([2,1]) # shape (2,)

In [7]: y @ a # shape (3,)
Out[7]: array([3, 6, 9])
```

NumPy 計算で線形代数的なベクトルとして扱うときは 1 次元 NumPy 配列を使えばよく、列ベクトルや行ベクトルを区別する必要はありません[3)]。したがって、以下のように 1 次元 NumPy 配列を線形代数的に ‘列ベクトル’ として (n,1) や ‘行ベクトル’ として (1,n) と reshape してしまうと（2 次元配列となり）、2 次元配列とで積を取った返り値は 2 次元 NumPy 配列となり、線形代数的な理解に添えなくなってしまいます（節 1.3.2 参照）。

```
In [8]: x1 = x.reshape(3,1) #

In [9]: a @ x1 # shape (2,1)
Out[9]:
array([[ 4],
       [22]])

In [10]: y1 = y.reshape(1,2)
```

3)　ただし、複素ベクトルでのスカラー積の計算には複素共役を有無についての考慮は必要です。

```
In [11]: y1 @ a # shape (1,3)
Out[11]: array([[3, 6, 9]])

In [12]: (a @ x1).flatten()
Out[12]: array([ 4, 22])

In [13]: (y1 @ a).flatten()
Out[13]: array([3, 6, 9])
```

最後の 2 つでは、メソッド `ndarray.flatten()` をつかって多次元 NumPy 配列を 1 次元に平坦化しています。

配列積を計算する `a.matmul(b)` や `a @ b` は 2 次元配列や 1 次元配列以外の多次元配列で利用することができます。`a`, `b` が両方とも 2 より大きな配列次元を持ち、それぞれの shape が $(s_0, \ldots, s_i, s_j, s_k)$ および $(s_0, \ldots, s_i, s_k, s_m)$ であるとき最後の 2 つの shape 値 (s_j, s_k) と (s_k, s_m) を持つ行列の積み重ねとして扱われて、s_k についての総和積（縮約）計算が試みられて shape $(s_0, \ldots, s_i, s_j, s_m)$ の配列を計算します。たとえば、shape (5,4,3,2) を持つ配列 `c` と shape (5,4,2,6) を持つ配列 `d` との積 `c @ d` は成功して、shape 軸値 2 は縮約され、shape (5, 4, 3, 6) を持つ配列を得ます。

```
In [14]: c = np.arange(5*4*3*2).reshape(5,4,3,2)

In [15]: d = np.arange(5*4*2*6).reshape(5,4,2,6)

In [16]: (c @ d).shape
Out[16]: (5, 4, 3, 6)
```

2.2.1.2 内積とドット積 vdot dot np.inner

NumPy には NumPy 配列同士 `a`, `b` の内積や総和積（sum product: ドット積とい言い、対応する要素を掛けて総和する操作）を計算するための関数として、`numpy.vdot(a, b)`, `numpy.dot` （そのメソッド版 `a.dot(b)`）、 `np.inner` が用意されていますが、それぞれ働きが異なります（紛らわしいのですが `numpy.inner` は複素共役を取らない単純なドット積を計算するため、内積計算には注意が必要です）。

`numpy.vdot(a, b)` は、複素配列同士ときでも複素共役をとって非負なスカラー積を返します（内積計算に向いています）。多次元配列の場合には平坦化 (flatten) されて 1 次元配列として計算するため、マニュアルには `numpy.vdot` は 1 次元配列（ベクトル）同士の計算に使うべきとあります。

```
In [1]: import numpy as np

In [2]: x = np.array([1j,2+3j])

In [3]: y = np.array([4+5j, 6+8j])

In [4]: np.vdot(x, y)
Out[4]: (41-6j)

In [5]: np.vdot(y, x)
Out[5]: (41+6j)
```

線形代数的には行列積 `np.matmul`（演算子 `@`）とスカラー積 `np.vdot` を知っていればよいのですが、`np.dot` も見ておきましょう。`np.dot(a, b)` は 1 次元配列同士のときは複素共役を取らないスカラー積を計算します。2 次元配列同士や 2 次元配列と 1 次元配列の場合にはその行列積を与えます（ユーザマニュアルでは `np.matmul` や演算子 `@` を使うべきとしています)。

さらに、`np.dot(a,b)` は多次元配列 `a` と `b` でも使えます。`a` の shape $(s_0, \ldots, s_{m-2}, \underline{k})$ の右端 k と `b` の shape の $(t_0, \ldots, t_{n-3}, \underline{k}, t_{n-1})$ の右端から 2 番目の k についての総和積（縮約）が計算され shape $(s_0, \ldots, s_{m-2}, t_0, \ldots, t_{n-3}, t_{n-1})$ の配列を計算し ます。

たとえば、`a` の shape が (2,3,4)、`b` の shape が (3, 4, 2) のときには shape 値 4 を持つ軸で縮約され、shape (2, 3, 3, 2) をもつ配列を返します。

```
In [6]: x.dot(y)
Out[6]: (-17+38j)

In [7]: y.dot(x)
Out[7]: (-17+38j)

In [8]: a = np.arange(2*3*4).reshape(2,3,4)

In [9]: b = np.arange(2*3*4).reshape(3,4,2)

In [10]: (a.dot(b)).shape
Out[10]: (2, 3, 3, 2)
```

`np.inner(a, b)` は 1 次元 NumPy 配列を複素共役を取らず総和積を返します（複素内積空間としてのスラカー積計算には向いていません)。一般の多次元配列同士の場合にはそれぞれの配列の最後の軸で縮約れます。この結果は、節 2.2.2 で紹介するテンソル積 `np.tensordot` を次のように使っても計算できます。

```
np.inner(a, b) = np.tensordot(a, b, axes=(-1,-1))
```

次の例で確かめてみましょう。

```
In [11]: a = np.arange(24).reshape(2,3,4)

In [12]: b = np.arange(4)

In [13]: np.inner(a, b)
Out[13]:
array([[ 14,  38,  62],
       [ 86, 110, 134]])

In [14]: c = np.arange(8).reshape(2,4)

In [15]: np.inner(a, c)
Out[16]:
array([[[ 14,  38],
        [ 38, 126],
        [ 62, 214]],

       [[ 86, 302],
        [110, 390],
        [134, 478]]])

In [17]: np.inner(a, c).shape
Out[17]: (2, 3, 2)
```

2.2.2 NumPy のテンソル積 `tensordot`

`numpy.tensordot(a, b, axes=N)` は NumPy でテンソル `a` と `b` のテンソル積に関わる計算をします。引数 axes=0 と指定して縮約なしの（どの軸についても総和積をとらない）テンソル積

$$\mathrm{a} \otimes \mathrm{b} = \texttt{np.tensordot(a,b, axes=0)}$$

を計算します。

```
In [1]: import numpy as np

In [2]: a = np.array(('a','b','c'), dtype=object) # shape (3,)

In [3]: b=np.array([1,2,3,4]) # shape (4,)
```

```
In [4]: np.tensordot(a, b, 0) # shape (3,4)
Out[4]:
array([['a', 'aa', 'aaa', 'aaaa'],
       ['b', 'bb', 'bbb', 'bbbb'],
       ['c', 'cc', 'ccc', 'cccc']], dtype=object)

In [5]: c = np.array([[1,2,3], [4,5,6]]) # shape (2,3)

In [6]: np.tensordot(a, c, axes=0) # shape (3,2,3)
Out[6]:
array([[['a', 'aa', 'aaa'],
        ['aaaa', 'aaaaa', 'aaaaaa']],

       [['b', 'bb', 'bbb'],
        ['bbbb', 'bbbbb', 'bbbbbb']],

       [['c', 'cc', 'ccc'],
        ['cccc', 'ccccc', 'cccccc']]], dtype=object)
```

numpy.tensordot において、axes=1 と指定したときには a の最後の shape 軸 $(s_0, \ldots, s_{m-2}, k)$ と b の最初の軸 $(k, t_1, \ldots, s_{n-1})$ で総和積（縮約）をとって階数を 2 つ下げて shape $(s_0, \ldots, s_{m-2}, t_1, \ldots, s_{n-1})$ の NumPy 配列を返します。

また、axes=2 と指定してときは a の最後の 2 つの軸と b の最初の 2 つの軸に沿った総和積（縮約）をとって次数を 4 つ下げます。

たとえば、a の shape を (2,3)、b の shape を (2,3,2) ときのテンソル積 np.tensordot(a, b, axes=0) は shape (2, 3, 2, 3, 2) を持つ 5 階テンソルを返します。axes=2 とすると shape (2,) の 1 階元テンソル配列を返しますが、axes=1 では軸のマッチに失敗してエラー ValueError: shape-mismatch となります。

```
In [7]: a =np.arange(2*3).reshape(2,3)

In [8]: b = np.arange(2*3*2).reshape(2,3,2)

In [9]: np.tensordot(a, b, axes=0).shape
Out[9]: (2, 3, 2, 3, 2)

In [10]: np.tensordot(a, b, axes=2).shape
Out[10]: (2,)

In [11]: np.tensordot(a, b, axes=1) # ValueError: shape-mismatch
```

さらに、テンソルの縮約させる軸をそれぞれのテンソルごとに軸番号をリストで指定す

ることができます（2 つのリスト長は等しく指定します)。次の例では、shape (2,3,2) を持つテンソル b と shape (2,4,3,2) を持つ c とのテンソル積において、b の 0 番目と c の 0 番目および b の 1 番目と c の 2 番目とで縮約しています。

```
In [12]: c = np.arange(2*4*3*2).reshape(2,4,3,2)

In [13]: np.tensordot(b, c, axes=([0,1], [0,2])).shape
Out[13]: (2, 4, 2)
```

2.2.3 格子点を計算する numpy.meshgrid

縦と横に配置された点の集まりを**格子点**と言います。平面における格子点は $\{(x_i, y_j)\}$ で表されます。空間内の格子点での接ベクトルを図 5.5 のように図示してベクトル場の様子を観察するには格子点の座標を知る必要があります。プログラミングにおいて、格子点を利用する計算はしばしば登場します。Python では格子を**グリッド** (grid) といいます。

関数 numpy.meshgrid(X, Y) は、1 次元配列 $X = [x_1, x_2, \ldots, x_p]$, $Y = [y_1, \ldots, y_q]$ で定まる直積集合 $X \times Y$ において、その x 成分からなる x-格子行列と y-成分からなる y-格子行列の組を返します（どちらも shape (p,q) を持つ)。同様に、3 つの 1 次元配列 X, Y, Z に対する numpy.meshgrid(X, Y, Z) も直積集合 $X \times Y \times Z$ における x, y, z-格子行列（いずれも shape (p,q,r) を持つ）の 3 組を返します。

具体的に見てみましょう。次のように長さ 3 の xlist と長さ 2 ylist の 1 次元 NumPy 配列を用意します。

```
In [1]: import numpy as np

In [2]: xlist = np.array([5, 0, 7])

In [3]: ylist = np.array([1, 3])
```

np.meshgrid(xlist, ylist) は xlist と ylist から作られる格子点集合から、その x-成分、y-成分を取り出した 2 組の格子行列を返します。ここでは、$3 \times 2 = 6$ 個からなる格子点集合 $\{(x_i, y_j)\}, i = 0, 1, j = 0, 1, 2$ である $(5,1), (0,1), (7,1), (5,3), (0,3), (7,3)$ からその x-成分、y-成分を取り出した格子行列をそれぞれ gx, gy に格納しました。

```
In [4]: np.meshgrid(xlist, ylist)

In [5]: gx, gy = np.meshgrid(xlist, ylist)

In [6]: gx # shape (2, 3)
Out [6]:
```

```
array([[5, 0, 7],
       [5, 0, 7]])

In [7]: gy # shape (2, 3)
Out [7]:
array([[1, 1, 1],
       [3, 3, 3]])
```

NumPy 配列 gx と gy の各.shape を調べればわかるように、gx と gy が同じ配列構造を持つことに注意してください。

図 2.1 は、平面上の実関数 $f(x,y) = x\exp(-x^2-y^2)$ のグラフをワイヤフレームでプロットしたものです。プログラム 2.2-1: wireframe3d.py は与えられた x-成分区間と y-成分区間を等分した 1 次元リストから np.meshgrid を使って x,y-格子行列 xg と yg を求め、高さの格子行列 zg を f(xg,yg) で一気に計算し（みな同じ shape を持つ）、plot_wireframe(xg, yg, zg) でプロットしています。コンソールから IPython を使って描画した場合、マウスで 3D グラフィックスのビューポイントを変化させて眺めることができます。

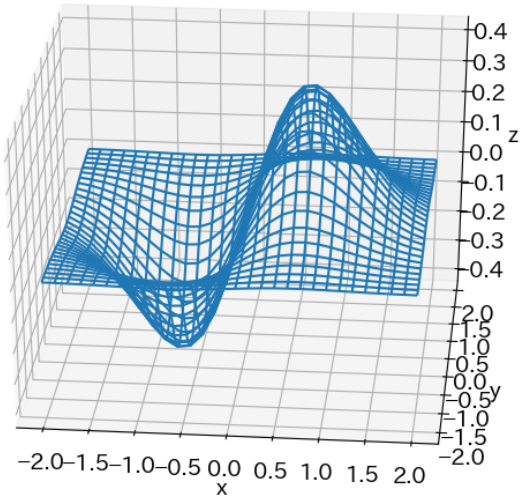


図 2.1　格子点 (x,y) 上の関数値 $f(x,y)$ をワイヤーフレームでプロット。領域 $[-2,2]\times[-2,2]$ をそれぞれ 30 等分して得られる格子点 (x,y) 上の値 $z = x\exp(-x^2-y^2)$ で定まる点 (x,y,z) をワイヤーフレームとしてプロット（プログラム 2.2-1）。

コード 2.2-1　wireframe3d.py

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from mpl_toolkits.mplot3d import Axes3D

def f(x, y: float) -> np.float_:
        return(x * np.exp(-x ** 2 - y ** 2))

xRange = np.linspace(-2,2, 30)
yRange = np.linspace(-2,2, 30)
xg, yg = np.meshgrid(xRange, yRange)

zg = f(xg, yg)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
ax.plot_wireframe(xg, yg, zg)
ax.set(xlabel = 'x', ylabel = 'y', zlabel = 'z')
fig.savefig('wireframe3d.png')
fig.show()
```

2.3 線形代数の計算 `linalg.*`

NumPy には線形代数のためのパッケージ `numpy.linalg` が用意されています。ここでは 3 次の正方行列を例に本書で必要な範囲で線形代数に関わる計算の一部を紹介します。

NumPy 配列は値を格納しておく容器に過ぎないため、配列要素を示す添字には上付き・下付きの区別は必要ありません（表記において行列 A の要素を ${A^i}_j$ や A_{ij}、ベクトル要素を v^i や v_i のように混乱にない程度に自由に記しても構いません）。

2.3.1 対角和、行列式と逆行列 `trace  linalg.detlinalg.inv`

正方行列 $\boldsymbol{A}$ の**対角和**（トレース）$\mathrm{Tr}\,\boldsymbol{A} = \sum_k {A^k}_k$、**行列式** $\det \boldsymbol{A}$ および textbf 逆行列 $\boldsymbol{A}^{-1}$ を NumPy で計算してみましょう。トレースや行列式は座標のとり方に依らない不変量です。

`numpy.trace(a)` またはメソッド `a.trace()` は、2 次元 NumPy 配列 `a` の対角和、`numpy.linalg.det(a)` はその行列式 、そして `numpy.linalg.inv(a)` は正則行列（行列式が非ゼロ）の逆行列を返します。

```
In [1]: import numpy as np

In [2]: a = np.array([[1,2,3], [3,1,1], [2,0,1]])
```

```
In [3]: a
Out[3]:
array([[1, 2, 3],
       [3, 1, 1],
       [2, 0, 1]])

In [4]: a.trace()
Out[4]: 3

In [5]: np.linalg.det(a)
Out[5]: -7.000000000000001

In [6]: inva = np.linalg.inv(a)

In [7]: inva
Out[7]:
array([[-0.14285714,  0.28571429,  0.14285714],
       [ 0.14285714,  0.71428571, -1.14285714],
       [ 0.28571429, -0.57142857,  0.71428571]])

In [8]: a @ inva
Out[8]:
array([[1.00000000e+00, 2.22044605e-16, 1.11022302e-16],
       [0.00000000e+00, 1.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 0.00000000e+00, 1.00000000e+00]])
```

ここでは、その逆行列 `inva` を求めて元の行列との行列積 `a@inva` が単位行列となることを確認しています。整数行列を与えても、NumPy では行列式や逆行列は浮動小数点として計算され、僅かですが数値誤差が混入しています。したがって、次数 `n` の単位行列 `np.eye(n)` とは等しくなりません。

```
In [9]: inva @ a == np.eye(3)
Out[9]:
array([[False,  True, False],
       [ True,  True,  True],
       [False,  True,  True]])
```

数値計算ではこの種の‘数値誤差’が避けられないために、論理式の利用には注意する必要があります。

連続量である微分方程式の解軌道を離散的な点列として計算する微分方程式の数値解の各種の正当性、たとえば、得られた数値列が真の軌道を十分正しく反映しているのかな

ど、得られた計算結果の数値誤差をいかに評価するかは計算シミュレーション全般に及ぶ宿命的な課題で今なお多くの研究が続けられていますす。

演習 2.1行列式がゼロとなるような正方行列を探してみなさい。n 行 m 列のゼロ成分を持つ NumPy 配列を返す `np.zeros((n,m))` と比較してみなさい。

2.3.2 線形連立方程式を解く `linalg.solve`

`numpy.linalg.solve(A, b)` は、非斉次線形連立方程式 $\boldsymbol{Ax} = \boldsymbol{b}$ の解 $\boldsymbol{x}$ を 1 次元 NUmPy 配列として計算します。

n 次正則正方行列 $\boldsymbol{A}$ と列ベクトル $\boldsymbol{b} = [b^0, b^1, \ldots, b^{n-1}]^T$ に対する n 変数非斉次線形連立方程式 $\boldsymbol{Ax} = \boldsymbol{b}$ を考えてみましょう。

$$\boldsymbol{Ax} = \boldsymbol{b} \Leftrightarrow \begin{cases} {A^0}_0 x^0 + {A^0}_1 x^1 + \cdots + {A^0}_{n-1} x^{n-1} = b^0 \\ {A^1}_0 x^0 + {A^1}_1 x^1 + \cdots + {A^1}_{n-1} x^n = b^1 \\ \quad \vdots \\ {A^{n-1}}_0 x^0 + {A^{n-1}}_1 x^1 + \cdots + {A^{n-1}}_{n-1} x^n = b^{n-1} \end{cases}$$

形式的には、逆行列 $\boldsymbol{A}^{-1}$ を求めて解 $\boldsymbol{x}$ を $\boldsymbol{A}^{-1}\boldsymbol{b}$ から求めることができます。しかしながら素朴に考えても、逆行列 $\boldsymbol{A}^{-1}$ に $\mathcal{O}(n^3)$、行列積 $\boldsymbol{A}^{-1}\boldsymbol{b}$ に $\mathcal{O}(n^2)$ の計算量が必要で時間およびメモリ計算量ともに多くしかも計算精度もよくありません。NumPy で逆行列を計算してみると、1000×1000 の乱数行列から次数を 10 倍にすると計算時間は 1000 倍程度になります。

```
In [1]: import numpy as np

In [2]: A100 = np.random.random(100*100).reshape(100,100)

In [3]: %timeit np.linalg.inv(A100)
244 ţs ś 30 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)

In [4]: A1000 = np.random.random(1000*1000).reshape(1000,1000)

In [5]: %timeit np.linalg.inv(A1000)
25.8 ms ś 509 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

次数 n が大きくなると逆行列を求める計算資源は急増し、逆行列をつかう計算法は実用的ではありません。巨大連立線形方程式の解を求めることには多くの需要があり、さまざまな数値解法が今なお工夫されています。

実際に 1000 次元非斉次線形連立方程式を `numpy.linalg.solve(A, b)` を使って計

算してみましょう。

```
In [6]: A = np.random.random(1000*1000).reshape(1000,1000) # 1000 x 1000 乱数行列

In [7]: b = np.random.random(1000) # 乱数ベクトル

In [8]: %timeit np.linalg.solve(A, b)
12.2 ms ś 249 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

In [9]: x = np.linalg.solve(A, b)

In [10]: np.max(np.linalg.inv(A) @ b - x)
Out[10]: 1.1429746038515987e-13
```

逆行列を計算するよりも速く解を求めています。最後の計算では、逆行列方式の結果と `numpy.linalg.solve(A, b)` で得た結果との差異の最大値を求めています。

2.3.3 データの最小二乗フィット `linalg.lstsq`

n 個の測定データ対 $\{(x^i, y^i\}_{i=0,\ldots,n-1}$ が与えられているとき、これら測定データからテータ対 (x, y) の関数関係 $y = F(x)$ を推定することを**データフィット**といいます。

ここでは、2 変数 X, Y を観察し、その関係として、X から Y を説明 (explain) する仕方（それが関数関係 $Y = F(X)$）を問題にします。このとき、$Y = F(X)$ を**回帰方程式** (regression equation) といい、この X と Y の定量的な関係の構造ををを**回帰モデル**と呼ぶ。。一方、X とと Y との間にどの程度関係があるかを**相関** (correlation) といい、この場合には Y ををを X で説明するものではありません。

2.3.3.1 回帰モデル

X と Y とを $Y = aX + b$ のように Y ををを X の一次式で説明するモデルを**線形回帰** (linear regression) といい、それ以外を非線形回帰と呼びます。ここでは、非線形モデルを含むように一般的な場合を考えます。

いま、m 個の関数 $f_0, f_1, \ldots, f_{m-1}$ が存在すると仮定して、データ対 $\{(x^i, y^i)\}$ に対して、各関係式

$$y^i \approx F(x^i) = p^0 f_0(x^i) + p^1 f_1(x^1) + \cdots + p^{m-1} f_{m-1}(x^i)$$

が最良であるように係数 $p^0, p^1, \ldots, p^{m-1}$ を決めて推定したい関数 F を構成することを考えましょう。

いま、n 個の測定データの x-成分なからなる 1 次元配列 $\boldsymbol{x} = (x^0, \ldots, x^{n-1})$ に対して仮

定した m 個の関数 $f_0, f_1, \ldots, f_{m-1}$ それぞれを $\boldsymbol{x}$ の成分に適用して得られる m 個の n-次元ベクトル $\boldsymbol{f}_k$ を並べて行列 $\boldsymbol{A}$ を定めておきます。

$$A \equiv [\boldsymbol{f}_0, \ldots, \boldsymbol{f}_{m-1}] = \begin{bmatrix} f_0(x^0) & f_1(x^0) & \ldots & f_{m-1}(x^0) \\ f_0(x^1) & f_1(x^1) & \ldots & f_{m-1}(x^1) \\ \vdots & \vdots & \ldots & \vdots \\ f_0(x^{n-1}) & f_1(x^{n-1}) & \ldots & f_{m-1}(x^{n-1}) \end{bmatrix}.$$

これより、求める m 個の係数からなる係数ベクトル $\boldsymbol{p} = (p^0\, p^1\, \ldots\, p^{m-1})$ をつかって $\boldsymbol{Ap}$ が測定データの y-成分からなるベクトル $\boldsymbol{y} = (y^0, \ldots, y^{n-1})$ に近いという関係式

$$\boldsymbol{y} \approx \boldsymbol{Ap}, \qquad \begin{bmatrix} y^0 \\ y^1 \\ \vdots \\ y^{n-1} \end{bmatrix} \approx \begin{bmatrix} f_0(x^0) & f_1(x^0) & \ldots & f_{m-1}(x^0) \\ f_0(x^1) & f_1(x^1) & \ldots & f_{m-1}(x^1) \\ \vdots & \vdots & \ldots & \vdots \\ f_0(x^{n-1}) & f_1(x^{n-1}) & \ldots & f_{m-1}(x^{n-1}) \end{bmatrix} \begin{bmatrix} p^0 \\ p^1 \\ \vdots \\ p^{m-1} \end{bmatrix}$$

を問題にします。

回帰関係をどのように決定するか（どう近いかを決める）方法の 1 つに**最小二乗フィット**があり

$$\sum_{i=0}^{n} \left(y^i - \Big(\sum_{k=0}^{m-1} p^k f_k(x^i) \Big) \right)^2$$

を最小にする係数 $p^0, p^1, \ldots, p^{m-1}$ を求める方法です。今の場合、ノルム $\|\boldsymbol{y} - \boldsymbol{Ap}\|^2$ を最小化するベクトル $\boldsymbol{p}$ を求めることが問題の解となります。

`numpy.linalg.lstsq(A, y)` はこの最小二乗解である係数列ベクトル $\boldsymbol{p}$ を含んだ結果を返します。正確には次のように 4 つの返り値があり、求めたい係数ベクトル $\boldsymbol{p}$ はその 0 番目 `np.linalg.lstsq(A, y)[0]` の 1 次元配列です。

```
p, residues, rank, s = np.linalg.lstsq(A, y)
```

ここで、residues は残差の合計、rank は行列 `A` のランク、`s` は `A` の特異値です。

コード 2.3-1　minimal_fit

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

# データ対 X, Y
x: np.ndarray = np.array([0.5, 1.3, 2.6, 3.7, 4.5, 6.1, 9.2, 11])
```

```
y: np.ndarray = np.array([2.5, 4.1, 8.5, 13.5, 25.1, 36, 80, 138])
nParam: int = 3 # 係数ベクトルPの要素数
A = np.empty((len(x), nParam)) # shapeの整合性はプログラマの責任
A[:,0] = 1
A[:,1] = x
A[:,2] = x ** 2
p = np. linalg.lstsq(A, y, rcond=None)[0]

fig, ax = plt.subplots(figsize=(8.0, 5.0))
ax.plot(x, y, 'ro')
xx = np.linspace(np.min(x)-1, np.max(x)+1, 20)
yy = p[0] + p[1]*xx + p[2]*xx **2
ax.plot(xx, yy, color='blue', linestyle='solid', linewidth = 1.0)
ax.set(xlabel='x', ylabel='y')
#plt.show()
```

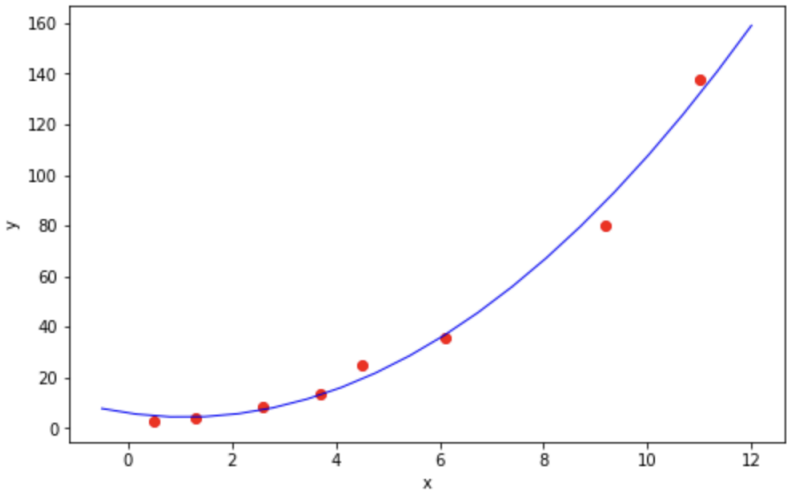


図 2.2　丸印のデータ対 $(0.5, 2.5)$,$(1.3, 4.1)$,$(2.6, 8.5)$,$(3.7, 13.5)$,$(4.5, 25.1)$,$(6.1, 36)$,$(9.2, 80)$,$(11, 138)$ **のプログラム 2.3-1 による関数** $\{1, x, x^2\}$ **を使った最小 2 乗フィット。得られた最適フィット径数を使った 2 次式** $y = 6.01003745 - 2.90272221x + 1.30489893x^2$ **の関数グラフと元データとの比較。**

プログラム 2.3-1 `minimal_fit.py` は、8 個の測定値の組 $(0.5, 0.2), (1.3, 1, 4), (2.6, 7), (3.7, 12, 5), (4.5, 22, 1)$ について、2 次関数

$$y = p_0 + p_1 x + p_2 x^2$$

でフィットさせるために、関数を $f_0(x) = 1, f_1(x) = x, f_2(x) = x^2$ と仮定して係数ベクトル (p_0, p_1, p_2) を求め、測定値を丸でフィットした点は $\times$ 印でつなげてプロットします。

```
In [15]: %run minimal_fit.py

In [16]: A
Out[16]:
array([[  1.  ,   0.5 ,   0.25],
       [  1.  ,   1.3 ,   1.69],
       [  1.  ,   2.6 ,   6.76],
       [  1.  ,   3.7 ,  13.69],
       [  1.  ,   4.5 ,  20.25],
       [  1.  ,   6.1 ,  37.21],
       [  1.  ,   9.2 ,  84.64],
       [  1.  ,  11.  , 121.  ]])

In [17]: p
Out[17]: array([ 6.01003745, -2.90272221,  1.30489893])
```

これより、観測データは 2 次間数の範囲では $y = 6.01003745 - 2.90272221x + 1.30489893x^2$ でフィットできるとなりました。図 2.2 は `np.linalg.lstsq` を使う最小二乗法によるフィットの良し悪しは、最初に仮定する関数列 $\{f_k(x)\}$ の関数形と関数の個数に依存します。

2.4 SymPy の利用

SymPy は Python で数式記号処理を行うパッケージで、代数、整数論、解析、組み合わせ論、微分幾何、離散 Fourier 変換、Lie 代数など高等数学における計算利用を可能にします[4)]。

特に SymPy は日常的で煩雑な計算、たとえば有理式の展開や因数分解、特殊関数を含む多くの関数を含む（偏）微積分や級数展開などの解析的計算、簡単な微分方程式の求積、行列の固有値とその固有ベクトルの確認に重宝します。

ここでは、NumPy や SciPy との親和性に注目し、SymPy 関数の NumPy の数値計算を視野に入れて SymPy の一部を紹介します。SymPy には知っておくと有用な多数の関数やメソッドが用意されているため、SymPy の利用に当たっては SymPy Documenttation `https://www.sympy.org` の SymPy Tutorial に目を通しておくと良いでしょう。

4) Python 言語でプログラミングすることのできる本格的なオープンソース数式処理系に SageMath `https://www.sagemath.org` があります。インストールしたパソコンで利用したり SageWorksheet としてクラウドからの利用も可能です。

2.4.1 記号変数 symbols

SymPy で記号変数（SymPy シンボル）を使うためには、

```
sympy.symbols()
```

で生成した SymPy シンボル（SymPy 変数）を Python 変数に代入します。

SymPy シンボルの生成では、SymPy シンボルとしたい文字列をカンマ (,) または空白 (␣) で区切って並べた上で全体を引用符 (') または (") で囲んで複数の SymPy シンボルを扱うことができます。

`y, x, b = sympy.symbols('x y a')` のような Python 変数への SymPy シンボルの割当は構文上可能ですが、大きな混乱を招くことになります。SymPy シンボルを同じ文字列を持つ Python 変数に代入するように心がけ、正しく記号計算が実行できるようにします。

```
In [1]: import sympy as sy

In [2]: a,b,c,d  = sy.symbols('a, b, c, d')

In [3]: x, y = sy.symbols('x y')

In [4]: x = a + b

In [5]: y = x ** 2

In [6]: y
Out[6]: (a + b)**2
```

2.4.1.1 変数の代入 .subs

SymPy メソッド

```
.subs(expr,value)
```

は、SymPy オブジェクト内の SymPy シンボル `expr` に SymPy シンボルまたは値 `value` を代入 (substitution) します。複数の SymPy シンボルに代入するには、SymPy シンボルと代入値の組をタプルとしたリストを与えます。

```
In [7]: sy.sin(x).subs(x, 2*a)
Out[7]: sin(2*a)
```

```
In [8]: sy.exp(x*y).subs([(x, y ** 2), (y, y**2)])
Out[8]: exp((a + b)**24)
```

SymPy 定数には、π は sympy.pi、虚数単位 i は sympy.I（ローマ字大文字）、無限大 ∞ は sympy.oo（オー・オー）などがあります。SymPy の組み込み関数 sin は sympy.sin()、指数関数 exp は sy.exp を始め、初等関数以外の特殊関数があります。表 2.3 に NumPy で利用可能な初等的な関数と各種数列の一部を掲載します。面倒な計算を要する数列の値が有理数で求めることができるため重宝します。例えば n 項までの調和級数 $\sum_{k=1}^{n} 1/k$ の 20 番目は次で得られます。

```
In [9]: sy.harmonic(20)
Out[9]: 55835135/15519504
```

関数	SymPy 関数	関数	SymPy 関数
絶対値	Abs	階乗	factorial
実部と虚部	re, im	複素共役	conjugate
複素数の偏角	arg	平方根	sqrt
指数	exp	対数	log
最大、最小	Max, Min	天井と床	ceiling, floor
分数部分	frac		
三角関数群	sin, cos, tan	逆三角関数	asin, acos, atan
	csc, sec, cot		acsc, asec, acot
双曲線関数	sinh, cosh, tanh	逆双曲線関数	asinh, acosh, atanh
	csch, sech, coth		acsch, asech, acoth
二項係数	binomial	ベル数	bell
カタラン数	catalan	ベルヌーイ数	bernoulli
オイラー数	euler	フィボナッチ数	fibonacci
調和数	harmonic	ルカス数	lucas
分割数	partition	スターリング数	stirling

表 2.3　SymPy で利用可能な主な初等的関数と数列（一部）

2.4.1.2　等式と真偽判定 Eq　srepr　.equals

SymPy 関数

```
sympy.Eq(exprA, exprB)
```

は、SymPy シンボルを含む表式 `exprA` と `exprB` を使って SymPy 等式 $\mathrm{exprA} = \mathrm{exprB}$ を設定します。SymPy 関数

```
sympy.srepr(expr)
```

は、SymPy 表式 `expr` が S どのように表現されているか表現木を表します。

等式として $\mathrm{exprA} = \mathrm{exprB} \Leftrightarrow \mathrm{exprA} - \mathrm{exprB} = 0$ は正しいのですが、SymPy 表現木が違っていると Python の真偽演算 == では False となり、数式評価には注意が必要です。$\sin 2a$ と $2 \sin a \cos a$ は SymPy 表現木が異なっているために真偽判定==は False となります。

表式が数学的に等しいかを調べるには、`exprA.equals(exprB)` を使って True を確認するか、`simplify(exprA - exprB)` を簡約化して 0 になるかを確かめます。

```
In [9]: sy.Eq(a + b, c)
Out[9]: Eq(a + b, c)

In [10]: sy.Eq(a + b, c) == sy.Eq(a + b - c, 0)
Out[10]: False

In [11]: sy.srepr(sy.Eq(a + b, c))
Out[11]: "Equality(Add(Symbol('a'), Symbol('b')), Symbol('c'))"

In [12]: sy.srepr(sy.Eq(a + b - c, 0))
Out[12]: "Equality(Add(Symbol('a'), Symbol('b'), Mul(Integer(-1), Symbol('c'))), Integer(0))"

In [13]: sy.Eq(x, a + b)
Out[13]: True

In [14]: sy.srepr(x)
Out[14]: "Add(Symbol('a'), Symbol('b'))"

In [15]: sy.sin(2 * a) == 2*sy.sin(a)*sy.cos(a)
Out[15]: False

In [16]: sy.sin(2 * a).equals(2*sy.sin(a)*sy.cos(a))
Out[16]: True

In [17]: sy.simplify(sy.sin(2 * a)-2*sy.sin(a)*sy.cos(a))
Out[17]: 0
```

2.4.1.3 展開と因数分解、括り出し expand factor collect

SYmPy 関数

```
sympy.expand(expr)
```

は、有理係数を持つ多項式や有理式 expr を展開します。一方、

```
sympy.factor(expanded)
```

はこの逆操作で、表式 expanded を可能な範囲で因数分解します。

また、SymPy 関数

```
collect(expr, ext)
```

は、表式 expr を項 ext で括り出します。次の例は $(x+y+a)^3$ を例に展開と因数分解、そして括り出しを示しています。括りだすを積 xy の形にしてもよく、また複数の項の並びをリストで指定しても構いません。ただし、項の並び順によって括りだされ方は違います。

$$
\begin{aligned}
(x+y+a)^3 &\longrightarrow a^3+3a^2x+3a^2y+3ax^2+6axy+3ay^2+x^3+3x^2y+3xy^2+y^3 \\
&\xrightarrow[x]{collect} a^3+3a^2y+3ay^2+x^3+x^2(3a+3y)+x(3a^2+6ay+3y^2)+y^3 \\
&\xrightarrow[[x,y]]{collect} a^3+3a^2y+3ay^2+x^3+x^2(3a+3)+x(3a^2+6ay+3y2)+y^3 \\
&\xrightarrow[[y,x]]{collect} a^3+3a^2x+3ax^2+x^3+y^3+y^2(3a+3x)+y(3a^2+6ax+3x^2) \\
&\xrightarrow[x*y]{collect} a^3+3a^2x+3a^2y+3ax^2+3ay^2+x^3+xy(6a+3x+3y)+y^3
\end{aligned}
$$

```
In [1]: import sympy as sy

In [2]: x, y, a = sy.symbols('x y a')

In [3]: expr = sy.expand((x + y + a) ** 3)

In [4]: expr
Out[4]: a**3 + 3*a**2*x + 3*a**2*y + 3*a*x**2 + 6*a*x*y + 3*a*y**2 + x**3 +
3*x**2*y + 3*x*y**2 + y**3

In [5]: sy.factor(expr)
Out[5]: (a + x + y)**3

In [6]: sy.collect(expr, x)
```

```
Out[6]: a**3 + 3*a**2*y + 3*a*y**2 +expr x**3 + x**2*(3*a + 3*y) +
x*(3*a**2 + 6*a*y + 3*y**2) + y**3

In [7]: sy.collect(expr, y)
Out[7]: a**3 + 3*a**2*x + 3*a*x**2 + x**3 + y**3 + y**2*(3*a + 3*x) + y*(3*a**2 + 6*a*x + 3*x**2)

In [8]: sy.collect(expr, [x, y])
Out[8]: a**3 + 3*a**2*y + 3*a*y**2 + x**3 + x**2*(3*a + 3*y) +
x*(3*a**2 + 6*a*y + 3*y**2) + y**3

In [9]: sy.collect(expr, [y, x])
Out[9]: a**3 + 3*a**2*x + 3*a*x**2 + x**3 + y**3 + y**2*(3*a + 3*x) +
y*(3*a**2 + 6*a*x + 3*x**2)

In [10]: sy.collect(expr, x*y)
Out[10]: a**3 + 3*a**2*x + 3*a**2*y + 3*a*x**2 + 3*a*y**2 + x**3 +
x*y*(6*a + 3*x + 3*y) + y**3

In [11]: sy.collect(expr, [x, a*x])
Out[11]: a**3 + 3*a**2*y + a*x*(3*a + 6*y) + 3*a*y**2 + x**3 +
3*x**2*y + x*(3*a*x + 3*y**2) + y**3

In [12]: sy.collect(sy.expand((x*sy.log(x) + x + a) ** 2),x* sy.log(x))
Out[12]: a**2 + 2*a*x + x**2*log(x)**2 + x**2 + x*(2*a + 2*x)*log(x)
```

sympy.expand_trig は三角関数式の公式に従って展開、sympy.trigsimp はその逆を計算します。

```
In [13]: sy.expand_trig(sy.cos(x+y))
Out[13]: -sin(x)*sin(y) + cos(x)*cos(y)

In [14]: sy.trigsimp(2 * sy.tan(x)/(1 - sy.tan(x) ** 2))
Out[15]: tan(2*x)
```

2.4.1.4 簡約 simplify cancel apart

sympy.simplify(expr) は、SymPy 表式 expr をより単純な形に簡約化 (simplify) を試みます。ただし、‘ふさわしい単純な形 ‘に簡約できる保証があるわけはなく、たとえば有理係数を持つ多項式式の因数分解は sympy.factor の仕事です。

$$\frac{x^3 + ax^2 - x - a}{x^2 + 2ax + a^2} \Rightarrow \frac{x^2 - 1}{x + a}$$

```
In [1]: import sympy as sy

In [2]: x, y, a, b = sy.symbols('x y a b')

In [3]: sy.simplify((x**3 + a * x**2 - x - a) / (x**2 + 2*a*x + a**2))
Out[3]: (x**2 - 1)/(a + x)

In [4]: sy.simplify(sy.gamma(x) / sy.gamma(x-2))
Out[4]: (x - 2)*(x - 1)
```

SymPy 関数

```
sympy.cancel(expr)
```

は、任意の有理関数を共通因子をもたない標準正規形 $p(x)/q(x)$ を返します。ここで $p(x), q(x)$ の最高次数は整数です。

$$x + \cfrac{1}{2x + 1 + \cfrac{6}{x}} \Rightarrow \frac{2x^3 + x^2 + 7x}{2x^2 + x + 6}$$

```
In [5]: sy.cancel(x+(1/(1+2*(x+3/x))))
Out[5]: (2*x**3 + x**2 + 7*x)/(2*x**2 + x + 6)
```

SymPy 関数

```
sympy.apart(expr)
```

は、有理式 `expr` を部分分数に展開します。

$$\frac{(3x^2 + 4x + 5}{x^3 3 + 3x^2 - 10x - 24} = \frac{37}{14(x+4)} - \frac{9}{10(x+2)} + \frac{44}{35(x-3)}$$

```
In [6]: sy.apart((3*x**2 + 4*x +5)/(x**3 + 3*x**2 - 10*x - 24))
Out[6]: 37/(14*(x + 4)) - 9/(10*(x + 2)) + 44/(35*(x - 3))
```

メソッド`.simplify` と紛らわしい関数に `sympy.sympify` があります。`sympy.sympify(str_expr)` は、文字列 `str_expr` を SymPy オブジェクトに変換します。

```
In [7]: str_expr = 'x**2 + 3*x - 1/2'

In [8]: sy.sympify(str_expr).subs(x, 3)
Out[8]: 35/2
```

2.4.1.5 極限 `limit`

SymPy 関数

```
sympy.limit(f, x, a)
```

は、SymPy 表式 f の変数 z を値 z_0 に近づけたときの**極限** (limit)

$$\lim_{z \to z_0} f(z)$$

を評価します。オプション `dir="+"`をつけると右極限（$\lim_{z \to z_0+0} f(z)$）、`dir="-"`をつけると左極限（$\lim_{z \to z_0-0} f(z)$）が得られます。

```
In [9]: sy.limit(sy.sin(x-a)/(x-a), x, a)
Out[9]: 1

In [10]: sy.limit(1/(x-a), x, a, dir="+")
Out[10]: oo

In [11]: sy.limit(1/(x-a), x, a, dir="-")
Out[11]: -oo
```

2.4.1.6 微分と微分方程式の求積 `diff Derivative doit dsolve`

関数 f の変数 x_i に関する偏微分 $\partial f/\partial x_i$ をしばしば f_{x_i} と略記し、また、$\partial^2 f/\partial x_i \partial x_j = f_{x_i,x_j}$ などとも記します。

SymPy 関数

```
sympy.diff(expr, x,y,..)  またはメソッド expr.diff(x,y..)
```

は SymPy 表式 `expr` について指定した変数 `x`（続いて `y,z` などの順番）について偏微分します。また、`sympy.diff(expr, x, n)` は変数 x について指定した n 回数だけ連続回微分（偏微分）します（n を省略すると 1 回）。

```
In [12]: expr = sy.exp(x*y)

In [13]: expr.diff(x)
Out[13]: y*exp(x*y)

In [14]: expr.diff(x,y,2)
Out[14]: x*(x*y + 2)*exp(x*y)
```

SymPy 関数

```
sympy.Derivative(expr,x,y..)
```

は、SymPy 表式 expr の微分をその場で評価しないで微分式に留めます。この性質を使うと見通しの良い計算が可能になる場合があります。一般関数 f, g を SymPy コンストラクタ Function で定義しておき、sy.Derivative(f * g, x) によって積 fg の微分 $(fg)_x$ を考えることができ、具体的関数 $f(x,y) = \sin xy, g(x,y) = x + y$ を代入して微分計算式を得ます。メソッド.doit() は、表式を評価してその結果を計算します。

```
In [15]: f = sy.Function('f')(x, y)

In [16]: g = sy.Function('g')(x, y)

In [17]: dx_fg = sy.Derivative(f * g, x)

In [18]: dx_fg
Out[18]: Derivative(f(x, y)*g(x, y), x)

In [19]: dx_fg.subs([(f, sy.sin(x*y)), (g, x+ y)])
Out[19]: Derivative((x + y)*sin(x*y), x)

In [20]: dx_fg.subs([(f, sy.sin(x*y)), (g, x+ y)]).doit()
Out[20]: y*(x + y)*cos(x*y) + sin(x*y)
```

Function で生成した未定義関数を使うと微分方程式を表すことができます。sympy.dsolve(diffeq, y) は、$y'' - y = \mathrm{e}^x$ のような簡単な微分方程式の解を陽に与えます。

```
In [21]: y = sy.Function('y')(x)

In [22]: diffeq = sy.Eq(y.diff(x,x) - y, sy.exp(x))

In [23]: sy.dsolve(diffeq, y)
Out[23]: Eq(y(x), C2*exp(-x) + (C1 + x/2)*exp(x))
```

2.4.1.7 積分 integrate

SymPy 関数

```
sympy.integrate(f, x)
```

またはメソッド f.integrate(x)

は表式 f の不定積分 $\int \mathrm{f}\, dx$ を試みます（未定定数は省かれます)。積分できないと判断したときは、Integral で返されます。

```
In [24]: sy.integrate(sy.sin(x) ** 2, x)
Out[24]: x/2 - sin(x)*cos(x)/2

In [25]: sy.integrate(sy.exp(-x**2), x)
Out[25]: sqrt(pi)*erf(x)/2

In [26]: sy.integrate(expr, x) # 初等関数内で積分可能
Out[26]: Integral(sqrt(a*x**2 + b*x + c), x)
```

SymPy 関数

```
sympy.integrate(f, (x, a, b)
```

は、区間 $[a, b]$ で定積分 $\int_a^b \mathrm{f}\, dx$ を試みます。

```
In [27]: sy.integrate(sy.exp(-x**2), (x, 0, sy.oo))
Out[27]: sqrt(pi)/2

In [28]: sy.integrate(sy.exp(x*y), x)
Out[28]: Piecewise((exp(x*y)/y, Ne(y, 0)), (x, True))

In [29]: sy.integrate(x**y * sy.exp(-x), (x, 0,sy.oo))
Out[29]: Piecewise((gamma(y + 1), re(y) > -1), (Integral(x**y*exp(-x), (x, 0, oo)), True))
```

ここで Piecewise は区分わけを意味し、

$$\int_0^\infty x^y \exp(-x)\, dx = \begin{cases} \Gamma(y+1) & \mathrm{Re}(y) > -1 \\ \int_0^\infty x^y \exp(-x)\, dx & \text{それ以外} \end{cases}$$

を表しています。もちろん、どんな関数でも積分できるわけではありませんが簡単な式でも手間のかかるような計算には重宝します。

2.4.1.8 整形出力 latex print_mathml

SymPy はその出力結果をさまざまな方式で整形出力することができます。

SymPy 整形関数

```
sympy.latex(expr)
```

は、SymPy 表式 `expr` を LaTeX コマンドで表します。ただし、バックスラッシュ (\) は \\ とエスケープして出力されるため、そのままコピーして使える LaTeX 出力を得るために Python の print に渡して表示します。`sympy.print_mathml(expr)` は SymPy 表式 `expr` を MathML として出力します。

次は 不定積分 $\int \sqrt{\frac{1}{x}}\,dx = x\sqrt{\frac{1}{x}}$ の表示例です。

```
In [30]: expr = sy.sqrt(1/x).integrate(x)

In [31]: expr.diff(x)
Out[31]: sqrt(1/x)

In [32]: print(sy.latex(expr))
2 x \sqrt{\frac{1}{x}}

In [33]: sy.print_mathml(expr)
<apply>
        <times/>
        <cn>2</cn>
        <ci>x</ci>
        <apply>
                <root/>
                <apply>
                        <power/>
                        <ci>x</ci>
                        <cn>-1</cn>
                </apply>
        </apply>
</apply>
```

2.4.2 SymPy の行列計算

SymPy には NumPy と同様に多次元配列を取り扱うためのクラス Array がありますが、$m \times n$ の SumPy 行列に特化したクラス Matrix を使った行列計算を紹介します[5)]。

2.4.2.1 SymPy 行列 `Matrix .row .col`

`sympy.Matrix(list)` は深さの揃ったリスト list から SymPy 行列を返します。リスト要素に SymPy シンボルを含むことが可能です。長さ mn の単リスト slist を与えて `sympy.Matrix(m, n, slist)` によって $m \times n$ 配列とすることもできます。

5) NumPy にもクラス matrix がありますが、節**??**で注意したようにクラス matrix は推奨されず一貫してクラス array を利用することが勧告されています。

SymPy 配列と NumPy を併用する際には配列形状についての注意が必要です。コンストラクタ sympy.Matrix に長さ n の単リストを渡すと、その shape は (n,1) となり $n \times 1$ 配列（線形代数で言う列ベクトル）となります。単リストから生成した 1 次元 NumPy 配列の shape が (n,) であったこととは違っています。クラス Matrix で SymPy 行列を使うときは列ベクトルと行ベクトルとは区別されます。

```
In [1]: import sympy as sy

In [2]: a,b,c,d,e,f = sy.symbols('a b c d e f')

In [3]: x,y = sy.symbols('x y')

In [4]: x = sy.Matrix([a,b,c])

In [5]: x.shape
Out[5]: (3, 1)

In [6]: y = sy.Matrix([[a,b,c]])

In [7]: y.shape
Out[7]: (1, 3)

In [8]: y.reshape(3,1)==x
Out[8]: True
```

SymPy メソッド

```
A.row(i)
```

は SymPy 行列 `A` から第 i 行を、

```
A.col(j)
```

は第 j 列を SymPy 配列として返します。行列の (i, j) 成分を得たいときは `A[i,j]` とスライスします。SymPy 行列の形状は NumPy と同じようにメソッド `.shape` で取得でき、また `.reshape` で形状を変えることや SymPy 行列のスライスも可能です。

```
In [9]: A = sy.Matrix([[a,b,c], [d, e,f]])

In [10]: A # shape (2,3)
Out[10]:
Matrix([
[a, b, c],
```

```
[d, e, f]])

In [11]: A.row(0) # A[0,:] と同じ
Out[11]: Matrix([[a, b, c]])

In [12]: A.col(0) # A[:,0] と同じ
Out[12]:
Matrix([
[a],
[d]])

In [13]: A[0,2]
Out[13]: c

In [14]: A.row(0).col(2)
Out[14]: Matrix([[c]])

In [15]: A.reshape(3,2)
Out[15]:
Matrix([
[a, b],
[c, d],
[e, f]])
```

節 2.4.3 で繰り返し確認するように、クラス Matrix で生成した配列はあくまでも行列（2 次元配列）であって、NumPy のように多次元配列として一貫した取り扱いはできません。SymPy の多次元配列は節 2.4.2.5 で紹介するコンストラクタ Array を使ってテンソルとして扱います。

2.4.2.2 SymPy 行列の積とドット積 `.multiply` `.dot`

SymPy 行列同士の行列積は演算子 `*` を使って `A * B` などと計算できます。NumPy のクラス array における要素ごとに計算する掛け算演算子 `*` と紛らわしくなるため、SymPy メソッド

```
A.multiply(B)
```

を使って可読性を向上させると良いでしょう。

クラス Matrix のメソッド `x.dot(y)` は、（行と列ベクトルの組み合わせいかんに依らず）長さの揃ったベクトル同士の総和積（ドット積）を計算してスカラー値を返します。行列積の計算に `.dot` を使うと、警告 SymPyDeprecationWarning が出されます。

```
In [29]: x.dot(y)
Out[29]: a**2 + b**2 + c**2

In [30]: y.dot(x)
Out[30]: a**2 + b**2 + c**2

In [31]: x.dot(x)
Out[31]: a**2 + b**2 + c**2

In [32]: A.dot(x) # 行列計算には * を使うよう警告
    SymPyDeprecationWarning(
Out[32]: [a**2 + b**2 + c**2, a*d + b*e + c*f]

In [33]: A.multiply(x) # A * x と同じ
Out[33]:
Matrix([
[a**2 + b**2 + c**2],
[   a*d + b*e + c*f]])

In [34]: x.multiply(A) # エラー Matrix size mismatch
```

オプション hermitian = True をつけると複素共役で計算され総和積（ドット積）になります。

```
In [35]: y = sy.Matrix([1, 2, 3*sy.I])

In [36]: y.dot(y)
Out[36]: -4

In [37]: y.dot(y, hermitian=True)
Out[37]: 14
```

なお、コンストラクタにシンボル（や数）をそのまま `sympy.Matrix(2)` とするとリスト値ではないと TypeError が返され、NumPy と違ってスカラーとして扱われません。空リスト `sympy.Matrix([])` あるいは空 `sy.Matrix()` で生成したものは、shape (0,0) の 0×0 行列でスカラーではありません。

2.4.2.3 行列式と逆行列 `det  .inv  .rank`

SymPy 関数

`sympy.det(A)`　またはクラス Matrix メソッド`A.det()`

は正方行列の行列式 $\det A$、メソッド

`A.inv()` または-1 乗 `A**-1`

は逆行列 A^{-1} を計算します。SymPy シンボルを含む場合、逆であることの確認には `simplify` などで簡約します。

```
In [21]: A = sy.Matrix([[a,b], [c,d]])

In [22]: A.det()
Out[22]: a*d - b*c

In [23]: invA = A.inv() # A ** -1 と同じ

In [24]: invA
Out[24]:
Matrix([
[ d/(a*d - b*c), -b/(a*d - b*c)],
[-c/(a*d - b*c),  a/(a*d - b*c)]])

In [25]: A.inv(method="LU")
Out[25]:
Matrix([
[(1 + b*c/(a*(d - b*c/a)))/a, -b/(a*(d - b*c/a))],
[         -c/(a*(d - b*c/a)),      1/(d - b*c/a)]])

In [26]: sy.simplify(invA * A)
Out[26]:
Matrix([
[1, 0],
[0, 1]])
```

SymPy の逆行列計算はデフォルトでガウス消去法を使いますが、LU 分解法を指定することもできます（結果表式が異なります)。

逆行列計算は NumPy 以上に多くの計算資源を使うので、大きさサイズの計算が必要な場合には目的に合わせてソルバーや分解アルゴリズムなど他の方法を検討してください。

2.4.2.4 有理計算と浮動小数評価 `Rational  evalf`

SymPy と NumPy とを併用する利点の一つに SymPy の有理計算があります。SymPy を使うと浮動小数点を使わずに分数計算を実行することができます。

整数 SymPy 行列の逆行列は分数で計算結果を返します。NumPy の逆行列計算

`np.linalg.inv(A)` は数値誤差付きの浮動小数計算で実行されました（節 2.3.1 参照）。実用上、それで問題はないのですが、計算上の見通しは良くありません。

```
In [27]: B = sy.Matrix([[1,2,3], [3,1,1], [2,0,1]])

In [28]: B.rank()
Out[28]: 3

In [29]: invB = B.inv()

In [30]: invB
Out[30]:
Matrix([
[-1/7,  2/7,  1/7],
[ 1/7,  5/7, -8/7],
[ 2/7, -4/7,  5/7]])

In [31]: invB.multiply(B) # invB * B と同じ
Out[31]:
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

ただし、SymPy 計算において有理数 (rational number) として取り扱うには注意が必要です。SymPy 行列を整数要素以外で生成する際にその要素を 2/3 のように与えると浮動小数（sympy.core.numbers.Float）として実行されます。この結果、SymPy で逆行列を計算すると NumPy 計算と同じくわずかに数値誤差が残ります。

```
In [32]: Bf = sy.Matrix([[-1/7,2/7,1/7], [1/7,5/7,-8/7], [2/7,-4/7,5/7]])

In [33]: invBf = Bf.inv()

In [34]: invBf
Out[34]:
Matrix([
[0.999999999999999, 2.0, 3.0],
[              3.0, 1.0, 1.0],
[              2.0,   0, 1.0]])

In [35]: invBf.multiply(Bf)
Out[35]:
Matrix([
```

```
[                  1.0, -2.22044604925031e-16,   0],
[-5.55111512312578e-17,                  1.0,   0],
[                   0,                    0, 1.0]])
```

SymPy 表式内の

```
sympy.Rational(p, q)
```

または

```
sympy.Integer(p)/sympy.Integer(q)
```

は、SymPy 計算において有理数 $p/q \in \mathbb{Q}$ の扱いとなります （表式中に浮動小数が混入すると浮動小数にキャストされてしまいます)。

```
In [36]: a = sy.Rational(2,3)

In [37]: a
Out[37]: 2/3

In [38]: type(a)
Out[38]: sympy.core.numbers.Rational

In [39]: b = sy.Integer(1)/sy.Integer(3)

In [40]: type(b)
Out[40]: sympy.core.numbers.Rational

In [41]: a+b
Out[41]: 1

In [42]: type(a+b)
Out[42]: sympy.core.numbers.One

In [43]: (2/3 + a)/2
Out[43]: 0.666666666666667
```

SymPy オブジェクトのメソッド

```
expr.evalf(n)
```

は SymPy 表式 `expr` の浮動小数点表示を評価します（n は桁数指定で、省略可能です)。SymPy シンボル `a`, `b` などを含む表式の浮動小数点評価では、`.evalf` の引数にフラグ

subs を使ってシンボルと値とをコロン (:) でペアにして並べて辞書型として

sexpr.evalf(n, subs={$a:a0, b:b0, ..$})

のようにします。すべてのシンボルの値をすべて解決しないと、浮動小数点表示内にシンボルが残ります。

```
In [44]: sy.sqrt(7).evalf()
Out[44]: 2.64575131106459

In [45]: sy.sqrt(7+a+b).evalf(subs={a:-4})
Out[45]: 2.64575131106459*(0.142857142857143*a + 0.142857142857143*b + 1)**0.5

In [46]: sy.sqrt(7+a+b).evalf(subs={a:-4, b:-1})
Out[46]: 1.41421356237310

In [332]: sy.pi.evalf(n=30)
Out[332]: 3.14159265358979323846264338328
```

evalf 内に n=N をつけて表示桁数指定も可能です（デフォルト n=15)。

2.4.2.5 SymPy のテンソル計算 tensorproduct

SymPy でテンソルを取り扱うためにはコンストラクタ Array にリストを適用して SymPy 多次元配列を生成します。長さ n の単リストから得られた 1 階テンソルの shape は NumPy と同じように (n,) となり、コンストラクタ Matrix を使った SymPy 配列での (n,1) とは異なります。

SymPy 関数

sympy.tensorproduct(tA, tB)

は、SymPy 多次元配列 tA, tB のテンソル積 tA $\otimes$ tB を計算します。

$$tA = \begin{bmatrix} a & b & c \end{bmatrix}, \quad tB = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix},$$

```
In [1]: import sympy as sy

In [2]: a, b, c = sy.symbols('a b c')

In [3]: tA = sy.Array([a, b, c])

In [4]: tA.shape
Out[4]: (3,)
```

```
In [5]: tB = sy.Array([1, 2, 3, 4]) # shape (3,)

In [6]: tAB = sy.tensorproduct(tA, tB) # shape (3,4)

in [7]: tAB
Out[7]: [[a, 2*a, 3*a, 4*a], [b, 2*b, 3*b, 4*b], [c, 2*c, 3*c, 4*c]]
```

$$tA \otimes tB = \begin{bmatrix} a & 2a & 3a & 4a \\ b & 2b & 3b & 4b \\ c & 2c & 3c & 4c \end{bmatrix}$$

```
In [8]: tC = sy.Array(range(12), (4, 3)) # shape (4,3)
```

$$tC = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix}$$

```
In [9]: tD = sy.tensorproduct(tAB, tC)

In [10]: tD.shape
Out[10]: (3, 4, 4, 3)
```

これら SymPy 計算で実行されるテンソル計算はその shape を観察してわかるように NumPy 配列のテンソル演算 `np.tensordot(na. nb, axes=0)` と同様です。SymPy 多次元配列（テンソル）のスライスについても同じです。

`sy.tensorcontractio(t,(j,k,...))` は n 次元 SymPy テンソル $t_{i_0,\ldots,i_{n-1}}$ の軸番号 $(j,k,\ldots)$ に関する総和積を取って縮約計算をします。

```
In [11]: tE = sy.tensorcontraction(tD, (1, 2)) # shape (3, 3)
```

$$tE = \begin{bmatrix} 60a & 70a & 80a \\ 60b & 70b & 80b \\ 60c & 70c & 80c \end{bmatrix}$$

```
In [12]: m = sy.Matrix(3,2, range(6))
```

$$M = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

```
In [13]: sy.Matrix(tE) * M # shape (3,2)
Out[13]:
Matrix([
[460*a, 670*a],
[460*b, 670*b],
[460*c, 670*c]])
```

$$tEM = \begin{bmatrix} 460a & 670a \\ 460b & 670b \\ 460c & 670c \end{bmatrix}.$$

2.4.3 SymPy 行列と NumPy 配列の相互変換

NumPy 配列と SymPy 行列との相互変換について検討しておきましょう。NumPy および SymPy で生成される配列の shape 値を観察すると、クラス Array を使う SymPy テンソルはクラス array の NumPy 配列の取り扱いとの平行性が認められます（節 2.4.2.5 参照)。

2.4.3.1 NumPy から SymPy 配列を構成 `ndarray.tolist`

NumPy 配列から SymPy 配列を構成するためにはメソッド `ndarray.tolist()` をつかってまず数リストに変換します。NumPy メソッド

```
nA.tolist()
```

NumPy 配列 `nA` からその数リストを返します（リストは NumPy 配列と同じレベルの深さを持ちます)。NumPy では、数リスト `plist` からコンストラクタ array を使って NumPy 配列を生成しました。`ndarray.tolist()` は逆に NumPy 配列から数リストを返します。なお、NumPy スカラー `np.array(n)` についてはスカラー値 n を返します。

一方、SymPy の数配列は NumPy 配列から得られた数リストを使って生成できますが、同じ数リストでも SymPy クラス Matrix と Array では shape が異なる場合があります。

```
In [1]: import numpy as np

In [2]: import sympy as sy

In [3]: na = np.arange(2*3).reshape(2,3)  # shape (2,3)

In [4]: nb = np.arange(2*3*4).reshape(2,3,4) # shape (2,3,4)

In [5]: alist = na.tolist()

In [6]: blist = nb.tolist()
```

```
In [7]: sa = sy.Matrix(alist) # SymPy 行列 # shape (2, 3)

In [7]: sa
Out[7]:
Matrix([
[0, 1, 2],
[3, 4, 5]])

In [8]: sb = sy.Array(blist) # shape (2, 3, 4) の SymPy テンソル

In [9]: sy.Matrix(blist).shape # sb.shape と違う
Out[9]: (2, 3)
```

この例のように NumPy から得たリストから SymPy 配列が得られますが、クラス Matrix の `sy.Matrix(blist)` の shape (2,3) は `sy.Array(blist)` の shape (2,3,4) と異なります。

クラス Matrix の SymPy 行列はあくまでも行列的（2 次元配列）であり、その運用において多次元 NumPy 配列との微妙な差異が生じることがあります。

2.4.3.2 SymPy 配列から NumPy 配列 `matrix2numpy` `list2numpy`

SymPy 関数

```
sympy.matrix2numpy(A, dtype='object'>)
```

は SymPy 行列 `A` を NumPy 配列に変換します。また、SymPy 関数

```
sympy.list2numpy(expr, dtype='object'>)
```

は SymPy 配列 `expr` の Python リストを NumPy 配列に変換します。

`matrix2numpy` では SymPy 行列と同じ shape の NumPy 配列を返し、`list2numpy` では SymPy 配列の shape 値をかけ合わせた要素総数の 1 次元 NumPy 配列を用意して SymP 配列の値を埋めていきます（余った箇所には None）。

```
In [10]: x, y = sy.symbols('x y')

In [11]: sa = sy.Matrix([[x, 1, 2], [y, 4, 5]])

In [12]: sa
Out[12]:
Matrix([
[x, 1, 2],
[y, 4, 5]])
```

```
In [13]: sy.matrix2numpy(sa)
Out[13]:
array([[x, 1, 2],
       [y, 4, 5]], dtype=object)

In [14]: sy.matrix2numpy(sa.subs([(x,0),(y,3)]), dtype=float)
Out[14]:
array([[0., 1., 2.],
       [3., 4., 5.]])

In [15]: blist = [[[0, 1, 2, 3],[4, 5, 6, 7],[8, 9, 10, 11]],
         [[12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 23]]]

In [16]: sy.Matrix(blist)
Out[16]
Matrix([
[    [0, 1, 2, 3],     [4, 5, 6, 7],   [8, 9, 10, 11]],
[[12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 23]]])

In [17]: sy.matrix2numpy(sy.Matrix(blist))
Out[17]:
array([[list([0, 1, 2, 3]), list([4, 5, 6, 7]), list([8, 9, 10, 11])],
       [list([12, 13, 14, 15]), list([16, 17, 18, 19]),
        list([20, 21, 22, 23])]], dtype=object)

In [18]: sy.list2numpy(sa)
Out[18]: array([0, 1, 2, 3, 4, 5], dtype=object)

In [19]: sy.list2numpy(sb)
Out[19]:
array([[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]],
       [[12, 13, 14, 15], [16, 17, 18, 19], [20, 21, 22, 23]], None, None,
       None, None, None, None, None, None, None, None, None, None,
       None, None, None, None, None, None, None, None, None], dtype=object)

In [20]: sy.list2numpy(sy.Matrix(blist))
Out[20]:
array([list([0, 1, 2, 3]), list([4, 5, 6, 7]), list([8, 9, 10, 11]),
       list([12, 13, 14, 15]), list([16, 17, 18, 19]),
       list([20, 21, 22, 23])], dtype=object)
```

2.4.4 SymPy で方程式の解を求める

SymPy を使って方程式の解を求める方法として `sympy.linsolve`、`sympy.solveset` や `sympy.nonlinsolve` などが用意されています。

2.4.4.1 線形連立方程式を解く `linsolve`

SymPy 関数

```
sympy.linsolve((A, b), x0,x1,x2...)
```

は、SumPy を使って $n \times m$ 定数行列 A と m 次列ベクトル $\boldsymbol{b}$ に関する非斉次線形連立方程式系 $A\boldsymbol{x} = \boldsymbol{b}$ の解 $\boldsymbol{x} = (x^0, x^1, \ldots, x^{n-1})$ を計算します。

$$A\boldsymbol{x} = \boldsymbol{b} \Leftrightarrow \begin{bmatrix} {A^0}_0 & \cdots & {A^0}_{m-1} \\ \vdots & \ddots & \vdots \\ {A^{n-1}}_0 & \cdots & {A^{n-1}}_{m-1} \end{bmatrix} \begin{bmatrix} x^0 \\ \vdots \\ x^{n-1} \end{bmatrix} = \begin{bmatrix} b^0 \\ \vdots \\ b^{n-1} \end{bmatrix}$$

不定方程式系（変数の数 m よりも方程式の数 n が少ない $n < m$ とき）や過剰決定系（方程式の数 n が変数の数 m より大きい $n > m$ とき）の両方をサポートしています。解集合は値の並びのタプルとして有限集合返されます。不定方程式系 ($n < m$) では、解は与えられた変数によってパラメータ化されます。解がない場合には ValueError を返します。

$n \times (m + 1)$ 行列 $(A, \boldsymbol{b})$（n 行 m 列の A の右側に列ベクトル $\boldsymbol{b}$ を並べて得られる）を方程式系のシステムと称します。3 次正則行列 `A`、と 3 次元ベクトル `b` を SymPy 列ベクトルとして与えて `sy.linsolve()` を使って解いてみましょう。次の例では解は一意に決まります（解は 1 つのタプルからなる集合で与えられます)。

$$A\boldsymbol{x} = \boldsymbol{b} \Leftrightarrow \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 1 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 14 \\ -28 \\ 7 \end{bmatrix}, \qquad \boldsymbol{x} = \begin{bmatrix} -9 \\ -26 \\ 25 \end{bmatrix}$$

```
In [1]: import sympy as sy

In [2]: x, y, z = sy.symbols('x y z')

In [3]: A = sy.Matrix([[1,2,3], [3,1,1], [2,0,1]])

In [4]: b = sy.Matrix([14, -28, 7])

In [5]: sol_set = sy.linsolve((A, b), (x, y, z))
```

```
In [6]: sol_set
Out[6]: FiniteSet((-9, -26, 25))

In [7]: sx =  sy.Matrix(list(sol_set)).reshape(3,1) # 階集合を列ベクトル

In [8]:  A.multiply(sx) == b  # 検算
Out[8]: True
```

解 sol_set は 1 つのタプルからなる集合なので, list(sol_set) でリスト化した上で SymPy 行列とした上で、reshape して検算しています。

次の例では連立不定方程式の解を求めています。行列 B の階数が 2 であることから不定方程式系であることがわかります。実際、開集合は sy.linsolve の解は z でパラメータ化されています。

$$B\boldsymbol{x} = \boldsymbol{c} \Leftrightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix}, \qquad \boldsymbol{x} = \begin{bmatrix} z-1 \\ 2-2z \\ z \end{bmatrix}$$

```
In [9]: B = sy.Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [10]: B.rank()
Out[10]: 2

In [11]: c = sy.Matrix([3, 6, 9])

In [12]: sy.linsolve((B, c), x, y, z)
Out[12]: FiniteSet((z - 1, 2 - 2*z, z))
```

2.4.4.2 一般の方程式 solveset nonlinsolve

sympy.linsolve 以外にも方程式を解くために sympy.solveset や sympy.nonlinsolve などが用意されています。

SymPy 関数

```
sympy.solveset(eq, vars, domain)
```

は、方程式 eq(= 0) を変数 var に関して、指定した領域範囲（デフォルト sympy.Complexes または sympy.Reals）で解いた集合を返します。解を持たない場合は空集合 EmptySet を返します。もちろん、いつでも解が求まるわけではありませ

ん。また、式は代数方程式であるとは限りません。

$x^3 + 1 = 0, \qquad x = -1$実数の範囲.

```
In [13]: sy.solveset(x ** 3 + 1, x, domain=sy.Reals)
Out[13]: FiniteSet(-1)
```

$x^3 + 1 = 0, \qquad x = -1, \frac{1}{2} \pm \frac{i\sqrt{3}}{2}.$

```
In [14]: sy.solveset(x ** 3 + 1, x)
Out[14]: FiniteSet(-1, 1/2 - sqrt(3)*I/2, 1/2 + sqrt(3)*I/2)
```

$x^2 + y^2 = 0, \qquad x = -iy, iy.$

```
In [15]: sy.solve(x ** 2 + y ** 2, x)
Out[15]: [-I*y, I*y]
```

$\sin x - 1 = 0.$

```
In [16]: sy.solveset(sy.sin(x) - 1, x)
Out[16]: ImageSet(Lambda(_n, 2*_n*pi + pi/2), Integers)
```

最後の結果は

$$\left\{2\pi n + \frac{\pi}{2} \,|\, n \in \mathbb{Z}\right\}$$

を表しています。

また、閉じた結果として解を与えない場合もあります。

$x^5 + x^2 - 1 = 0.$

```
In [17]: sy.solveset(x ** 5 + x ** 2 - 1, x)
Out[17]: FiniteSet(CRootOf(x**5 + x**2 - 1, 0), CRootOf(x**5 + x**2 - 1, 1),
CRootOf(x**5 + x**2 - 1, 2),CRootOf(x**5 + x**2 - 1, 3), CRootOf(x**5 + x**2 - 1, 4))
```

ここで、`CRootOf(x**5 + x**2 - 1,n)` は解くべき対象の $x^5 + x^2 - 1 = 0$ の n 番目の複素根を表しています。したがって、この場合は解を与えているわけではありません。

SymPy 関数

```
sympy.nonlinsolve(eqs, vars)
```

は、リスト `eqs` で与えた非線形連立方程式をリストで指定した変数 `vars` について解いた結果を集合として返します。以下の方程式では解が自明な場合です。

$$\begin{cases} x^2 + ax = 0 \\ x - y = 0 \end{cases}, \qquad (x, y) = (0, 0), (-a, -a))$$

```
In [18]: a = sy.symbols('a', real=True) # a は実に設定

In [19]: sy.nonlinsolve([x**2 + a*x, x - y], [x, y])
Out[19]: FiniteSet((0, 0), (-a, -a))
```

$$\begin{cases} x^2+1=0 \\ y^2+1=0 \end{cases}, \qquad (x,y)=(i,i),(-i,i),(i,-i),(-i,-i).$$

```
In [20]: sy.nonlinsolve([x**2 + 1, y**2 + 1], [x, y])
Out[20]: FiniteSet((I, I), (-I, I), (I, -I), (-I, -I))
```

$$\begin{cases} \mathrm{e}^x-\sin y=0 \\ 1/y-3=0. \end{cases}.$$

```
In [21]: sy.nonlinsolve([sy.exp(x) - sy.sin(y), 1/y - 3], [x, y])
Out[21]: FiniteSet((ImageSet(Lambda(_n, 2*_n*I*pi + log(sin(1/3))), Integers), 1/3))
```

最後の結果は $\{(x,y)\}$ として

$$\left(i2\pi n+\log\left(\sin\frac{1}{3}\right),\frac{1}{3}\right),\quad n\in\mathbb{Z}$$

からなる集合を表します。

2.4.5 行列の微分と多変数関数のヘッセ行列

2.4.5.1 SymPy 行列の微分

SymPy の微分（偏微分）操作 `diff` は、SymPy 配列の各要素に対して同時に適用されます。

次の例は、関数 f,g,h,k から定義した変数 x,y を持つ関数行列 A（の各要素）に対して x さらに y について連続偏微分した $\partial_{xy}A = A_{x,y}$ を計算した上で、$f(x,y)=\sin xy, k(x,y)=\exp xy$ を代入した場合の結果 `result` を計算します。

```
In [1]: import sympy as sy

In [2]:x, y = sy.symbols('x y')

In [3]: f,g,h,k = sy.symbols('f g h k', cls=sy.Function)

In [4]: A = sy.Matrix([[f(x,y),g(x,y)], [h(x,y),k(x,y)]])
```

$$A=\begin{bmatrix} f(x,y) & g(x,y) \\ h(x,y) & k(x,y) \end{bmatrix}.$$

```
In [5]: Axy = A.diff(x, y)

In [6]: Axy
Out[6]:
Matrix([
[Derivative(f(x, y), x, y), Derivative(g(x, y), x, y)],
[Derivative(h(x, y), x, y), Derivative(k(x, y), x, y)]])

In [7]: result = Axy.subs([(f, sy.sin(x*y)), (k, sy.exp(x*y))])

In [8]: result.doit()
Out[8]:
Matrix([
[ -x*y*sin(x*y) + cos(x*y), Derivative(g(x, y), x, y)],
[Derivative(h(x, y), x, y),         (x*y + 1)*exp(x*y)]])
```

$$\text{result} = \begin{bmatrix} -xy\sin xy + \cos xy & g_{xy}(x,y) \\ h_{xy}(x,y) & (xy+1)\mathrm{e}^{xy} \end{bmatrix}$$

2.4.5.2 ヘッセ行列 `hessian`

多変数実関数 $f(x_0, \ldots, x_{n-1})$ が値 $\boldsymbol{a} = (a_0, \ldots, a_{n-1})$ で**極値**であるとは、a での各変数についての偏微分 f_{x_i} の値がすべて 0

$$f_{x_i}(\boldsymbol{a}) \equiv \frac{\partial f}{\partial x_i}(\boldsymbol{a}) = 0, \quad \mathbf{0} \leqq i \leqq n-1$$

になっているときです。極値が極大または極小であるかを判断するためには、1 変数関数の場合と同様に 2 階微分の情報が必要になります。

n-変数実関数 $f(x_0, \ldots, x_{n-1})$ に対して、次で定義される n 次正方実行列関数を**ヘッセ行列** (Hessian) と呼んでいて（$f_{x_j,x_k} = f_{x_k,x_j}$ のときは対称行列)、多変数実関数の極値判定で使われます。

$$H(f) = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_0^2} & \dfrac{\partial^2 f}{\partial x_0 \partial x_1} & \cdots & \dfrac{\partial^2 f}{\partial x_0 \partial x_{n-1}} \\ \dfrac{\partial^2 f}{\partial x_1 \partial x_0} & \dfrac{\partial^2 f}{\partial x_1^2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 \partial x_{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_{n-1} \partial x_0} & \dfrac{\partial^2 f}{\partial x_{n-1} \partial x_1} & \cdots & \dfrac{\partial^2 f}{\partial x_{n-1}^2} \end{bmatrix} \tag{2.1}$$

2 変数関数 $f(x, y)$ の場合には、極値 (a, b) は $f_{xx}(a, b) > 0$ かつ $\det H(f)(a, b) > 0$ のとき極大、$f_{xx}(a, b) < 0$ かつ $\det H(f)(a, b) > 0$ のとき極小、$\det H(f)(a, b) < 0$ では極大でも極

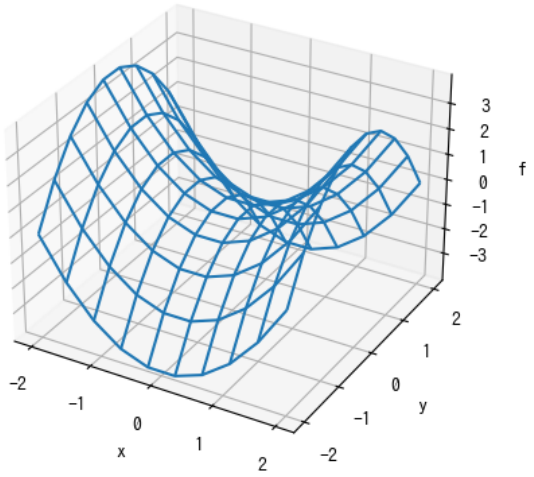


図 2.3　関数 $f(x,y)=x^2-y^2$ は原点 $(,0)$ で停留点を持つ。x-軸方向に $f(x,0)=x^2$ で f は極小であり、y-軸方向に $f(0,y)=-y^2$ で極大であるような鞍点になっている。ヘッセ行列 $H(f)$ の行列式 $\det H(f)(0,0)=-4$.

小でもない**停留点**であることが知られています。実際、$f(x,y)=x^2-y^2$ に対して原点 $(0,0)$ は鞍点で極大でも極小でもありません（図**??**）。

SymPy 関数

`sympy.hessian(f,`$[x_0,\ldots,x_{n-1}]$`)`

は、多変数実関数 f の変数 $x_0,\ldots,x_{n-1}$ に関する Hesse 行列 $H(f)$ を計算します。

$$f(x,y)=x^2-y^2,\quad H(f)=\begin{bmatrix}2 & 0\\ 0 & -2\end{bmatrix}.$$

```
In [9]: x, y = sy.symbols('x y')

In [10]: f = x ** 2 - y ** 2

In [11]: sy.hessian(f, [x,y])
Out[11]:
Matrix([
[2,  0],
[0, -2]])
```

ヘッセ行列を使う例として、直方体の 3 辺の長さの合計を固定したときにその体積を最大にする辺の長さを求めてみます。いま、直行する方向に沿った辺の長さを x,y,z とする直方体について、その辺の長さの合計が一定値 $a>0$ であるときの体積 $V(x,y)\equiv xyz=xy(a-x-y)$ を最大にする x,y を考えます。

x および y 方向に偏微分して $V_x(x^*, y^*) = 0$, $V_y(x^*, y^*) = 0$ を満たす極値 (x^*, y^*) の集合 sols

$$\text{sols} = \left\{ (x^*, y^*) \,|\, \frac{\partial V}{\partial x}(x^*, y^*) = 0, \frac{\partial V}{\partial y}(x^*, y^*) = 0 \right\}$$

を `sympy.nonlinsolve` で求めてみます（節 2.4.4.2）が得られます。すると sols として $(0,0), (0,a), (a,0), (a/3, a/3)$ の 4 つがあることがわかります。

```
In [12]: x, y, a = sy.symbols(x'x y a')

In [13]: V = x * y * (a-x-y)

In [14]: dxV = sy.diff(V, x)

In [15]: dyV = sy.diff(V, y)

In [16]: sols = sy.nonlinsolve([dxV, dyV], (x,y)) #極値の計算

In [17]: sols
Out[17]: FiniteSet((0, 0), (0, a), (a, 0), (a/3, a/3))
```

そこで、$V(x,y)$ のヘッセ行列 $H(f)$ における $(0,0)$-成分の極値 $(a/3, a/3)$ での値、おそび行列式 $\det H(f)$ の極値 $(a/3, a/3)$ での値を確認します。

```
In [18]: HV = sy.hessian(V, [x,y]) # ヘッセ行列

In [19]: # ヘッセ行列 [0,0] 成分の極値 (a/3,a/3) での評価（負値）
         HV[0,0].subs([(x,a/3),(y,a/3)])
Out[19]: -2*a/3

In [20]: HVDet = sy.det(HV) # ヘッセ行列の行列式

In [21]: HVDet
Out[21]: -a**2 + 4*a*x + 4*a*y - 4*x**2 - 4*x*y - 4*y**2

In [22]: HVDet.subs([(x, a/3), (y,a/3)]) # 極値 (a/3,a/3) での評価（正値）
Out[22]: a**2/3
```

$H(V)$ の $(0,0)$-成分の極値 $(a/3, a/3)$ での値が負 $H(f)[0,0](a/3, a/3) < 0$、またそこでの行列式の値が正 $\det H(f)(a/3, a/3) > 0$ であることが確認できました。これより、極値 $(a/3, a/3)$ は極大点、つまり辺の長さを $x = a/3, y = a/3(, z = a/3)$ とした立方体の体積が最大になることがわかりました。

2.4.6 SymPy 表式を NumPy 関数に変換する `lambdify`

SymPy 関数 `lambdify(args, expr)` を使って SymPy 表式 `expr` を引数 `args` の**ラムダ関数** (lambda function) に変換し、指定した数値計算モジュールでの計算に利用できるようになります（デフォルトは `modules="numpy"`）。

代入メソッド `.subs`（節 2.4.1.1）や浮動小数表示 `.evalf`（節 2.4.2.4）は SymPy 表式を特定の参照点で評価できますが、多数の点の値を求めることは現実的ではありません。`lambdify` を使うと、SymPy 関数を NumPy（や SciPy または Tensorflow）の効率よくベクトル計算可能な数値関数として直接利用できるようになります。

```
In [1]: import sympy as sy

In [2]: import numpy as np

In [3]: x, y = sy.symbols('x y')

In [4]: f = sy.lambdify(x, sy.sin(x), modules="numpy") # NumPy 関数化

In [5]: xlist = np.linspace(0, np.pi, 10)

In [6]: xlist
Out[6]:
array([0.        , 0.34906585, 0.6981317 , 1.04719755, 1.3962634 ,
       1.74532925, 2.0943951 , 2.44346095, 2.7925268 , 3.14159265])

In [7]: f(xlist)
Out[7]:
array([0.00000000e+00, 3.42020143e-01, 6.42787610e-01, 8.66025404e-01,
       9.84807753e-01, 9.84807753e-01, 8.66025404e-01, 6.42787610e-01,
       3.42020143e-01, 1.22464680e-16])
```

コード**??** `sympy_saddle_function.py` は、鞍関数 $f(x,y) = x^2 - y^2$ を SymPy 関数で定義して NumPy 関数に変換した上で、x, y-平面上の値を節 1.4.3 で紹介したプログラム 1.4-4 と同じようにして関数のワイヤーフレームを描きます（図 2.3）。確かに原点が鞍点（x-方向に極小、y-方向に極大）であることがわかります。

コード 2.4-1　sympy_saddle_function

```
import numpy as np
import sympy as sy
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

```
x, y = sy.symbols('x y')
saddle = x ** 2 - y ** 2
npsaddle = sy.lambdify([x,y], saddle, modules="numpy") # NumPy関数化

xnum, ynum = 10, 10
xpoint = np.linspace(-2, 2, num=xnum)
ypoint = np.linspace(-2, 2, num=ynum)
xg, yg = np.meshgrid(xpoint, ypoint)
zg = npsaddle(xg, yg)

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_wireframe(xg, yg, zg)
ax.set(xlabel='x', ylabel='y', zlabel='f')
#fig.savefig('sympy_saddle_function.png')
fig.show()
```

2.5 固有値と固有ベクトル

SymPy と NumPy を並行利用して行列の固有値および固有ベクトルの計算を紹介します。

正方行列 A に対して、ある非零ベクトル $\boldsymbol{x}$ とスカラー λ が存在して

$$A\boldsymbol{x} = \lambda\boldsymbol{x}$$

とできるときに、λ を**固有値**、$\boldsymbol{x}$ を λ に属する（右）**固有ベクトル**といいます。固有ベクトルは、行列 $\boldsymbol{A}$ の固有値がその**特性多項式** (characteristic polynomial)

$$\det(\boldsymbol{x}I - A) = 0$$

である n-次方程式の解となっています（I は単位行列)。

2.5.1 固有多項式 `charpoly numpy.poly`

正方行列 A の固有多項式に関わるものとして、SymPy には特性多項式を与える `A.charpoly`、NumPy には特性多項式の係数を与える `A.poly` があります。

SymPy の Matrix クラスのメソッド

```
.charpoly(x)
```

は、$n \times n$ の SymPyy 配列 A に対する特性多項式 $\det(xI - A)$ を返します。引数 x を省略したときのデフォルトシンボルは lambda です[6)]。

$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ の特性多項式 $\lambda^2 - (a+d)\lambda + ad - bc$

```
In [1]: import sympy as sy

In [2]: a,b,c,d = sy.symbols('a b c d')

In [3]: A = sy.Matrix([[a,b], [c,d]])

In [4]: Achar = A.charpoly('lamda')

In [5]: Achar
Out[5]: PurePoly(lamda**2 + (-a - d)*lamda + a*d - b*c, lamda, domain='ZZ[a,b,c,d]')
```

NumPy 関数

```
numpy.poly(npA)
```

は、NumPy 行列 npA の特性多項式の係数の並びを最高次 n から 0 次までの長さ $n+1$ の 1 次元配列を返します。SymPy 行列 A に $a = 1, b = 2, c = 3, d = 4$ を代入して NumPy 配列に変換して確かめます。

$\mathrm{npA} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ の固有多項式の係数 $1.0, -5.0, -2.0$

```
In [5]: npA = sy.matrix2numpy(A.subs([(a,1), (b,2), (c,3), (d,4)]), dtype='float64')

In [6]: npA
Out[6]:
array([[1., 2.],
       [3., 4.]])

In [7]: np.poly(npA)
Out[7]: array([ 1., -5., -2.])
```

6) lambda は Python の予約語です。SymPy および Python で λ として生成するには 'b' 抜きの lamda を使います。

ここでは、A に値を代入した後に指定したデータ型を持つ NumPy 配列に変換するためにメソッド.matrix2numpy を使いました（節 2.4.3.2）。

2.5.2 固有値と固有ベクトルを求める

行列の固有値とその固有ベクトルをぞれぞれ NumPy と SymPy をつかって計算計算することができます。SymPy を使う計算から紹介した方が見通しが良くなります。

2.5.3 SymPy を使う固有値と固有ベクトル eigenvals eigenvects

SymPy の Matrix クラス A のメソッド

```
A.eigenvals()
```

は、SymPy 行列の固有値 λ_i とその重複度 $\mathrm{mul}(\lambda_i)$ をコロン：で組 $\lambda_i : \mathrm{mul}(\lambda_i)$ としたディクショナリ型

$$\mathrm{A.eigenvals}() = \{\lambda_0 : \mathrm{mul}(\lambda_0),\ \lambda_1 : \mathrm{mul}(\lambda_1), \dots\}$$

を返します。固有値と重複度の順に注意します。

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 0 & 3 & -2 \\ 0 & 2 & -2 \end{bmatrix} \quad \text{の固有値 } \lambda = 2, 1, -1.$$

行列 A の 3 つの固有値はすべて重複度 1 で $2, 1, -1$ です。

```
In [1]: import sympy as sy

In [2]: A = sy.Matrix([[1, 2, -1], [0, 3, -2], [0, 2, -2]])

In [3]: evalA = A.eigenvals()

In [4]: evalA
Out[4]: {2: 1, 1: 1, -1: 1}
```

$$B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix} \quad \text{の固有値 } \lambda = 1, 1, 3.$$

固有値が自明な対角行列 B では、固有値 1 の重複度は 2，固有値 3 の重複度は 1 であることを確かめます。

```
In [5]: B = sy.diag(1, 1, 3)

In [6]: evalB = B.eigenvals()

In [7]: evalB
Out[7]: {1: 2, 3: 1}
```

SymPy Matrix クラス A のメソッド

```
A.eigenvects()
```

は、SymPy 行列の固有値 λ_i とその重複度 $\mathrm{mul}(\lambda_i)$ と固有ベクトル $v_i^0, \ldots, v_i^{\mathrm{mul}(\lambda_i)-1}$ から成るタプル $(\lambda_i, \mathrm{mul}(\lambda_i), [v_i^0, \ldots, v_i^{\mathrm{mul}(\lambda_i)-1}])$ を並べたリスト

$$\mathrm{A.eigenvects}() = \left[(\lambda_0, \mathrm{mul}(\lambda_0), [v_0^0, \ldots, v_0^{\mathrm{mul}(\lambda_0)-1}]),\ (\lambda_1, \mathrm{mul}(\lambda_1), [v_1^0, \ldots, v_i^{\mathrm{mul}(\lambda_1)-1}]), \ldots\right]$$

を返します。重複度の数だけ対応する固有ベクトルが並びます。SymPy の固有ベクトルは shape(n,1) を持つ $n \times 1$ の 2 次元列配列（線形代数的には列ベクトル）で与えられることに注意してください。

$A = \begin{bmatrix} 1 & 2 & -1 \\ 0 & 3 & -2 \\ 0 & 2 & -2 \end{bmatrix}$ について次のことを確かめます（カッコ内は固有値 λ_i の重複度 $\mathrm{mul}(\lambda_i)$）。

$$\lambda_0 = -1\,(1),\ v_0 = \begin{bmatrix} 0 \\ 1/2 \\ 1 \end{bmatrix};\quad \lambda_1 = 1\,(1),\ v_1 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix};\quad \lambda_2 = 2\,(1),\ v_2 = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

```
In [8]: evectA = A.eigenvects()

In [9]: evectA
Out[9]:
[(-1,
  1,
  [Matrix([
   [  0],
   [1/2],
   [  1]])]),
 (1,
  1,
  [Matrix([
   [1],
```

```
   [0],
   [0]])]),
 (2,
  1,
  [Matrix([
   [3],
   [2],
   [1]])])]

In [10]: evectA[2][2][0]
Out[10]:
Matrix([
[3],
[2],
[1]])
```

SymPy の固有ベクトルは規格化されていないことがわかりました。i 番目の固有値が重複度 1 の場合の固有ベクトルは `evectA[i-1][2][0]` のように取り出せます。

重複度 2 以上の固有値を持つ場合には次のようにやや込み入った結果になります（NumPy 計算との比較は節 2.5.3.2）。

$B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 3 \end{bmatrix}$ について次のことを確かめます（カッコ内は固有値 λ_i の重複度 $\mathrm{mul}(\lambda_i)$）。

$$\lambda_0 = 1\,(2),\ v_0^0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, v_0^1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix};\quad \lambda_1 = 3\,(1),\ v_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

```
In [11]: evectB = B.eigenvects()

In [12]: evectB
Out[12]:
[(1,
  2,
  [Matrix([
   [1],
   [0],
   [0]]),
   Matrix([
   [0],
   [1],
```

```
   [0]])]),
 (3,
  1,
  [Matrix([
   [0],
   [0],
   [1]])])]

In [13]: evectB[0][2][1]
Out[13]:
Matrix([
[0],
[1],
[0]])
```

確かに行列 B の重複度 2 の固有値 1 に対応する列ベクトルがリストに 2 つ現れています。i 番目の固有値の重複度が $k \geqq 2$ の場合、その j 番目（$0 \leqq j \leqq k-1$）の固有ベクトルは `evectB[i-1][2][j]` のように取り出せます。

2.5.3.1 SymPy を使う固有値計算の注意

SymPy を使った行列の固有値計算は手計算のように見通しのよい結果を与えてくれますが、とくに SymPy シンボルや有理数計算を実行する場合には多くの計算資源が必要です。

```
In [14]: a = sy.symbols('a')

In [15]: A = sy.Matrix([[a, 1], [-1, 0]])

In [16]: %timeit A.eigenvals()
2.78 ms ś 26.1 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

In [17]: %timeit A.eigenvects()
3.11 ms ś 67.6 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

In [18]: C = sy.Matrix([[a,1,1], [-1,0,0], [1,2,0]])

In [19]: %timeit C.eigenvals()
2.63 ms ś 21.4 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

SymPy シンボルを含む 3 次以上の有理計算は高々固有値計算がせいぜいで、固有ベクトルの計算は現実的ではありません。事実、この例の行列 C の固有ベクトルの計算には

$x^3 - ax^2 + 2$ の根が必要になり、通常は数分以上かかります。

SymPy で計算した A の固有ベクトルを `print(sy.latex(A.eigenvects()))` で書き出すと次のようになります。

$$A \text{ の固有ベクトル} = \left[\left(\frac{a}{2} - \frac{\sqrt{(a-2)(a+2)}}{2},\ 1,\ \begin{bmatrix} -\frac{a}{2} + \frac{\sqrt{a^2-4}}{2} \\ 1 \end{bmatrix} \right), \left(\frac{a}{2} + \frac{\sqrt{(a-2)(a+2)}}{2},\ 1,\ \begin{bmatrix} -\frac{a}{2} - \frac{\sqrt{a^2-4}}{2} \\ 1 \end{bmatrix} \right) \right]$$

A の 2 つの固有値 $\lambda_\pm = \frac{a}{2} \pm \frac{\sqrt{(a-2)(a+2)}}{2}$ は重複度 1 であることが確認でき、パラメータ a の関数 $\lambda_\pm(a)$ と見たとき固有値 $\{\lambda_+(a), \lambda_-(a)\}$ は $-2 < a < 2$ のとき虚数部分を持つ複素共役の関係になります。固有値のパラメータ変化によって固有値の性質が変化する様子を **分岐** (bifurcation) と言います。今の場合、虚部を持たない 2 つの実固有値が同じ実部を有する 1 組の複素固有値に分岐します。

プログラム 2.5-1 `eigen_value_plot.py` は、パラメータ a を持つ固有値 $\lambda_\pm(a) = a/2 \pm \sqrt{(a-2)(a+2)}/2$ を SymPy 関数として取り出して `sympy.lamdify` を使って NumPy 関数化 ev0 および ev1 とした上で、パラメータ区間 `alist` $= [-3, 0]$ について複素平面上の固有値の分岐を描きます。lamdify して利用する SymPy 関数が虚数計算できるように、NumPy 化した関数に渡す NumPy 配列 `alist` を dtype = np.complex64 と複素数だと型指定しておく必要があります。

図 2.4 は、コード 2.5-1 から得られるプロット図です。$\lambda_+(a)$ の各点を赤の実線で、$\lambda_-(a)$ の各点を青の破線でつないだ折れ線としてプロットしています。区間 $[-3, 0]$ を $num = 50$ 等分と粗く設定したために $a = 1$ での固有値の分岐点に間隙が生じています。$num \sim 70$ 程度にすると 1 点から分岐するように描くことができます。

コード 2.5-1　eigen_value_plot

```
%matplotlib inline
import numpy as np
import sympy as sy
import matplotlib.pyplot as plt

a = sy.symbols('a')
A = sy.Matrix([[a, 1], [-1, 0]])
evdic = A.eigenvals() # Aの固有値
evlist = [ei for ei in evdic.keys()] # 固有値リスト
```

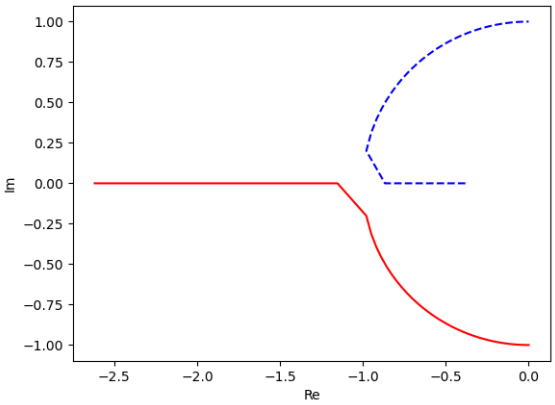


図 2.4　パラメータ a を持つ行列 $\begin{bmatrix} a & 1 \\ -1 & 0 \end{bmatrix}$ の 2 つの固有値 $\lambda_{\pm}(a)$ のの実部と虚部をパラメータ区間 $[-3, 0]$ を 50 等分して折れ線でプロット。分割個数（$num = 50$）が少ないために、分岐曲線の間に隙間が空いている。$num \sim 70$ 程度にすると 1 点から分岐するように描くことができる。

```
# evlist = [a/2 - sqrt((a - 2)*(a + 2))/2, a/2 + sqrt((a - 2)*(a + 2))/2]
eigfunc0 = sy.lambdify(a, evlist[0], modules="numpy") # 0番目固有値関数
eigfunc1 = sy.lambdify(a, evlist[1], modules="numpy") # 1番目固有値関数
an = 70 # パラメータの分割点の数
alist = np.linspace(-3,0, num=an, dtype=np.complex64) # 複素型指定

fig = plt.figure()
ax = plt.axes()
ax.plot(np.real(eigfunc0(alist)), np.imag(eigfunc0(alist)), 'r')
ax.plot(np.real(eigfunc1(alist)), np.imag(eigfunc1(alist)), 'b', linestyle="
    dashed")
ax.set(xlabel='Re', ylabel='Im')
# plt.show()
```

2.5.3.2　NumPy を使う固有値、固有ベクトル `linalg.eig　linalg.eigvals`

NumPy 関数

```
eval,evec = np.linalg.eig(A)
```

は、NumPy 配列（行列）`A` の固有値の並びを 1 次元配列 `eval`、および対応する固有ベクトル（1 次元配列）を要素として並べた 2 次元 NumPy 配列 `evec` からなるタプルを計算します。

SymPy 計算のように固有値ごとにその重複度を返す（節 2.5.3）のではなく、NumPy 計

算では eval ではその重複度の個数分だけの同じ固有値が並び、evec はそれらに応じて固有ベクトルが並んだ配列となっています。

i 番目の固有値に対応する固有ベクトルは evec を列についてスライスした $\mathrm{evec}[:, i]$ で、数値的に次の関係にあります。

$$A \mathbin{@} \mathrm{evec}[:, i] \approx \mathrm{eval}[i] * \mathrm{evec}[:, i]$$

NumPy が返す各ベクトル eval[i] は長さ 1 に**規格化**されていることに注意してください。

一方、NumPy 関数 np.linalg.eigvals(A) は、A の固有値の並びだけからなる 1 次元 NumPy 配列を返し、固有ベクトルは返しません。

次の計算は、対角成分が 2 つ以上同じ要素を持つ NumPy 対角行列（対角成分が固有値となるので、したがって対応する固有値の重複度は 1 より大きい）の例です。重複度 2 の固有値 1 は重複度の数だけ並んでいます。

```
In [20]: nB = np.diag([1, 1, 3])

In [21]: eval, evec = np.linalg.eig(nB)

In [22]: eval
Out[22]: array([1., 1., 3.])

In [23]: evec
Out[24]:
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])

In [24]: np.linalg.eigvals(nB)
Out[24]: array([1., 1., 3.])
```

NumPy で計算された固有ベクトルは規格化され浮動小数点で計算されるため結果の見通しはよくありませんが、計算は高速に実行されます。

```
In [25]:  nA = np.array([[1, 2, -1], [0, 3, -2], [0, 2, -2]])

In [26]: np.linalg.eig(nA)
Out[26]:
(array([ 1.,  2., -1.]),
 array([[1.00000000e+00, 8.01783726e-01, 7.58594935e-18],
        [0.00000000e+00, 5.34522484e-01, 4.47213595e-01],
        [0.00000000e+00, 2.67261242e-01, 8.94427191e-01]]))
```

```
In [27]: %timeit np.linalg.eig(nA)
47.3 µs ± 3.74 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

In [28]: np.linalg.eigvals(nA)
Out[28]: array([ 1.,  2., -1.])

In [29]: %timeit np.linalg.eigvals(nA)
38.3 µs ± 3.59 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)


In [30]: A = sy.Matrix([[1, 2, -1], [0, 3, -2], [0, 2, -2]])

In [31]: %timeit A.eigenvects()
15.7 ms ± 923 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

繰り返しになりますが、NumPy 計算で得られるベクトル $\mathrm{eval}[i]$ は固有ベクトルではなく、2 次元配列 `evec` の各列に沿ってスライスした 1 次元配列 `evec[:, i]` が i 番目の固有値 `vec[i]` に対応する固有ベクトルです。

```
In [32]: nA @ evec[:,0] - eval[0] * evec[:,0]
Out[32]: array([0., 0., 0.])

In [33]: nA @ evec[:,1] - eval[1] * evec[:,1]
Out[33]: array([2.22044605e-16, 0.00000000e+00, 0.00000000e+00])

In [34]: nA @ evec[:,2] - eval[2] * evec[:,2]
Out[34]: array([7.58594935e-18, 5.55111512e-17, 0.00000000e+00])
```

2.6 行列の指数関数

n-変数の定係数 1 階線形微分方程式は、n-次正方定行列 $\boldsymbol{A}$ を使って次のように表されます（節 5.2.1 参照）。

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{A}\boldsymbol{x}(a), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \tag{2.2}$$

式 (2.2) の解は、1 変数微分方程式の $dx/dt = ax$ の解 $x(t) = \mathrm{e}^{a(t-t_0)}x_0$ と同じように、形式的に

$$\boldsymbol{x}(t) = \exp\left(\boldsymbol{A}(t-t_0)\right)\boldsymbol{x}_0 \tag{2.3}$$

で与えられます。ここで、行列の指数関数 $\exp(\boldsymbol{A}t)$ を

$$\exp(\boldsymbol{A}t) \equiv \sum_{k=0}^{\infty} \frac{\boldsymbol{A}^k t^k}{k!} \tag{2.4}$$

と定義しています。

2.6.1 行列指数関数 `scipy.linalg.expm`

$\boldsymbol{A}$ が対角行列のときのは、その指数関数は直ちに計算でき、次の形ににになります。

$$\boldsymbol{A} = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix}, \qquad \exp(\boldsymbol{A}t) = \begin{bmatrix} \mathrm{e}^{\lambda_1} & & & \\ & \mathrm{e}^{\lambda_2} & & \\ & & \ddots & \\ & & & \mathrm{e}^{\lambda_n} \end{bmatrix}.$$

$\boldsymbol{A}$ の指数関数 $\exp(\boldsymbol{A}t)$ を SciPy の関数 `scipy.linalg.expm()` を使って数値計算することができます。次は NumPy 正方配列 `a` の指数関数の計算例です。

```
In [1]: import numpy as np

In [2]: from scipy.linalg import expm

In [3]: a = np.array([[1, 2, 0], [0, 3, 1], [0, 0, -1]])

In [4]:expm(a)
Out[4]:
array([[ 2.71828183, 17.36725509,  3.75421318],
       [ 0.        , 20.08553692,  4.92941437],
       [ 0.        ,  0.        ,  0.36787944]])
```

整数配列であってもその要素は浮動少数値で表示されるため、いささかわかり難くなります。SymPy を使うと、SymPy 正方行列 `A` の指数関数を `A.exp()` によって計算できます。次の例は、Numpy 正方配列 `a` から SymPy 配列 `A` を定義し、`A.exp()` を計算しています（直接 SymPy 配列を `sympy.Matrix()` を使って与えても構いません)。

SymPy 記号 `E` は数 $\exp(1)$ を表します。

```
In [5]: import sympy as sym

In [6]: a = np.array([[1, 2, 0], [0, 3, 1], [0, 0, -1]])

In [7]: A = sym.Matrix(a)
```

```
In [8]: A
Out [8]:
Matrix([
[1, 2,  0],
[0, 3,  1],
[0, 0, -1]])

In [9]: A.exp()
Out[9]:
Matrix([
[E, -E + exp(3), -E/2 + exp(-1)/4 + exp(3)/4],
[0,      exp(3),       -exp(-1)/4 + exp(3)/4],
[0,           0,                    exp(-1)]])

In [10]: sym.matrix2numpy(A.exp()).astype(np.float)
Out[10]:
array([[ 2.71828183, 17.36725509,  3.75421318],
       [ 0.        , 20.08553692,  4.92941437],
       [ 0.        ,  0.        ,  0.36787944]])
```

最後の結果は、SymPy 指数関数配列 `A.exp()` を浮動小数点要素を持つ NumPy 配列に変換しています。

2.6.2 行列の Jordan 標準形 `.jordan_form`

一般的行列の指数関数の計算には、まずその行列の標準形を求め、標準形の指数関数を計算することで容易になります。行列 $\boldsymbol{A}$ で定まる正則行列 $\boldsymbol{P}$ によって、$\boldsymbol{A}$ は標準形 $\boldsymbol{J} = \boldsymbol{P}^{-1}\boldsymbol{A}\boldsymbol{P}$ に変換されます。行列の標準形を **Jordan 標準形**といいます。行列の指数関数の定義 (2.4) から

$$\boldsymbol{P}^{-1}\exp(\boldsymbol{A})\boldsymbol{P} = \exp(\boldsymbol{P}^{-1}\boldsymbol{A}\boldsymbol{P}) = \exp(\boldsymbol{J})$$

が成立することに注意すると、求める $\boldsymbol{A}$ の行列指数関数 $\exp(\boldsymbol{A})$ は次のように計算されます。

$$\exp(\boldsymbol{A}) = \boldsymbol{P}\exp(\boldsymbol{J})\boldsymbol{P}^{-1} \tag{2.5}$$

SymPy を使ってこれらを確かめてみましょう。SymPy では $\boldsymbol{A}$ の Jordan 標準化行列とその標準形を求めることができます。SymPy 正方行列 `A` に関してメソッド `.jordan_form()` は、標準化正則行列 `P` とその Jordan 行列 `J` の両方の並びを

```
P, J = A.jordan_form()
```

として返す。

逆行列をメソッド.inv() で求め、$\boldsymbol{P}^{-1}\exp(\boldsymbol{A})\boldsymbol{P}$ が実際に Jordan 行列になっていること、および $\boldsymbol{P}\exp(\boldsymbol{J})\boldsymbol{P}^{-1}$ が A.exp()= $\exp \boldsymbol{A}$ となっていることを確かめている。ただし、SynPy 配列では行列積の演算は記号*で行うことに注意する（NumPy 配列では要素ごとの積になる）。

```
In [11]: (P, J) = A.jordan_form()

In [12]: P
Out[12]:
Matrix([
[ 1/4, 1, 1],
[-1/4, 0, 1],
[   1, 0, 0]])

In [13]: J
Out[13]:
Matrix([
[-1, 0, 0],
[ 0, 1, 0],
[ 0, 0, 3]])

In [14]: Pv = P.inv()

In [15]: Pv * A * P
Out[15]:
Matrix([
[-1, 0, 0],
[ 0, 1, 0],
[ 0, 0, 3]])

In [16]: P * A * Pv

In [17]: Je = J.exp()

In [18]: Je
Out[18]:
Matrix([
[exp(-1), 0,      0],
[      0, E,      0],
[      0, 0, exp(3)]])

In [19]: P * Je * Pv
Out[19]:
```

```
Matrix([
[E, -E + exp(3), -E/2 + exp(-1)/4 + exp(3)/4],
[0,       exp(3),        -exp(-1)/4 + exp(3)/4],
[0,            0,                      exp(-1)]])
```

2.7 関数のテーラー展開

関数のテーラー展開は与えられた関数を多項式級数で近似する方法の 1 つに方法を与えます。

連続微分可能な実関数 $f : \mathbb{R} \to \mathbb{R}$ の点 x_0 の近くの点 x における値 $f(x)$ は次のような多項式級数

$$f(x) = f(x_0) + \frac{df(x_0)}{dx}(x-x_0) + \frac{1}{2!}\frac{d^2f(x_0)}{dx^2}(x-x_0)^2 + \cdots + \frac{1}{n!}\frac{d^nf(x_0)}{dx^n}(x-x_0)^n + R_{n+1}(x, x_0) \quad (2.6)$$

で表されます。これを関数 f の点 x_0 の周りの **Taylor 展開**（Taylor expansion）といいます。とくに、原点 $x_0 = 0$ のまわりの Taylor 展開を**マクローリン展開** Maclaurin expansion といいます。最後の項 $R_{n+1}(x, x_0)$ は $(n+1)$-次の**剰余項**といい、$0 < \theta < 1$ であるような θ があって

$$R_{n+1}(x, x_0) = \frac{1}{(n+1)!}\frac{d^{(n+1)}f(\theta(x-x_0))}{dx^{(n+1)}}(x-x_0)^{(n+1)}$$

と表されることが知られています。Taylor 展開が $x \neq x_0$ で収束し、$f(x)$ に一致するかどうかは剰余項の評価によって判定されます。

2.7.1 Landau 記法 ビッグ・オー $\mathcal{O}$ とスモール・オー o

関数 f が $\lim_{x\to a} f(x) = 0$ となるとき f は a において無限小、$\lim_{x\to a} f(x) = +\infty$ または $-\infty$ であるとき f は無限大といいます。同一の点 a の近くで無限小（または無限大）となる関数を比較するためにしばしば Landau 記法、o-記法と O-記法が用いられます [8, 第 II 章 §4]。関数 $\sqrt{x}$ と $\sin x$ はどちらも $x \to +0$ のとき 0 に収束しますが、x の減り方と比べると $\sqrt{x}$ はより遅く 0 に、$\sin x$ はより速く 0 に近づきます。

$$\lim_{x\to+0}\frac{\sqrt{x}}{x} = +\infty, \quad \lim_{x\to+0}\frac{\sin x - 1}{x} = 0$$

関数 g が a の除外近傍（a を含まない a の近傍）で $g(x) \neq 0$ とします。関数 $f(x)$ について、

$$\lim_{x\to a, \neq a}\frac{f(x)}{g(x)} = 0$$

が成り立つとき、f は g に比べて無視できるといい、**スモール・オー記法**を使って

$$f(x) = o(g(x)), \quad (x \to a)$$

と記します（$f \ll g$ と記す流儀もあります）。正確には $o(g)$ は関数族を表し、$f \in o(g)$ であるという意味です。f, g が共に a における無限小（または無限大）であるとき、$f(x) = o(g(x))$ なら f は g より高次の無限小（または高次の無限大）といいます。

一方、a の除外近傍において

$$\lim_{x \to a} \left| \frac{f(x)}{g(x)} \right| \leqq C$$

であるような定数 C が存在するとき（$|f/g|$ が有界なとき）、$x \to a$ のとき f は g で**押さえられる** (bounded) といい、**ビッグ・オー記法**を使って

$$f(x) = O(g(x)), \quad (x \to a)$$

と記します（$f \preccurlyeq g$ と記す流儀もあります）。$O(g)$ は関数族を表し、$f \in O(g)$ であるという意味です。特に、

$$\lim_{x \to a, \neq a} \frac{f(x)}{g(x)} = 1$$

のとき、f は a の近傍で g と**同値** (equivalcence) といい、$f \sim g \ (x \to a)$ と記すことがあります。

$a > b$ のとき x^a, x^b について、$x^a = o(x^b) \ (x \to 0)$ となります。一方、$f(x) = o(1) \ (x \to a)$ のとき $f(x) \to 0 \ (x \to a)$ であること、$f(x) = O(1) \ (x \to a)$ であれば $f(x)$ は a の近傍で有界であることを表します。$f(x) = 3x^3 - 4x^2$ は $x \to 0$ のとき、それぞれの項を考えて $f(x) = o(x^2) + o(x) = o(x)$ と書けます。

2.7.2 関数の多項式近似

関数 $f(x)$ の点 x_0 の周りの Taylor 展開における $n+1$ 次剰余項 $R_{n+1}(x, x_0)$ の代わりに Landau 記法を使って

$$R_{n+1}(x, x_0) = O\left((x - x_0)^{n+1}\right), \quad x \to x_0$$

または

$$R_{n+1}(x, x_0) = o\left((x - x_0)^{n}\right), \quad x \to x_0$$

と表すことができます。

例えば、$\sin x$ は $x = 0$ のまわりで第 1 次近似である x よりも正確な第 2 次近似として次のように表わされます。

$$\sin x = x - \frac{x^3}{6} + \mathcal{O}(x^5), \quad x \to 0.$$

実際、$\sin x$ の展開において 4 次より高次な項は $x^5/5! - x^7/7! + \cdots$ ですが、x^5 によって上から押さえられる

$$\lim_{x\to 0} \frac{x^5/5! - x^7/7! + \ldots}{x^5} = \frac{1}{5!}$$

ことが確認でき、$x \to 0$ のとき剰余項 $R_5(x, 0)$ を $\mathcal{O}(x^5)$ で置き換えることができます。一方、$\sin^2 x = x^2 - x^4/3 + 2x^6/45 - x^8/315 + 2x^{10}/14175 - \cdots$ は $x = 0$ のまわりで次のように表わされます。

$$\sin^2 ax = \left(ax - \frac{a^3}{6}x^3 + \mathcal{O}(x^5)\right)^2 = a^2x^2 - \frac{a^4}{3}x^4 + \mathcal{O}(x^6), \quad x \to 0.$$

こうした $\mathcal{O}$-記法（または o-記法）が意味を持つためには、どの値に近づくときの表式かを明記する必要があります。

Taylor 展開は与えられた関数 $f(x)$ を多項式で近似する議論で有用です。x が x_0 で連続で十分に近いとき、第 1 次近似

$$f(x) \sim f(x_0) + \frac{df(x_0)}{dx}(x - x_0), \quad x \to x_0$$

第 2 次近似

$$f(x) \sim f(x_0) + \frac{df(x_0)}{dx}(x - x_0) + \frac{1}{2!}\frac{d^2f(x_0)}{dx^2}(x - x_0)^2, \quad x \to x_0$$

がしばしば利用されます。テーラー展開を使った関数 $f(x)$ の $x \to x_0$ での第 k-次近似は多項式

$$f(x) \sim f(x_0) + \sum_{k=1}^{n} \frac{1}{k!}\frac{d^kf(x_0)}{dx^k}(x - x_0)^k, \quad x \to x_0$$

で表されます。

2.7.3 SymPy の級数展開 `series　.removeO`

SymPy 関数

`sympy.seriese(f, x, x0, n)`　または SymPy メソッド `f.series(x, x0, n)`

は、SymPy 表式 f の x について点 x0 の周りで n 次以上の項を $\mathcal{O}((x-x_0)^n)$ で置き換えたベキ級数を計算します。返されるオーダー項 O（大文字のオー） は Landau のビッグ・オー記法です。n を省略したときはデフォルト値 6 が使われます。なお、返り値にオーダ項が不要なときは、メソッド.removeO()（O は大文字オー）を使います。

$$\sin(x+y^2) \overset{x\text{ で展開}}{\longrightarrow} \sin y^2 + x\cos y^2 - \frac{x^2\sin y^2}{2} - \frac{x^3\cos y^2}{6} + \frac{x^4\sin y^2}{24} + \mathcal{O}(x^5)$$
$$\overset{y\text{ で展開}}{\longrightarrow} \sin x + y^2\cos x - \frac{y^4\sin x}{2} + \mathcal{O}(y^5)$$

```
In [1]: import sympy as sym

In [2]: x, y = sym.symbols('x y')

In [3]: f = sym.sin(x)

In [4]: g = sym.sin(x + y**2)

In [5]: f.series(x, 0, 5)
Out[5]: x - x**3/6 + O(x**5)

In [6]: g.series(x, 0, 5)
Out[6]: sin(y**2) + x*cos(y**2) - x**2*sin(y**2)/2 - x**3*cos(y**2)/6
     + x**4*sin(y**2)/24 + O(x**5)

In [7]: g.series(y, 0, 5)
Out[7]: sin(x) + y**2*cos(x) - y**4*sin(x)/2 + O(y**5)
```

多変数実（複素）関数 $f(x_0,\ldots,x_{n-1})$ のテーラー展開も定義できますが（節 2.7.6）、sympy.series は上の $g(x,y) = \sin(x+y)$ でみたような指定した 1 つの変数についてのみの級数展開しか計算しません。

2.7.4 SymPy でのプロット

sympy.plot または sympy.plot3d は、グラフィックライブラリ matplotlib がインストールされているときには、これを陽にインポートすることなく呼び出して 2 次元または 3 次元プロットが可能です。

プログラム 2.7-1 approx_function.py は、関数 $f(x) = \mathrm{e}^x\sin(x-a)$ を f.series(x, 0, 5) によって $x = 0$ の周りで 5 次までの展開して得られる式からオーダー項 $\mathcal{O}(x^5)$ を取り除いた（removeO）x の 4 次多項式

$$-\sin a + x(\cos a - \sin a) + x^2\cos a + x^3\left(\frac{\sin a}{3} + \frac{\cos a}{3}\right) + \frac{x^4\sin a}{6}$$

について、$a = 2$（`.subs(a, 2)`）として求められる関数（青線）と元の関数 $f(x, a = 2)$（赤線）とを区間 $[-\pi, \pi]$ でプロットして比較します（図 2.5）。得られる 4 次多項式は確かに原点 $x = 0$ の近くで良好な近似となっていることがわかります。

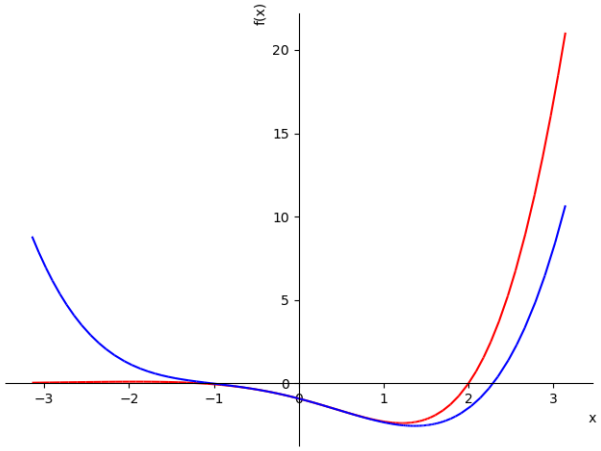


図 2.5　関数 $f(x) = \mathrm{e}^x \sin(x - a)$ を $x = 0$ の周りで展開して得られる 4 次多項式に $a = 2$ を代入した関数（青線）と区間 $[-\pi, \pi]$ で元の関数 $f(x, a = 2)$（赤線）とをプロットするプログラム 2.7-1 結果。原点付近で x の 4 次多項式は $f(x, a = 2)$ を良好に近似する。

コード 2.7-1　approx_function.py

```
import sympy as sy # matplotlibのインストールが前提

x, a = sy.symbols('x a')
f = sy.exp(x) * sy.sin(x - a)
fseri = f.series(x, 0, 5)

xmax = sy.pi
f2 = f.subs(a, 2)
fs2 = fseri.removeO().subs(a, 2)
plt = sy.plot(f2, fs2, (x, -xmax, xmax), show=False)
plt[0].line_color='red'
plt[1].line_color = 'blue'
plt.show()
```

2.7.5　スカラー場の勾配

スカラー場 (scalar field) とは、空間領域 $\mathcal{D} \subset \mathbb{R}^n(\mathbb{C}^n)$ 上の多変数実（複素）関数 $f : \mathcal{D} \to \mathbb{R}$ が各点 $\boldsymbol{x} = (x_0, \ldots, x_{n-1}) \in \mathcal{D}$ で 1 つの値（スカラー値）$f(x_0, \ldots, x_{n-1})$ を定めている様

子です（f 自身をスカラー関数と呼ぶことがあります)。たとえば、3 次元空間内の気温 $T(x,y,z)$ や圧力 $p(x,y,z)$ はスカラー場（関数）です。

2.7.5.1 勾配ベクトル

スカラー関数 $f(\boldsymbol{x})$ の**勾配** (gradient) $\operatorname{grad} f$（または ∇f)、各変数に関する 1 階偏微分 $\dfrac{\partial f}{\partial x_i}$ を要素とするベクトル値関数（通常は列ベクトル）で表します。

$$\operatorname{grad} f = \nabla f = \begin{bmatrix} \dfrac{\partial f}{\partial x_0} \\ \vdots \\ \dfrac{\partial f}{\partial x_{n-1}} \end{bmatrix}. \tag{2.7}$$

スカラー関数 f の勾配を取るとベクトル値関数 $\operatorname{grad} f(\boldsymbol{x})$ となります。

スカラー値 $f(\boldsymbol{x}) = c$ を関数 f の高さと見なすと、同じ高さ c を持つ空間内の集合は等位面 $L_c(f)$

$$L_c(f) = \{(x_0, \ldots, x_{n-1}) \,|\, f(x_0, \ldots, x_{n-1}) = c\}. \tag{2.8}$$

をなしています。

このとき、スカラー場の各点 $\boldsymbol{x}$ における勾配ベクトル $\operatorname{grad} f(\boldsymbol{x})$ は、その点 $\boldsymbol{x}$ が載っている等位面に直交する**法線ベクトル**となっていることがわかります。実際、ある点 $\boldsymbol{x}_0$ で高さ $f(\boldsymbol{x}_0)$ を持つ等位面 $L_{f(\boldsymbol{x}_0)}$ を考えます。この等位面内にある曲線 $\boldsymbol{x}(t)$ が、$t = 0$ で $\boldsymbol{x}_0$ を通過 $\boldsymbol{x}(0) = \boldsymbol{x}_0$ するとします。曲線 $\boldsymbol{x}(t)$ は等位面上にあるので値は変わりません。

$$f(\boldsymbol{x}(0)) = f(\boldsymbol{x}(t)).$$

この両辺を t で微分すると

$$0 = \frac{d}{dt} f(\boldsymbol{x}(t)) = \sum_{i=0}^{n-1} \frac{\partial f(\boldsymbol{x}(0))}{\partial x_i} \frac{dx_i(0)}{dt}$$

を得ますが、右辺の表式は点 $\boldsymbol{x}_0 = \boldsymbol{x}(0)$ での勾配ベクトル $\operatorname{grad} f(\boldsymbol{x}(0))$ と等位面に触れる接ベクトル $d\boldsymbol{x}(0)/dt$ との内積がゼロ

$$\left(\operatorname{grad} f(\boldsymbol{x}(0)), \frac{d\boldsymbol{x}(0)}{dt}\right) = 0$$

であることを表します。これより、勾配ベクトルと等位面上の接ベクトルが互いに直交し、勾配は等位面に対する法線ベクトルになっていることがわかりました。

図 2.6 は、プログラム 2.7-2 を使って描いた x, y-平面上のスカラー場

$$f(x, y) = x \exp\left(-x^2 - y^2\right) \tag{2.9}$$

の等高線と勾配ベクトルの様子を表しています（$f(x, y)$ の様子は節 2.2.3 の図 2.1）。矢印で表される勾配ベクトルの方向と等位面（等高線）が直交している様子がわかります。

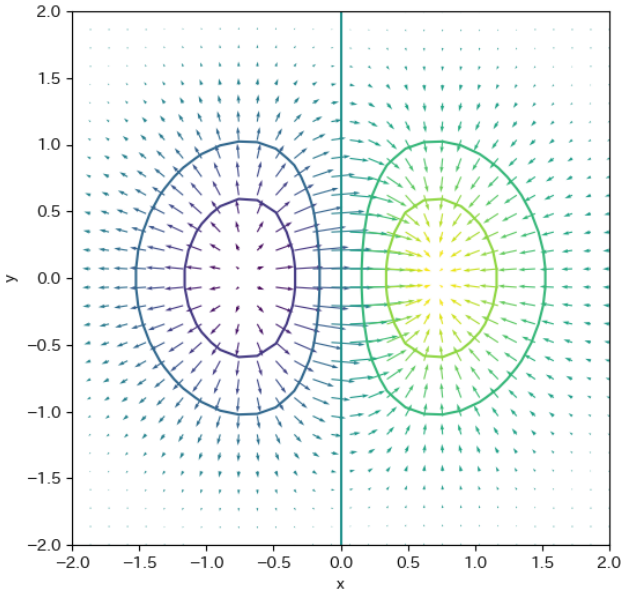


図 2.6　スカラー場 $f(x, y) = x \exp\left(-x^2 - y^2\right)$（図 2.1）の等高線と勾配ベクトル（プログラム 2.2-1）。各点における関数の勾配ベクトルはその点を通過する等高線に直交する法線ベクトルになっている。

2.7.5.2　NumPy の数値勾配列 `numpy.gradient`

関数 f の点 x_i における傾きを（偏）微分で計算するのではなく、幅 h を取って

$$\operatorname{grad} f(x_i) \approx \frac{f(x_i + h) - f(x_i - h)}{2h} \tag{2.10}$$

と中心差分値を数値的に近似することができます（$h \to 0$ で点 x_i での傾きに一致します）。この考えを推し進めて、与えられた点列 $f_{seq} = f_0, f_1, \ldots, f_{n-1}, f_{n-1}$ を等間隔 $h = 1$ に並んだ離散的関数値列とみなし、その中心差分列

$$g_i \equiv \frac{f_{i+1} - f_{i-1}}{2}, \quad i = 1, \ldots, n-2$$

ただし、端点は 1 階差分

$$g_0 \equiv \frac{f_1 - f_0}{1}, \quad g_{n-1} \equiv \frac{f_{n-1} - f_{n-2}}{1}$$

を勾配列 $\{g_i\}$ とみなすことができます。これを数値勾配列と言います。

NumPy 関数

```
numpy.gradient(fseq)
```

は、`fseq` が 1 次元 NumPy 配列のときはこの数値勾配列（数値微分列）を計算します。式 (2.10) の幅 h を変えるオプションも使えます)。`fseq` が多次元配列の場合、たとえば $m \times n$ 行列 $A = (a^i{}_j)$ のとき（行を第 0 軸として m 個の要素、列を第 1 軸として n 個の要素が並んでいるとき)、2 つの軸方向（行方向と列方向）に沿って得られる $m \times n$ 行列（浮動小数）を 2 つの要素とするリストを返します。

行方向の勾配配列は A の第 j 列について選んだ数列 $a^0{}_j, a^1{}_j, \ldots, a^{m-1}{}_j$ から得られる m 個の数値勾配列を n-列並べれ得られる $m \times n$ 行列、列方向の勾配配列は A の第 i 行について選んだ数列 $a^i{}_0, a^i{}_1, \ldots, a^i{}_{n-1}$ から得られる n 個の数値勾配列を m-行並べれ得られる $m \times n$ 行列です。

$$\text{numpy.gradient}\left(\begin{bmatrix} 1 & 2 & 6 \\ 3 & 5 & 4 \\ 7 & 8 & 10 \end{bmatrix}\right) \Rightarrow \left[\begin{bmatrix} 2 & 3 & -2 \\ 3 & 3 & 2 \\ 4 & 3 & 6 \end{bmatrix}, \begin{bmatrix} 1 & 5/2 & 4 \\ 2 & 1/2 & -1 \\ 1 & 3/2 & 2 \end{bmatrix}\right]$$

プログラム 2.7-2 `gradient_equilevel.py` は、スカラー場 $f(x, y) = x \exp -x^2 - y^2$ の勾配ベクトルを求め、matplotlib の quiver を使って矢印、`contour` を使ってスカラー場の等位面集合として等高線を描きます（図 2.6)。

x, y 上の格子点の x-方向配列と y-方向配列 `xg, yg` から高さ（スカラー場の値）`zg` $= f($`xg,yg`$)$ を計算し、`np.gradient(zg)` を使って、まず y-方向の、次いで x-方向の勾配ベクトル (dy, dx) を計算します（勾配ベクトルの成分の順番に注意)。グリッド点を含む詳しい計算方法は節 5.3 で改めて説明します。

コード 2.7-2　gradient_equilevel.py

```
import numpy as np
import matplotlib.pyplot as plt

def f(x: np.float_, y: np.float_) -> np.float_:
        return(x * np.exp(-x ** 2 - y ** 2))

xn, yn = 30, 30
xpoint: np.ndarray = np.linspace(-2,2, num=xn)
ypoint: np.ndarray = np.linspace(-2,2, num=yn)
xg, yg = np.meshgrid(xpoint, ypoint) # xg.shape=(yn,xn) yg.shape=(yn,xn)
```

```
zg: np.ndarray = f(xg, yg) # zg.shape=(yn,xn)
(dy, dx) = np.gradient(zg)  # 数値勾配 順番に注意  y方向の傾き,x方向の傾き

fig, ax = plt.subplots(figsize = (6, 6))
ax.contour(xg, yg, zg)
ax.quiver(xg, yg, dx, dy, zg)
ax.set(xlabel = 'x', ylabel = 'y')
#fig.savefig('gradient_equilevel.png')
fig.show()
```

2.7.6 多変数実関数の Taylor 展開

連続偏微分可能な多変数実関数（スカラー場）$f : \mathbb{R}^n \to \mathbb{R}$ の点 $\boldsymbol{x}_0 = (x_{0,0}, \ldots, x_{n-1,0})$ のまわりの k 次までの Taylor 展開は次のようになります。

$$f(\boldsymbol{x}) = \sum_{m=1}^{k} \frac{1}{m!} \left((x_0 - x_{0,0}) \frac{\partial}{\partial x_0} + \cdots + (x_{n-1,0}) \frac{\partial}{\partial x^n} \right)^m f(\boldsymbol{x}_0) + \mathcal{O}\left(\boldsymbol{x}^{k+1}\right) \tag{2.11}$$

ここで、式 (2.7) の勾配 $\mathrm{grad} f$ との内積や式 (2.1) の Hesse 行列 $H(f)$ を使って表すと

$$\begin{aligned} &= f(\boldsymbol{x}_0) + \left(\mathrm{grad} f(\boldsymbol{x}_0),\, \boldsymbol{x} - \boldsymbol{x}_0 \right) + \frac{1}{2!} (\boldsymbol{x} - \boldsymbol{x}_0)^T H(f)(\boldsymbol{x}_0) (\boldsymbol{x} - \boldsymbol{x}_0) \\ &\quad + \sum_{m=3}^{k} \frac{1}{m!} \left((x_0 - x_{0,0}) \frac{\partial}{\partial x_0} + \cdots + (x_{n-1} - x_{n-1,0}) \frac{\partial}{\partial x_{n-1}} \right)^m f(\boldsymbol{x}_0) + \mathcal{O}\left(\boldsymbol{x}^{k+1}\right) \end{aligned} \tag{2.12}$$

さらに、偏微分の順番が交換可能 $\frac{\partial^2}{\partial x_i \partial x_j} f = \frac{\partial^2}{\partial x_j \partial x_i} f$ なスカラー関数のときには

$$\begin{aligned} &= f(\boldsymbol{x}_0) + \left(\mathrm{grad} f(\boldsymbol{x}_0),\, \boldsymbol{x} - \boldsymbol{x}_0 \right) \\ &\quad + \sum_{m=2}^{k} \sum_{\ell_1 + \cdots + \ell_n = m} \frac{1}{\ell_1! \ldots \ell_n!} \prod_{i=1}^{n} (x_i - x_{i0})^{\ell_i} \frac{\partial^m}{\partial x_1^{\ell_1} \ldots \partial x_n^{\ell_n}} f(\boldsymbol{x}_0) + O\left(\boldsymbol{x}^{k+1}\right). \end{aligned} \tag{2.12′}$$

これらの計算において演算子 $\partial/\partial x^i$ は線形演算子として作用しています。

$$\frac{\partial}{\partial x^i} \left(a_j \frac{\partial}{\partial x^j} + b_k \frac{\partial}{\partial x^k} \right) = a_i \frac{\partial}{\partial x^i} \frac{\partial}{\partial x^j} + b_j \frac{\partial}{\partial x^i} \frac{\partial}{\partial x^k}.$$

以降では、演算子 $\frac{\partial}{\partial x_k}$ を ∂_{x_k}、関数 f の偏微分 $\frac{\partial f}{\partial x_k}$ を f_{x_k} などと略記します。

たとえば、式 (2.9) の 2 変数のスカラー場 $f(x,y) = x\exp(-x^2-y^2)$ を 3 次まで展開してみましょう。この関数は偏微分の順序が交換できるため、次を計算すればよいことがわかります。

$$\begin{aligned}&\left((x-x_0)\frac{\partial}{\partial x}+(y-y_0)\frac{\partial}{\partial y}\right)^3\\&\quad=(x-x_0)^3\partial_{xxx}+3(x-x_0)^2(y-y_0)\partial_{xxy}+3(x-x_0)(y-y_0)^2\partial_{xyy}+(y-y_0)^3\partial_{yyy}\end{aligned}$$

これより、スカラー場 $f(x,y)$ を原点 $(0,0)$ のまわりで x,y に関して 3 次多項式として展開すると次のようになります。

$$\begin{aligned}f(x,y)&\sim f(0,0)+f_x(0,0)x+f_y(0,0)y+\frac{1}{2!}\left(f_{xx}(0,0)x^2+2f_{xy}(0,0)xy+f_{yy}(0,0)y^2\right)\\&\quad+\frac{1}{3!}\left(f_{x,x,x}(0,0)x^3+3f_{xxy}(0,0)x^2y+3f_{xyy}(0,0)xy^2+f_{y,y,y}(0,0)y^3\right)\\&=x-xy^2-x^3\end{aligned}$$

SymPy を使う場合、節 2.7.3 で触れたように、多変数スカラー関数 $f:\mathbb{R}^n\to\mathbb{R}$ に関してその Taylor 展開を直接返す組込み関数はいまのところ見当たりません。個別に偏微分して得た結果に点 $\boldsymbol{x}_0$ の値を代入して展開係数を求めることはできます。

2.7.7 ベクトル値関数の展開と Jabobi 行列

n 次元空間上の多変数ベクトル値関数 $\boldsymbol{f}:\mathbb{R}^n\to\mathbb{R}^m$ は各点 $\boldsymbol{x}=(x_0,\ldots,x_{n-1})$ でベクトル値を定めています。このためベクトル値関数を**ベクトル場**ということがあります。ベクトル場 $\boldsymbol{f}(\boldsymbol{x})$ を m 個の n-変数実関数 $\{f_i(x_0,\ldots,x_{n-1})\}(i=0,\ldots m-1)$ を成分とする列ベクトルで表します。

$$\boldsymbol{f}(\boldsymbol{x})=\begin{bmatrix}f_0(\boldsymbol{x})\\\vdots\\f_{m-1}(\boldsymbol{x})\end{bmatrix}$$

ベクトル場関数の Taylor 展開もいままでと同様に考えることができます。ここでは、実用上しばしば利用される第 1 次近似を与える 2 次までの展開を扱います。

ベクトル値関数 $\boldsymbol{f}(\boldsymbol{x})$ の点 $\boldsymbol{x}_0$ の周りの 1 次までの Taylor 展開は、式 (2.11) から次のよう

に表されます。

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_0(\boldsymbol{x}_0) \\ \\ f_1(\boldsymbol{x}_0) \\ \\ \vdots \\ \\ f_{m-1}(\boldsymbol{x}_0) \end{bmatrix} + \begin{bmatrix} \dfrac{\partial f_0(\boldsymbol{x}_0)}{\partial x_0} & \dfrac{\partial f_0(\boldsymbol{x}_0)}{\partial x_1} & \cdots & \dfrac{\partial f_0(\boldsymbol{x}_0)}{\partial x_{n-1}} \\ \dfrac{\partial f_1(\boldsymbol{x}_0)}{\partial x_0} & \dfrac{\partial f_1(\boldsymbol{x}_0)}{\partial x_1} & \cdots & \dfrac{\partial f_1(\boldsymbol{x}_0)}{\partial x_{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial f_{m-1}(\boldsymbol{x}_0)}{\partial x_0} & \dfrac{\partial f_{m-1}(\boldsymbol{x}_0)}{\partial x_1} & \cdots & \dfrac{\partial f_{m-1}(\boldsymbol{x}_0)}{\partial x_{n-1}} \end{bmatrix} (\boldsymbol{x} - \boldsymbol{x}_0) + \mathcal{O}\left(\boldsymbol{x}^2\right) \tag{2.13}$$

$\boldsymbol{f}(\boldsymbol{x})$ が滑らかで $\boldsymbol{x}$ が $\boldsymbol{x}_0$ に十分近いとき、2 次以上の高次項を無視して第一次近似

$$\boldsymbol{f}(\boldsymbol{x}) \approx \boldsymbol{f}(\boldsymbol{x}_0) + J_{\boldsymbol{x}}(\boldsymbol{f})(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0))$$

を考えることがあります。これをベクトル場の**線形化** (linearization) といいます（節 6.2 参照)。

$J_{\boldsymbol{x}}(\boldsymbol{f})$ は式 (2.13) で現れた m 行 n 列の関数行列で、点 $\boldsymbol{x}$ におけるベクトル値関数 $\boldsymbol{f}$ の**ヤコビ行列** (Jacobian) といいます。

$$J_{\boldsymbol{x}}(\boldsymbol{f}) \equiv \begin{bmatrix} \dfrac{\partial f_0}{\partial x_0} & \dfrac{\partial f_0}{\partial x_1} & \cdots & \dfrac{\partial f_0}{\partial x_{n-1}} \\ \dfrac{\partial f_1}{\partial x_0} & \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial f_{m-1}}{\partial x_0} & \dfrac{\partial f_{m-1}}{\partial x_1} & \cdots & \dfrac{\partial f_{m-1}}{\partial x_{n-1}} \end{bmatrix} = \frac{\partial(f_0, \ldots, f_{m-1})}{\partial(x_0, \ldots, x_{n-1})} \tag{2.14}$$

たとえば、σ, r, b をパラメータ、$\boldsymbol{x} = (x, y, z)$ を変数とする次のベクトル場関数 $\boldsymbol{V}(\boldsymbol{x})$ を考えてみます。

$$\boldsymbol{V}(\boldsymbol{x}) = \begin{bmatrix} v_0(\boldsymbol{x}) \\ v_1(\boldsymbol{x}) \\ v_2(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} -\sigma x + \sigma y \\ -xz + rx - y \\ xy - bz \end{bmatrix} \tag{2.15}$$

このとき、その Jacobi 行列は次のように計算できます。

$$J_{\boldsymbol{x}}(\boldsymbol{V}) = \begin{bmatrix} \dfrac{\partial v_0}{\partial x} & \dfrac{\partial v_0}{\partial y} & \dfrac{\partial v_0}{\partial z} \\ \dfrac{\partial v_1}{\partial x} & \dfrac{\partial v_1}{\partial y} & \dfrac{\partial v_1}{\partial z} \\ \dfrac{\partial v_2}{\partial x} & \dfrac{\partial v_2}{\partial y} & \dfrac{\partial v_2}{\partial z} \end{bmatrix} = \begin{bmatrix} -\sigma & \sigma & 0 \\ r - z & -1 & -x \\ y & x & -b \end{bmatrix} \tag{2.16}$$

2.7.8 SymPy のヤコビ行列 `jacobian`

SymPy メソッド

```
vec.jacobian((x,y,z,..))
```

は、ベクトル値関数 `vec` の変数 `x,y,z,..` に関するヤコビ行列を計算します。

式 (2.15) で与えられたベクトル場関数の 3×3 ヤコビ行列を求めます。

```
In [1]: import sympy as sy

In [2]: x, y, z = sy.symbols('x y z')

In [3]: p, r, b = sy.symbols('p r b')

In [4]: lorenzv = sy.Matrix([[-p*x + p*y], [-x*z + r*x - y], [x*y - b*z]])

In [5]: lorenzv
Out[5]:
Matrix([
[   -p*x + p*y],
[r*x - x*z - y],
[   -b*z + x*y]])

In [6]: lorenzv.jacobian((x, y, z))
Out[6]:
Matrix([
[   -p,  p,  0],
[r - z, -1, -x],
[    y,  x, -b]])
```

第 3 章　離散力学系のプロット

3.1　離散力学系

微分方程式は時刻 $t = 0$ で初期値 $x(0) = x_0$ が与えられると、時間の連続変化につれてその解軌道 $x(t)$ は（通常は）連続的に変化していきます。単純な場合な場合として、初期点 x_0 を与えられた写像 T の反復適用によって得られる点列を軌道とみなす離散系を考えることができます。

写像の反復が微分方程式の研究に深く関わっていることが後に明らかになります。

3.1.1　写像の反復

区間 I 上の関数 $f(x)$ を点 x を点 $f(x)$ に写す写像 $x \mapsto f(x)$ と見なします。このとき、区間内の点 $x_0 \in I$ の関数値 $x_1 = f(x_0)$ を計算し、さらにその関数値 $x_2 = f(x_1) = f(f(x_0)) = f^2(x_0)$ を計算していく過程をさらに繰り返すことを写像 f の**反復** (iteration) と呼びます。このとき、反復開始点 x_0 を初期点とする次のような点列

$$x_0,\ x_1 = f(x_0),\ x_2 = f(x_1),\ \ldots,\ x_n = f(x_{n-1}) = f^n(x_0),\ \ldots$$

を考えます（関数の定義区間 I は、反復写像点列 $\{f^n(x_0)\}_{n=0,1,2,\ldots}$ を含むように取られているとします)。

x_0 から写像 f の反復によって得られる無限点列

$$\mathcal{O}_f(x_0) = \{x_0, x_1, x_2, \ldots, x_n, \ldots\}$$

を初期値 x_0 の**軌道** (orbit) といいます。軌道においては、f の反復回数（整数）を離散的時間とみなすと、

$$x_{s+t} = f^{s+t}(x_0) = f^t(f^s(x_0)), \quad s, t \geqq 0$$

の関係があります。このとき写像 f とその定義区間 I を組にした (f, I) は**離散力学系** (discrete dynamical system) を定めるといいます。図 3.1 は 1 次元力学系で初期点 x_0 から二次関数 f の反復から得られる軌道点列 $x_0, x_1, x_2, \ldots$ を示しています。

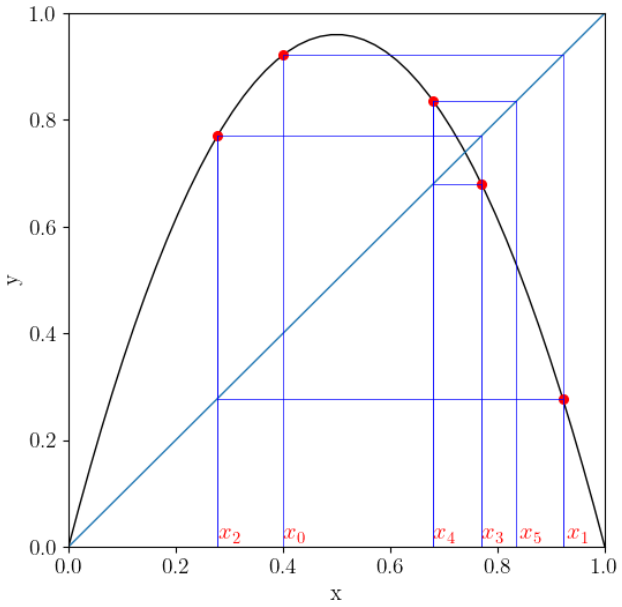


図 3.1　ロジスティック写像 $f_a(x) = ax(1-x)$ の区間 $[0,1]$ 上のグラフ ($a = 3.84$) と初期値 $x_0 = 0.4$ の 5 回反復した点列 x_1, x_2, x_3, x_4, x_5 (クモの巣図)。f_a のグラフは $(0,0)$ および $(1,0)$ を通り $x = 1/2$ で極大値 $a/4$ を取り、$a > 1$ のとき $f_a(x)$ のグラフは $x^* = 1 - \frac{1}{a}$ で直線 $y = x$ と交わり x^* は不動点 $f_a(x^*) = x^*$ である。

3.1.2　一次元写像の反復

離散力学系では、軌道の追跡電卓を使って実行できるだけで、軌道点列の挙動が単純なわけではありません。有名な例として 1 次元区間 $[0,1]$ 上の**ロジスティック関数**

$$f_a(x) = ax(1-x), \quad 0 < a \leqq 4 \tag{3.1}$$

があります。f_a で区間 I で連続な写像で、パラメータ a は $0 < a \leqq 4$ の範囲を考えます。

図 3.1 からわかるように単位区間 $I = [0,1]$ に対して $f(I) \subseteq I$ に注意すると、任意の初期値 $x_0 \in [0,1]$ から出発する f_a による反復軌道は常に区間 $[0,1]$ 上にあり、a を留めるごとに離散力学系 $(f_a, [0,1])$ が定まります。図 3.1 は、パラメータ $a = 3.84$ としたときに初期値 $x_0 = 0.4$ から 5 回反復した点列 x_1, x_2, x_3, x_4, x_5 の求め方を示すクモの巣図（節 3.5.1）です。

ロジスティック写像で定まる力学系は改めて節 3.5 で紹介しますが、パラメータ a によって定まる軌道はたいへん複雑な構造を持っているため、深く研究されてきました [17]。

3.1.3 一般次元写像の反復

離散力学系は n-次元空間においても同様に考えることができます。領域 $\mathcal{D} \subset \mathbb{R}^n$ 内の点 $\boldsymbol{x}$ を再び $\mathcal{D}$ 内の点 $T(x)$ に写す写像 $\boldsymbol{x} \mapsto T(\boldsymbol{x})$ の反復によって点列 $\{T^n(\boldsymbol{x})\}_{n=0,1,2,\ldots}$ を考えることができ、離散力学系 $(T, \mathcal{D})$ が定まります。

離散力学系 $(T, \mathcal{D})$ の**不動点** (fixed points) とは、写像 T によって動かない点からなる集合

$$\{\boldsymbol{x}^* \,|\, T(\boldsymbol{x}^*) = \boldsymbol{x}^*\}$$

です。ある $p > 0$ について T の p 回反復について $T^p(\boldsymbol{x}) = \boldsymbol{x}$ であるとき、点 $\boldsymbol{x}$ を p-**周期点** (periodic points) といいます（T^p の不動点となっています)。ただし、周期 p として $T^m(\boldsymbol{x}) = \boldsymbol{x}$ を満たす最小の正整数を考えることにします。

3.2 離散軌道のプロット

2 次元離散力学系では写像 T の反復で得られる平面上の軌道点列プロットがその力学系を理解する格好の手段の 1 つを提供します。

平面 $\mathbb{R}^2$ 上の写像 $T : \mathbb{R}^2 \to \mathbb{R}^2$ は平面上の点 $\boldsymbol{x}$ を平面上の別の点 $\boldsymbol{x}' = T(\boldsymbol{x})$ に写します。点 $\boldsymbol{x}$ を x, y-成分で表記して列ベクトル $\boldsymbol{x}=[x, y]^T$ で表すと、写像 T は 2 つの 2 変数実関数 $f, g : \mathbb{R}^2 \to \mathbb{R}$ を使って一般的に次のように表されます。

$$\boldsymbol{x}' = T(\boldsymbol{x}) \Leftrightarrow \begin{bmatrix} x' \\ y' \end{bmatrix} = T \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f(x, y) \\ g(x, y) \end{bmatrix}.$$

点 $\boldsymbol{x}_0 = [x_0, y_0]^T$ を初期条件として写像 T を反復して適用して得られる無限点列を T の軌道 $\mathcal{O}_T(\boldsymbol{x}_0) = \{\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \ldots\}$ と記したとき、$\boldsymbol{x}_n$ は次のように計算できます。

$$\boldsymbol{x}_n = \begin{bmatrix} x_n \\ y_n \end{bmatrix} = T \begin{bmatrix} x_{n-1} \\ y_{n-1} \end{bmatrix} = T^n \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}.$$

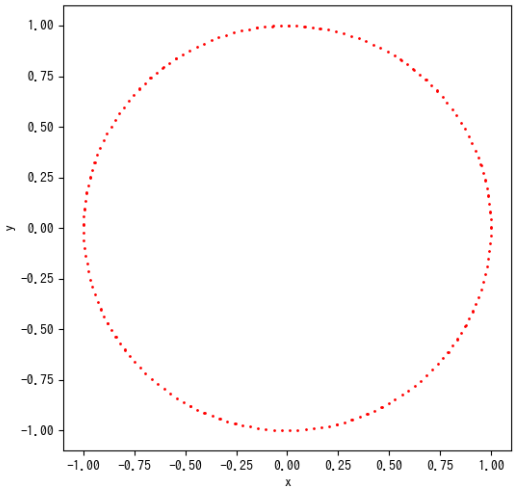


図 3.2　原点のまわりで回転させる写像を $n = 200$ 回反復して得られる軌道プロット（コード 3.2-1）。回転角 $\pi/\sqrt{5}$（ラジアン）、初期点 $(1.0, 0.0)$。非有理角の回転写像では、反復を繰り返すと軌道は円周上を稠密に埋め尽くしていく。

3.2.1　回転写像

写像で定まる離散力学系を調べるには初期点からの軌道をプロットすると大いに参考になります。

簡単な例として原点を中心に点 (x, y) を角度 θ 回転して点 (x', y') に移す回転写像 $R_\theta : \mathbb{R}^2 \to \mathbb{R}^2$ を考えましょう。回転 $(x, y) \mapsto R_\theta(x, y)$ は回転行列 R_θ によって次のように表されます。

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R_\theta(x, y) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}.$$

回転の場合、n 回反復した写像 ${R_\theta}^n$ は角度 $n\theta$ の回転行列 $R_{n\theta} = \begin{bmatrix} \cos n\theta & -\sin n\theta \\ \sin n\theta & \cos n\theta \end{bmatrix}$ で表されるのですが、ここでは一般写像でのように反復しながら逐次的に計算して軌道点列を求めます。

コード 3.2-1 `rotation_plot` は、与えた点を角度 `theta`（ラジアン）回転した点計算する関数 `rotation(x, theta)` を $n = 200$ 回反復して得られる軌道点列 $\{(x_i, y_i)\}_{i=0,1,\ldots,n-1}$ をプロットします（図 3.2）。ここでは、回転角を π(NumPy 円周率 `numpy.pi`）に関して非有理な角度 $\pi/\sqrt{5}$、初期値を $x0 = (1.0, 0.0)$ としてを軌道計算して

います。π の非有理数倍の回転を繰り返して得られる軌道無限列は円周上を稠密に埋め尽くします（図 3.2）。

コード 3.2-1　rotation_plot

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

def rotation(x: np.ndarray , theta: np.float_) -> np.ndarray:
    x1 = x[0] * np.cos(theta) - x[1] * np.sin(theta)
    y1 = x[0] * np.sin(theta) + x[1] * np.cos(theta)
    return(np.array([x1, y1]))

n = 200 # プロット回数
theta = np.pi / np.sqrt(5) # 非有理回転角度
xini, yini = 1.0, 0.0
x0 = np.array([xini, yini])  # 初期点 NumPy配列

norbit = np.empty((n,2)) # 軌道要素の配列を確保
for k in range(n):
    norbit[k] = x0
    x0 = rotation(x0, theta)

fig, ax = plt.subplots(figsize=(6.0, 6.0))
ax.plot(norbit[:,0], norbit[:,1], 'r.', markersize=2)
ax.set(xlabel='x', ylabel='y')
#fig.savefig('rotation_plot.png')
#fig.show()
```

3.2.2　軌道を求める逐次計算

軌道点列は逐次的に計算する必要があり、反復回数が増えると計算法に起因する差異の蓄積は無視できなくなります。反復を順次繰り返すという計算アルゴリズムは変えようがありませんが、計算の実現にはいくつかの方法があります。

表 3.1 に軌道点列を計算するための 3 つの方法を掲げ、繰り返し回数を $n = 10000$ として `%%timeit` を使って複数行のコードの実行時間を計測した結果を添えました（計測値は利用環境に依存するのであくまで参考値です）。コード 3.2-1 `rotation_plot` は表 3.1 の方法 (1) を使って初期点 x_0 からの軌道を計算しています。

表 3.1　反復写像による軌道プロットの 3 方法とその実行時間比較。

方法	コード

(1) NumPy 配列を確保	`x0 = np.array([xini, yini])`
	`norbit = np.empty((n,2))`
	`for k in range(n):`
	`  norbit[k] = x0`
	`  x0 = rotation(x0, theta)`

実行時間（参考）	151 ms　($n = 10000$)

(2) NumPy 配列に積み上げ	`x0 = np.array([xini, yini])`
	`norbit = x0`
	`for k in range(n-1):`
	`  x0 = rotation(x0, theta)`
	`  norbit = np.vstack((norbit,x0))`

実行時間（参考）	361 ms

(3) x, y リストに要素追加	`xorbit = []`
	`yorbit = []`
	`for i in range(n):`
	`  xorbit.append(xini)`
	`  yorbit.append(yini)`
	`  xnext, ynext`
	`  = rotation([xini,yini], theta)`
	`  xini, yini = xnext, ynext`

実行時間（参考）	173 ms

方法 (1) は、`numpy.empty` を使って軌道数列を格納する n 行 2 列（shape (n,2) の）配列 `norbit` をまず最初に確保しておいた上で、点 `x0` を `norbit[k]` に代入し、写像で写った点を `x0` として更新を繰り返します。3 つの方法の中ではこの方法が最速です。

方法 (2) は、初期点 x_0（shape (2,0) を持つ）を軌道要素として格納した NumPy 配列 `norbit` を、写像で写った点を `x0` に更新しながら NumPy 配列 `norbit` を 0-軸（行方向）に `norbit = np.vstack((norbit,x0))` と積み上げて NumPy 配列の更新を繰り返します。3 つの方法の中ではこの方法が最も遅く、最速値の 2 倍以上の計算時間を要しました。

方法 (3) は、x, y-方向の空リスト `xorbit`, `yorbit` を 2 つ用意しておき、各成分の初期値 `xini`, `yini` を各リストに追加し、点 `[xini,yini]` を写した点の x, y-成分を初期値の更新を繰り返します。3 つの方法の中では方法 (1) には及びませんが、方法 (2) よりも格段に速く計算できました。

この方法比較によって、NumPy を使えばいつでも計算が高速化されるわけではないことがわかりました。NumPy では多次元配列を高速に処理するために、配列に必要なメモリ領域を確保します。このため、方法 (2) のようにな `numpy.append()` や `numpy.vstack()numpy.hstack()`（節 1.3.6）などを繰り返し利用すると、その度ごとに配列形状を作り直すための要素コピーが発生するための時間がかかります。

一方、方法 (3) のような可変長で自由な型要素を保持する Python リストを使う場合には、要素ごとに少し大きめのメモリを要しますが、要素の追加に際してリストポインタを伸ばすだけで頻繁な要素コピーが発生しないために、その度ごとに配列形状変化を伴う NumPy よりもすばやく実行できました。

いずれの場合でも、繰り返し計算のたびにプロット `plot` をコールするような計算では、反復回数 n が大きくなると非常に大きな計算コストを要することになりうんと遅くなります。軌道のプロットのような反復プロットでは、必要なデータを一度に渡して `plot` のコール回数を減らす工夫を探ってください。

3.2.3 エノンの面積保存写像

回転写像では軌道は自明で、どの軌道も原点を中心とし初期値を載せる円上にあります。回転角度が有理角度の場合は円周上にある有限個の周期点、非有理角度の場合は軌道は円周を稠密に覆います。

回転写像に摂動を加えてわずかに変更した場合、一般に劇的な違いが生じます。写像クラスを限定するために、ここでは 2 次元写像 $T(x, y) = (f(x, y), g(x, y))$ のヤコビ行列（節

2.14） $J(T)$ の行列式が 1 となる連続写像を考えます。

$$\det J(T) = \begin{vmatrix} \dfrac{\partial f}{\partial x} & \dfrac{\partial f}{\partial y} \\ \dfrac{\partial g}{\partial x} & \dfrac{\partial g}{\partial y} \end{vmatrix} = \frac{\partial f}{\partial x}\cdot\frac{\partial g}{\partial y} - \frac{\partial f}{\partial y}\cdot\frac{\partial g}{\partial x} = 1 \tag{3.2}$$

このとき、平面上の閉曲線 $\mathcal{C}$ は写像 T によって再び閉曲線 $T(\mathcal{C})$ に写され、閉曲線で囲まれた面積は変わりません。このような性質を持つ写像を**面積保存写像** (area preserving map) または保測写像といいます。回転写像は明らかに円を円に写す保測変換になっています。

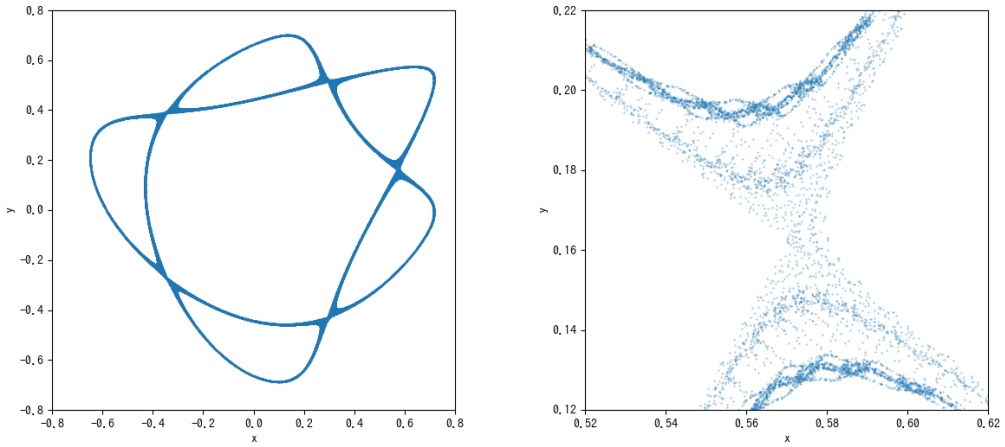
エノン（M. Hénon）は、原点 $(0,0)$ が不動点である平面上の面積を不変に保つ条件、式 (3.2) を満たす角度 a の回転 R_a に関係付けられる 2 次多項式写像 H_a を研究しました [35]。

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = H_a(x,y) = \begin{bmatrix} x\cos a - (y-x^2)\sin a \\ x\sin a + (y-x^2)\cos a \end{bmatrix} = R_a(x,y) + x^2 \begin{bmatrix} \sin a \\ -\cos a \end{bmatrix} \tag{3.3}$$

実際、写像 H_a のヤコビ行列 $J(H_a)$ を計算してみるとその行列式は $\det J(H_a) = \sin^2 a + \cos^2 a = 1$ になって面積保存条件 (3.2) を満たします。この面積保存条件は、ハミルトン力学系のベクトル場の発散がゼロである式 (7.6) に対応して領域体積は流れに共に変化しないことに対応します（節 7.1 参照）。

図 3.3 は式 (3.3) でパラメータ $a = \cos^{-1} 0.24$、初期点 $x_0, y_0) = (0.569, 0.15)$ と選んで $n = 80000$ 回反復して得られた軌道のプロット図です。一見、図 (a) は 2 つの輪が 5 箇所で交差したように見えますが、図 (b) のように領域 $[0.52, 0.62] \times [0.12, 0.22]$ を拡大してみると大変複雑な構造を持つ軌道になっていることがわかります（これらの点は 1 つの初期点を出発した 1 本の軌道点列です）。

こうした反復計算が数値誤差を含むことは避けられませんが、詳しい数学的研究からこうした計算結果は少なくとも定性的な観点から正しいことがわかっており、この様子は自由度 2 の古典力学系の一般的な状況を表しています（節 7.2）。



(a) 軌道全体図　　(b) 拡大図

図 3.3　エノン写像（式 (3.3)、パラメータ $a = \cos^{-1} 0.24$）を $n = 80000$ 回反復して得られる軌道。(a) 初期値 $(x_0, y_0) = (0.580, 0.15)$ を出発した軌道全体。(b) 拡大して領域 $[0.52, 0.62] \times [0.12, 0.22]$ を見た図。軌道点列は 1 次元的な曲線上には載っておらず複雑な構造を持つ。

コード 3.2-2 `henon_area_map` は、エノン写像 (3.3) の軌道をプロットします。パラメータ a を与える `np.arccos(0.24)` は逆余弦関数 $\cos^{-1}$ を表す NumPy 関数です。文献 [35, Table 1, Fig.13] では、$\cos a = 0.24$ のときに様々な初期値から 10000 回反復して得た軌道を重ねた描いています（1960 年代当時は今とは違って計算コストは大変高価でした）。

コード 3.2-2　henon_area_map

```
import numpy as np
import matplotlib.pyplot as plt

def henon_area(x: np.ndarray, a: float) -> np.ndarray:
    x1 = x[0] * np.cos(a) - (x[1] - x[0] ** 2) * np.sin(a)
    y1 = x[0] * np.sin(a) + (x[1] - x[0] ** 2) * np.cos(a)
    return(np.array([x1, y1]))

a = np.arccos(0.24)# パラメータ
iteration = 80000# 反復回数
xini, yini = 0.580, 0.15
x0 = np.array([xini, yini])  # 初期点

orbit = np.empty((iteration,2))
for k in range(iteration):
```

```
    orbit[k] = x0
    x0 = henon_area(x0, a)

fig, ax = plt.subplots(figsize=(6.0, 6.0))
ax.plot(norbit[:,0], norbit[:,1], '.', markersize=0.5)
ax.set(xlabel='x', ylabel='y',
    xlim=(0.52, 0.62), ylim=(0.12, 0.22))
#fig.savefig('henon_area_map.png')
#fig.show()
```

3.2.4 曲線の写像反復

平面上の閉曲線 $\mathcal{C}$ が連続写像 T によってどのように $\mathcal{C}, T(\mathcal{C}), T^2(\mathcal{C}), \ldots$ と変形されていくか各閉曲線をプロットしえ観察してみましょう。

図 3.4 は、エノン写像 (3.3) のパラメータを $a = \cos^{-1} 0.24$（図 3.3 の場合と同じ）としたとき、閉曲線が反復写像されていく様子です。初期閉曲線を原点 $(0, 0)$ を中心とした半径 0.7 の円とし、写像されるたびに線種を（実線、破線、一点鎖線、点線）を繰り返して変えながら閉曲線としてプロットしました。変形されながらも閉曲線で囲まれた面積を保っていることがわかります。

与えられた閉曲線 $\mathcal{C}$ 上にある点 $\boldsymbol{x}_0 \in \mathcal{C}$ を初期点とする写像 T を $k = (N-1)$ 回まで繰り返して得られる N 個の軌道点列を $\mathcal{O}_T^N(\boldsymbol{x}_0) = \{\boldsymbol{x}_0, \ldots, T^{N-1}(\boldsymbol{x}_0)\}$ と記すことにします。

閉曲線 $\mathcal{C}$ 上に並んだ $s+1$ 個の点 $\{\boldsymbol{x}_0^j\}_{j=0,\ldots,s}$ は、反復した時点のたびに $\{T^m(\boldsymbol{x}_0^j)\}_{j=0,\ldots,s} \in T^m(\mathcal{C})$ となっていることから、$s+1$ 本の軌道点列 $\{\mathcal{O}_T^N(\boldsymbol{x}_0^j)\}_j$ のそれぞれ k-番目（$k = 0, \ldots, N-1$）の点をつないで閉曲線 $T^k(\mathcal{C})$ と見立てることにします。

これ状況をプログラミングするにはいくつかのアプローチが考えられます。ここでは、0 行目と 1 行目の要素をプロットすると N 点が閉曲線 $\mathcal{C}$ となるような shape (2,N+1) の初期配列 `points` を用意します。写像 T を NumPy 配列を引数とするように定義してあれば、ベクトル化計算によって次に写像された点列 `next` は次のように計算できます。

```
for k in range(N):
    plot(points[0], points[1]) # 閉曲線としてプロットされる
    next =T(points)
    points = next
```

コード 3.2-3 `henon_map_of_closedcurve` は、エノン写像によって初期に与えた円から写像反復を繰り返してプロットします（図 3.4）。関数 `points_circle(cx, cy, r, n)` は、原点 (cx, cy) を中心とする半径 r の円周上にある点列を shape (2,n+1) の NumPy

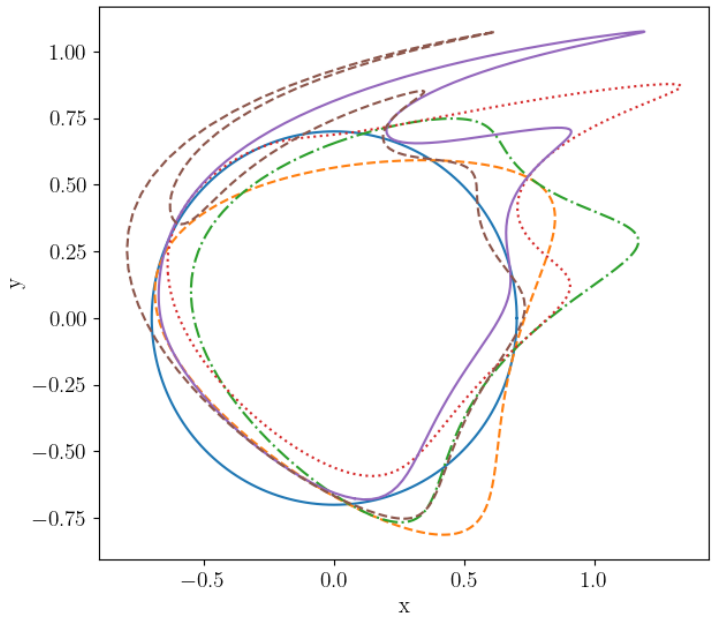


図 3.4　エノン写像 (3.3) の面積保存性（図 **3.3** と同じパラメータ $a = \cos^{-1} 0.24$）。初期閉曲線を原点 $(0, 0)$ を中心とした半径 **0.7** の円とし、写像されるたびに線種を（実線、破線、一点鎖線、点線）を繰り返して変えながら閉曲線としてプロット。閉曲線に囲まれた面積は変わらない。

配列を与えます（点列端点を一致させてプロットしたときに閉じるようにしています)。なお、変形された閉曲線上の点をプロットする際、4 つの線種（実線、破線、一点鎖線、点線）を linestyles として用意しておき、繰り返し変数 k を使って $k \pmod 4$ で周期的にプロット線種を変えるようにします。

コード 3.2-3　henon_map_of_closedcurve

```
import numpy as np
import matplotlib.pyplot as plt

def henon_area(x : np.ndarray, a: float) -> np.ndarray:
    x1 = x[0] * np.cos(a) - (x[1] - x[0] ** 2) * np.sin(a)
    y1 = x[0] * np.sin(a) + (x[1] - x[0] ** 2) * np.cos(a)
    return(np.array([x1, y1]))

# 中心(cx,cy)、半径rの円上のn+1個の点 cpoint[:,i] からなる配列 shape (2, n+1)
def circle_points(cx: np.float_, cy: np.float_, r: np.float_, n: int) -> np.
    ndarray:
    xpoints = []
    ypoints = []
    for k in range(n+1):
        xpoints.append(cx + r * np.cos(2*np.pi * k / n))
        ypoints.append(cy + r * np.sin(2*np.pi * k / n))
    cpoint = np.array([np.array(xpoints), np.array(ypoints)])
    return(cpoint)

a = np.arccos(0.24)# パラメータ
cnum = 200 # 円周上の点の数
circlepts = circle_points(0, 0, 0.7, cnum)
iteration = 6 # 反復回数

fig, ax = plt.subplots(figsize=(6.0, 6.0))
linestyles = ['solid','dashed','dashdot','dotted', 'solid'] # 線種

for k in range(iteration):
    ax.plot(circlepts[0], circlepts[1], ls=linestyles[np.mod(k, 4)])
    next_circlepts = henon_area(circlepts, a)
    circlepts = next_circlepts
ax.set(xlabel='x', ylabel='y')
#fig.savefig('henon_map_of_closedcurve.png')
```

```
plt.show()
```

3.3 反復関数系 (IFS)

写像の反復によって得られる軌道プロットの例として、与えられた関数の集まり $\{F_1, \ldots, F_M\}$ からランダムに選んで k 回反復した $F_{i_k} \circ F_{i_{k-1}} \circ \ldots F_{i_1}(\boldsymbol{x}) = F_{i_k}(F_{i_{k-1}} \cdots (F_{i_1}(\boldsymbol{x})) \cdots)$ の極限的振る舞い

$$\lim_{k\to\infty} F_{i_k} \circ F_{i_{k-1}} \circ \ldots F_{i_1}(\boldsymbol{x}) \tag{3.4}$$

を考えます。各関数 F_i が縮小写像である場合がたいへん興味深く、このとき関数の集まり $\{F_1, \ldots, F_M\}$ を**反復関数系** (IFS: iterated function system) をなすと呼ばれます。n 次元空間の写像 $F : \mathbb{R}^n \to \mathbb{R}^n$ が**縮小写像** (contractive mapping) であるとは、2 点 $\boldsymbol{x}, \boldsymbol{y}$ 間の距離 $d(\boldsymbol{x}, \boldsymbol{y})$ が写像した後に縮まる

$$d(F(\boldsymbol{x}), F(\boldsymbol{y})) < d(\boldsymbol{x}, \boldsymbol{y})$$

ような写像です。

3.3.1 反復写像系が定める不変集合

縮小写像の集まり $\{F_1, \ldots, F_M\}$ で定まる IFS において、関係式

$$\mathcal{K} = F_1(\mathcal{K}) \cup \cdots \cup F_M(\mathcal{K}) \tag{3.5}$$

を満たす不変集合 $\mathcal{K}$ が存在するというコラージュ定理が知られています [18]。$\mathcal{K}$ は各縮小写像の像 $F_i(\mathcal{K})$ の貼り合わせとなっており、一般に複雑な構造を持つフラクタル集合となります。

縮小写像の集まりが定める不変集合 $\mathcal{K}$ は例外的な点を除いたほとんどすべての（a.e: almost all）点 $\boldsymbol{x}_0$ から縮小写像をランダムに選んで反復した軌道に次のような極限集合として関連付けることができます。

$$\mathcal{K} = \omega(x, \{F_1, \ldots, F_M\}) \equiv \bigcap_{n=1}^{\infty} \overline{\bigcup_{k=n}^{\infty} F_{i_k} \circ F_{i_{k-1}} \circ \ldots F_{i_1}(\boldsymbol{x})} \tag{3.6}$$

ここで、$\overline{X}$ は集合 X に関する閉包です。この $\mathcal{K}$ を関数の集まり $\{F_1, \ldots, F_M\}$ に関する**ω-極限集合** (omega limit set) $\omega(x, \{F_1, \ldots, F_M\})$ といいます（式 (3.14) 参照）。

3.3.2 IFS の不変集合のプロット

IFS の不変集合 $\mathcal{K}$ を関係式 (3.5) の ‘解’ として求めることは難しいのですが、ω-極限集合として近似計算することは容易です。IFS が縮小写像の集まりであることから、適当な初期点 $\boldsymbol{x}_0$ を与えるたとき十分大きな回数ランダム反復を繰り返した軌道点列

$$F_{i_N} \circ F_{i_{N-1}} \circ \cdots F_{i_m} \underbrace{F_{i_{m-1}} \circ \cdots \circ F_{i_1}}_{m-1 \text{ 回の推移的反復}} (\boldsymbol{x}_0) \tag{3.7}$$

から、点 $F_{i_m} \circ \cdots \circ F_{i_1}(\boldsymbol{x}_0)$ に達する前の（初期点 $\boldsymbol{x}_0$ を含む）m 個 $(N \gg m)$ の推移的軌道点列を取り除くと、残った点列の多くが不変集合 $\mathcal{K}$ のごく近くで挙動していると考えることができます。つまり、目的とする集合 $\mathcal{K}$ は、初期点 $\boldsymbol{x}_0$ を適当に選んで IFS を構成する縮小写像からランダムに関数を選んで反復適用して得られる 1 本の軌道列から先頭の m 点を省いた点列として近似計算できることになります。

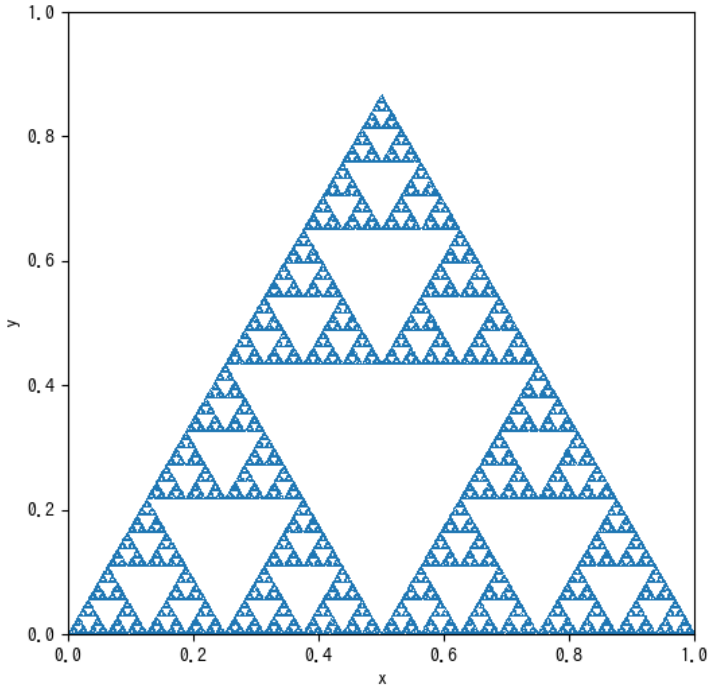


図 3.5　Sierpinski の三角形集合 $\mathcal{S}$。3 つの縮小写像 S_1, S_2, S_3 を一様ランダムに選んで初期値 $(0.0, 0.0)$ から 100,000 回反復し、最初の推移軌道を取り除いたプロット結果（プログラム 3.3-1: `sierpinski_IFS.py`）。

たとえば、縮小写像系として平面上の次の 3 つの関数からなる関数系（シェルピンスキー写像系）を考えます。

$$S_1(x,y)=\left(\frac{x}{2}+\frac{1}{4},\frac{y}{2}+\frac{\sqrt{3}}{4}\right),\quad S_2(x,y)=\left(\frac{x}{2}+\frac{1}{2},\frac{y}{2}\right),\quad S_3(x,y)=\left(\frac{x}{2},\frac{y}{2}\right). \tag{3.8}$$

プログラム 3.3-1: `sierpinski_IFS.py` は、関数 `sierpinski(x, k)` で $k=0,1,2$ に応じた縮小シェルピンスキー関数 S_1,S_2,S_3 で定まる不変集合に十分近い集合をプロットします（図 3.5）。`numpy.random.randint(0,3)` で一様整数乱数 0,1,2 を発生させてシェルピンスキー関数をランダムに選び出し、初期点 `x0` から `times` 回反復して得られる軌道点列 `norbit` を計算し、推移軌道として最初の `transit` 個を取り除いて残った点列をプロットしています。平面上の適当な初期点から出発した軌道は縮小写像 S_1,S_2,S_3 で定まる不変集合 $\mathcal{S}$ に急速に引き込まれてしまいます。

この $\mathcal{S}$ は**シェルピンスキーの三角形** (Sierpiński triangle) と呼ばれる自己相似フラクタルで、面積ゼロの全非連結集合になっています。

コード 3.3-1　sierpinski_IFS.py

```
import numpy as np
import matplotlib.pyplot as plt

def sierpinski(x: np.ndarray, k: int) -> np.ndarray:
    if k == 0:
        x1 = 0.5 * x[0] + 0.25
        y1 = 0.5 * x[1] + np.sqrt(3)/4
    elif k == 1:
        x1 = 0.5 * x[0] + 0.5
        y1 = 0.5 * x[1]
    elif k == 2:
        x1 = 0.5 * x[0]
        y1 = 0.5 * x[1]
    return(np.array([x1, y1]))

times = 100000 # プロット回数
transit = 100 # 推移回数
xini, yini = 0.5, 0.5
x0 = np.array([xini, yini])  # 初期点

norbit = np.empty((times,2))
for k in range(times):
```

```
    norbit[k] = x0
    x0 = sierpinski(x0, np.random.randint(0,3))

fig, ax = plt.subplots(figsize=(6.0, 6.0))
ax.plot(norbit[transit:,0], norbit[transit:,1], ',', markersize=1)
ax.set(xlabel='x', ylabel='y', xlim = (0, 1), ylim = (0, 1))
#fig.savefig('sierpinski_IFS.png')
fig.show()
```

3.3.3 脱出時間アルゴリズム

今、平面上の写像として $f_S : \mathbb{R}^2 \to \mathbb{R}^2$

$$f_S(x,y) = \begin{cases} (2x - \frac{1}{2}, 2y - \frac{\sqrt{3}}{2}), & y \geqq \sqrt{3}/4 \text{ のとき} \\ (2x-1, 2y), & x \geqq \frac{1}{2} \text{ かつ } y < \sqrt{3}/4 \text{ のとき} \\ (2x, 2y), & \text{それ以外} \end{cases} \tag{3.9}$$

を考えてみましょう。写像 f_S は式 (3.8) の縮小写像系を成す S_1, S_2, S_3 で定まるシェルピンスキーの三角形 $\mathcal{S}$ 上にその定義域を制限した時には

$$f_S(x,y)\Big|_{\mathcal{S}} = \begin{cases} S_1^{-1}(x,y), & (x,y) \in S_1(\mathcal{S}) \text{ のとき} \\ S_2^{-1}(x,y), & (x,y) \in S_2(\mathcal{S}) \text{ のとき} \\ S_3^{-1}(x,y), & (x,y) \in S_3(\mathcal{S}) \text{ のとき} \end{cases}$$

の関係にあります。このため、f_S を拡大的シェルピンスキー写像ということにします。領域 $\mathcal{S}$ 上の力学系 $(\mathcal{S}, f_S)$ において、$\boldsymbol{x}_0 \in \mathcal{S}$ から出発する軌道 $\mathcal{O}(\boldsymbol{x}_0) = \{\boldsymbol{x}_n = {F_S}^n(\boldsymbol{x}_0)\}_{n=0}^{\infty}$ は全非連結集合 $\mathcal{S}$ 上を移動するため、点 $\boldsymbol{x}_n$ がシェルピンスキー三角形のどの部分三角形 $\mathcal{S}_i \equiv S_i(\mathcal{S})$ に属しているかを表せるように $\boldsymbol{x}_n \in \mathcal{S}_i$ のとき記号 $i(=1,2,3)$ を対応させます。この対応によって、力学系 $(\mathcal{S}, f_S)$ の軌道 $\mathcal{O}(x_0)$ は記号列 $\epsilon_0\epsilon_1\epsilon_2\ldots(\epsilon_i \in \{1,2,3\})$ で記述できることになります。こうした理由から、$\mathcal{S}$ 上の写像 f_S を IFS(3.8) に対する**シフト写像** (shift map) と呼ぶことがあります。

写像 f_S は $f_S(\mathcal{S}) = \mathcal{S}$ とシェルピンスキーの三角形集合を不変 に保ちますが、明らかに f_S は $\mathbb{R}^2$ 上で拡大的 (expansive) であるので $\mathcal{S}$ は F_S の反発的不動集合 (repeller) です。$\mathcal{S}$ 上の点 $\boldsymbol{x}_S \in \mathcal{S}$ に対して $\{{f_S}^n(\boldsymbol{x}_S)\} \in \mathcal{S}$（$\mathcal{S}$ から出発した軌道は $\mathcal{S}$ に留まる）ですが、$\boldsymbol{x}_0 \notin \mathcal{S}$ に対して軌道 ${f_S}^n(\boldsymbol{x}_0)$ は無限遠に遠ざかります（無限遠に引き込まれると考えることができます)。

一般に平面上の力学系 $(\mathbb{R}^2, f)$ あるいは複素平面 $\mathbb{C}^2$ 上の力学系 $(\mathbb{C}^2, f)$ において、ある有界領域 $\mathcal{W}$ 内の点 $\boldsymbol{x}_0 \in \mathcal{W}$ が写像 f の反復 $f^n(\boldsymbol{x}_0)$ によってどのように無限遠に遠ざかるかの様子を調べることは「ジュリア集合とマンデルブロー集合」（節 3.4）で見るようにたいへん大きな意味があります。

脱出時間アルゴリズム (ETA: Escape Time Algorithm) [18] とは、$\mathbb{R}^2$ 内の領域 $\mathcal{W}$ 内の各点について、写像の反復によって無限遠に至る写像の反復回数（脱出数）を割当て、脱出数によって $\mathcal{W}$ を塗り分ける方法です。ある点 $\boldsymbol{x}$ を初期条件とする軌道点列が有界に留まる（$\{|f^n(x)| < +\infty\}_{n\in\mathbb{N}}$）とき、$\boldsymbol{x}$ の脱出数は無限大になります。実際に計算では、計算を打ち切るために前もって反復数の最大値 K_{max} を決めておきます（プログラム 3.3-2）。

まず、原点から限界半径 R 以上離れた領域を無限遠を含ませて脱出領域 $\mathcal{V}$ として

$$\mathcal{V} = \{\boldsymbol{x} \in \mathbb{R}^2 \,|\, |\boldsymbol{x}| > R\} \cup \{\infty\}$$

で定義します。ただし、$\mathcal{V}$ に属する点 $\boldsymbol{x}_\infty \in \mathcal{V}$ は f の反復によって確実に発散する（無限遠に引き込まれる）ように限界半径 $R > 0$ が選ばれているようにしておきます。実際、写像 f が 1 次以上の多項式であればそのような $R > 0$ が存在します。

有界領域 $\mathcal{W}$ を脱出半径内に取ります（$\mathbb{R}^2 \backslash \mathcal{V}$）。このとき、$\mathcal{W}$ の各点 $\boldsymbol{x} \in \mathcal{W}$ に対して、次のようにして脱出数 $E(\boldsymbol{x}) \in \{1, \ldots, K_{max}\}$ を割り当てることができます。

$$E(\boldsymbol{x}) = \min\{f^k(\boldsymbol{x}) \in \mathcal{V} \text{ となる最小の } k,\ K_{max}\}.$$

ここで、$K_{max} > 0$ はあらかじめ定めておいた反復回数の最大値です。この割当手続きでは有界軌道列を有する点の脱出数は最大値の K_{max} に抑えられます。脱出数が無限大であるような点は無限大に引き込まれるようなっことがなく、その軌道は有界に留まります。あらかじめ決めておく反復回数の最大値 K_{max} は脱出領域 $\mathcal{V}$ の半径 R との兼ね合いで決めますがある程度大きめに設定しておくと無難です（その代わりに脱出数の算出のための計算量は増加します）。

脱出時間アルゴリズムを式 (3.9) の拡大的シェルピンスキー写像 f_S に適用してみましょう。f_S は $\mathcal{S}$（面積ゼロの全非連結集合）以外の点は反復によって無限遠に発散してしまい、有界軌道列を有する点は $\mathcal{S}$ 上以外にはありません。図 3.6 はシェルピンスキーの三角形 $\mathcal{S}$ を囲む領域をプログラム 3.3-2: `sierpinski_ETA.py` に従って脱出値によって塗り分けしたものです。

プログラム 3.3-2: `sierpinski_ETA.py` では、領域 $\mathcal{W}$ の各点を `np.meshgrid` で生成した格子行列 `xg` と `yg` で表される格子点としています。各格子点の脱出時間を格納する `escapeMatix` は、まず格子点数 `gridNum = len(xg.ravel())` を持つ一次元配列として `escapeMatix = np.zeros((gridNum,))` とゼロに初期化してお

き、格子点を `x = [xg.ravel()[i], yg.ravel()[i]]` と 1 次元化して調査して、その脱出時間を計算しています。各点の脱出時間を計算した後、`escapeMatix = np.reshape(escapeMatix, xg.shape)` と格子配列に reshape してから `imshow` を使って各点の脱出時間を塗り分けています。

反発不変集合となっているシェルピンスキーの三角形 $\mathcal{S}$ に近い点ほど脱出数は大きくなっています。集合 $\mathcal{S}$ は最大反復数 K_{max} を脱出数とする有領域に含まれるようになっています。言い換えれば、最大反復数を持つ点をプロットするとシェルピンスキーの三角形（の近似集合）を描くことができることになります。

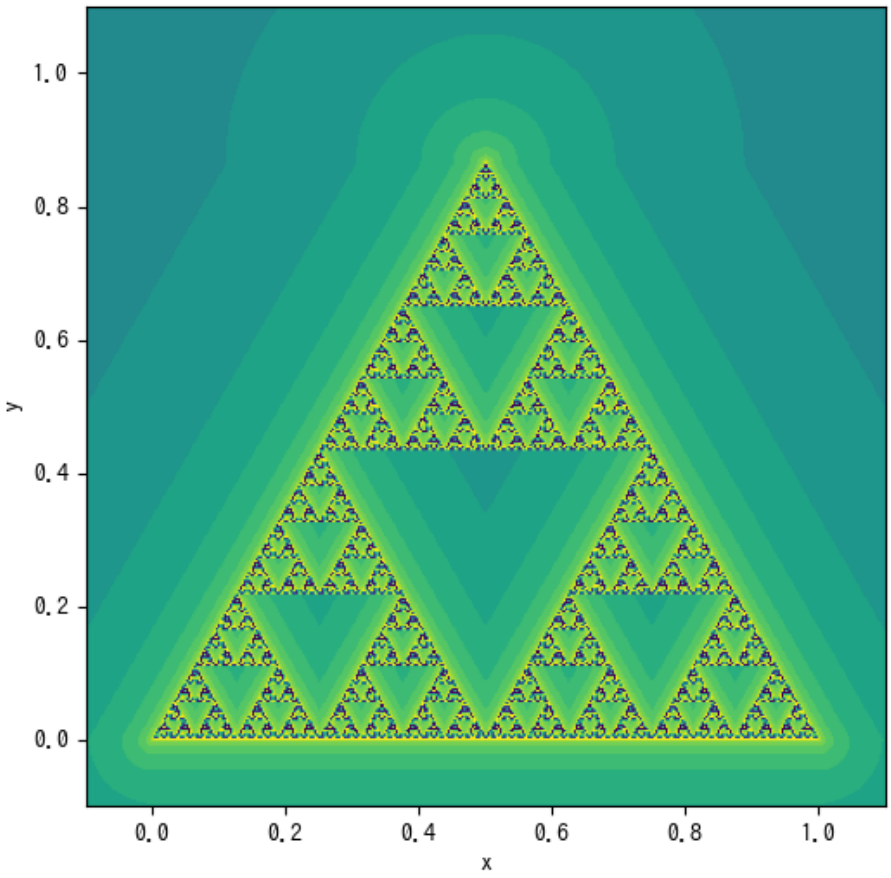


図 3.6　式 (3.9) の拡大シェルピンスキー写像 f_S における脱出数による塗り分け。脱出時間アルゴリズムを使って、最大反復数を脱出数とする点をプロットすればシェルピンスキーの三角形（の近似集合）を描くことができる。

コード 3.3-2　sierpinski_ETA.py

```
import numpy as np
import matplotlib.pyplot as plt

def ex_sierpinski(x: np.ndarray) -> np.ndarray:
    if x[1] >= np.sqrt(3)/4:
        return np.array([2*x[0] - 0.5, 2*x[1] - np.sqrt(3)/2])
    elif x[0] >= 0.5 and x[1] < np.sqrt(3)/4:
        return np.array([2*x[0] - 1, 2*x[1]])
    else:
        return np.array([2*x[0], 2*x[1]])

xrange = (-0.1, 1.1)
yrange = (-0.1, 1.1)
matrixSize = (400, 400)
maxRadius = 200
maxStep = 20

xg, yg = np.meshgrid( # 格子点のx,y成分行列 xg, yg
    np.linspace(*xrange, matrixSize[0]),
    np.linspace(*yrange, matrixSize[1]))
gridNum = len(xg.ravel())

escapeMatix = np.zeros((gridNum,))# shape=xg.ravel().shape
for i in range(gridNum):
    depth = 0
    x = [xg.ravel()[i], yg.ravel()[i]]
    while np.linalg.norm([x]) <= maxRadius and not depth == maxStep:
        x = ex_sierpinski(x)
        depth += 1
    if depth == maxStep:
        escapeMatix[i] = 0
    else:
        escapeMatix[i] = depth
escapeMatix = np.reshape(escapeMatix, xg.shape)

fig, ax = plt.subplots(figsize=(6.0, 6.0))
ax.imshow(escapeMatix / maxStep, origin='lower',
```

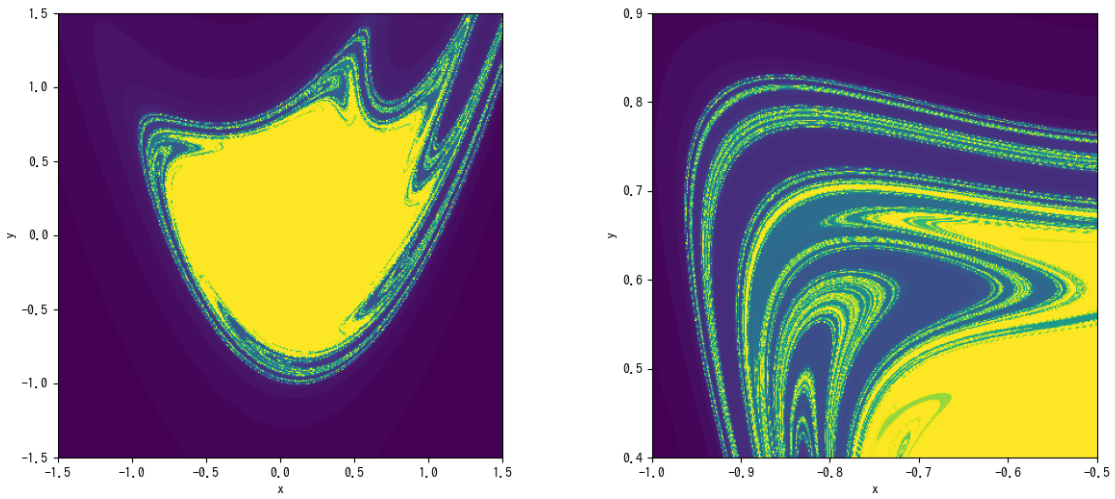
```
            extent=[xrange[0], xrange[1], yrange[0],yrange[1]])
ax.set(xlabel='x', ylabel='y')
#fig.savefig('Sierpinski_ETA.png')
fig.show()
```

3.4　ジュリア集合とマンデルブロー集合

3.4.1　ジュリア集合

図 3.7 は、節 3.2.3 で取り上げた 2 次多項式のエノン写像 (3.3) を複素平面上の写像として再定義し、プログラム 3.4-1: `julia.py` と同様にして脱出時間アルゴリズム（節 3.3.3）の計算を高速化してプロットした原点周辺領域の脱出数による塗り分けです。

原点を含む黄色で塗りつぶされた領域は反復によって無限遠に吸引されずに有界に留まる点領域ですが、その境界は無限大の脱出回数を持ち、その外側は同じ脱出回数を持つ等脱出回数領域の層になっており、複雑な微細構造を持っていることが観察できます。



(a) $a = 3.84$　　(b) $a = 3.86$

図 3.7　エノン写像 (3.3)（$a = \cos^{-1} 0.24$）における原点まわりの矩形領域の脱出数による塗り分け。複素平面上の写像として、`np.where` を使って計算を高速化したプログラム 3.4-1: `julia.py`)。(a) 領域 $[-1.5, 1.5] \times [-1.5, 1.5]$, (b) 領域 $[-1.0, -0.5] \times [0.4, 0.9]$. 原点の周りの点の軌道が有界に留まるか無限に発散するかの境目は複雑な微細構造を有している。

格子領域の各点を複素数（スカラー）とする写像を `mapping(z, para)` と定義してお

くと、各格子点の脱出回数のカウントと最大脱出数の判定・更新を NumPy 配列を構成する各要素の条件を一気に判定して置換などの処理を実行できる NumPy 関数 np.where を使うことができ、計算時間の大幅な短縮が可能になります。

```
escapeMatix = np.zeros(Z.shape, dtype=int)# 反復回数行列の全要素を 0 で初期化
for i in range(maxStep):
    Z = np.where(np.abs(Z) <= maxRadius, mapping(Z), Z)
    escapeMatix += np.abs(Z) <= maxRadius # 対応する脱出回数行列要素を+1 更新
```

矩形領域の各点の脱出回数行列 escapeMatix を 0 に初期化しておいて、矩形領域の各点 $z \in Z$ が限界半径 maxRadius の円内部にあれば z を mapping(z, para) で更新し、限界円の外部にあるときは z はそのままにします。そして、脱出回数が maxRadius 以内ならば $+1$ していきます。

コード 3.4-1　julia.py

```
import numpy as np
import matplotlib.pyplot as plt

def quadra_map(z: complex, c: complex) -> complex:
    return (z ** 2 + c)

#複素パラメータc
c = 1.0 + 0.0 * 1j
#c = -0.12256117 + 0.74486177 * 1j # Douady's Rabbit
matrixSize = (700, 700)
maxRadius = 200
xRange = (-1.7, 1.7)
yRange = (-1.1, 1.1)
maxStep = 40
xRe, yIm = np.meshgrid( # パラメータ範囲の格子点グリッド
        np.linspace(*xRange, matrixSize[0]),
        np.linspace(*yRange, matrixSize[1])
    )
Z = xRe + yIm * 1j # 矩形の複素領域

escapeMatix = np.zeros(xRe.shape, dtype=int)# 反復回数行列の全要素を0で初期化
for i in range(maxStep):
    Z = np.where(np.abs(Z) <= maxRadius, quadra_map(Z, c), Z)
    escapeMatix += np.abs(Z) <= maxRadius # 対応する反復行列を +1更新
```

```
fig, ax = plt.subplots(figsize=(6.0, 6.0))
ax.imshow(escapeMatix / maxStep, origin='lower',
          extent=[xRange[0],xRange[1], yRange[0],yRange[1]])
ax.set(xlabel='z.re', ylabel='z.im')
#fig.savefig('julia.png')
fig.show()
```

図 3.7 において、原点の近くの（黄色く）塗りつぶされている領域（充填ジュリア集合）から出発した軌道は無限遠に発散せずに有界に留まります。しかし、軌道が無限遠に発散していく脱出数を持つ領域は複雑な微細構造を持つ境域を取り囲んでいることがわかります。この興味深い状況は一般的で、$\hat{\mathbb{C}} = \mathbb{C} \cup \{\infty\}$ 上の複素力学系 $(\hat{\mathbb{C}}, f)$ の枠組みで深く研究されてきました。

$\hat{\mathbb{C}}$ 上の正則関数として、ここでは簡単にパラメータ λ を持つ

$$f_\lambda(z) = z^2 + \lambda, \qquad \lambda \in \mathbb{C} \tag{3.10}$$

を考えます。任意の 2 次写像 $R(z) = az^2+2bz+d$ は、Möbius 変換として $M(z) = az+b, c = ad+b-b^2$ について $M^{-1} \circ f_c \circ M(z) = M^{-1}((az+b)^2+c) = \frac{(a^2z^2+2abz+b^2+c)-b}{a} = R(z)$ であることから $f_c(z)$ と共役であることがわかります。このことから、$z \mapsto z^2 + \lambda$ の形のクラスを考えればすべての 2 次式の力学系を理解できることになります。

$\lambda = 0$ の場合は単純です。単位円盤 $F = \{z \in \mathbb{C} \,|\, |z| \geqq 1\}$ の補集合にあるすべての点は f_0 の反復によって無限遠に発散し、D の内部の点は原点に収束します。したがって、F を取り囲む十分大きい半径 R の領域 $\mathcal{V}$ は、脱出時間アルゴリズムによって F を同心円上に色付けされます。反復軌道点列が有界に留まる集合 F の境界 ∂F は単位円になっており無限遠に発散しません（その外側の点は無限遠に発散します）。$\lambda \neq 0$ の場合、この状況はどうなるでしょうか。

1 次以上の有理関数からなる複素力学系 $(\hat{\mathbb{C}}, f)$ において次のような集合を考えることができます。

$$F_f = \{z \in \mathbb{C} \,|\, \text{点列 } \{|f^n(z)|\}_n = 0^\infty \text{ は有界}\} \tag{3.11}$$

F_f を f に関する**充填ジュリア集合** (filled Julia set) 、その境界 $J_f = \partial F_f$ を**ジュリア集合** (Julia set) といいます（ジュリア集合の補集合 F_f を**ファトゥ集合** (Fatou set) ということもあります）。力学系の用語を使うと、ジュリア集合はすべての反発的周期点の閉包であり、その補集合であるファトゥ集合はアトラクタに引き込まれていく領域（吸引の

鉢）になっていて、ジュリア集合 J_f およびファトゥ集合 F_f は写像 f に関する不変集合 $f(J_f) = J_f = f^{-1}(J - f), f(F_f) = F_f = f^{-1}(F_f)$ です。

$f(z) = z^2$ の場合、単位円の内部は原点に、また単位円の外部は無限遠に引き込まれることから、ファトゥ集合は吸引の鉢としてこれら領域を併せた集合になっています。ファトゥ集合の補集合であるジュリア集合は単位円です。実際、任意の n について単位円上に等しい角度で反発的 n 周期点が載っており、すべての反発的周期点の閉包が単位円になっています。

ジュリア集合は常に非空集合で一般には滑らかでなく、ほとんどの場合、フラクタル的微細構造を持ちます。図 3.8 は、式 (3.10) において $\lambda = -1$ のとき、脱出時間アルゴリズムをプログラム 3.4-1 を使って計算した充填ジュリア集合（黄色領域）です。充填ジュリア集合は等脱出時間集合を成す円環列で取り囲まれ、このことより充填ジュリア集合の補集合（以下で定義される $\mathcal{V}_\infty$）は連結であることが観察できます。

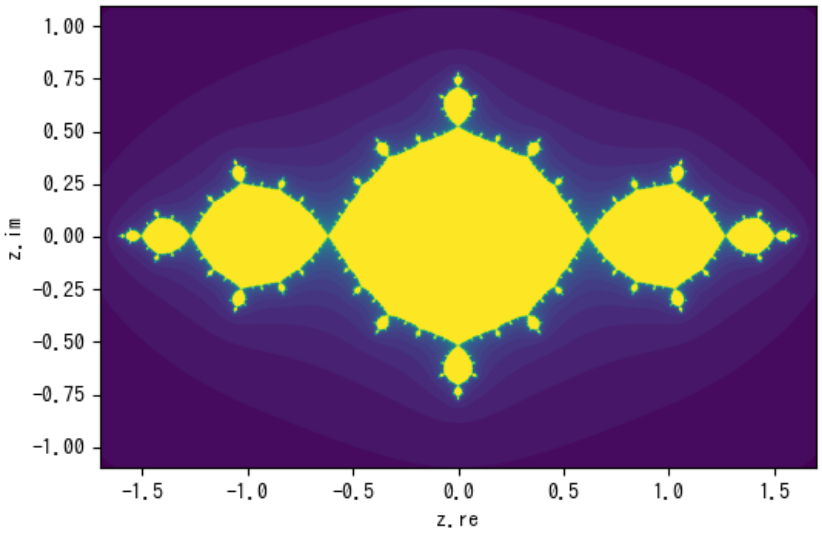


図 3.8　充填ジュリア集合（黄色領域の境界）。$z \mapsto z^2 - 1$ での脱出時間アルゴリズム（節 3.3.3）による計算。充填ジュリア集合を取り囲む等脱出時間集合の列の極限 $\mathcal{V}_\infty$）は連結（定理 3.1）。

一般に $\hat{\mathbb{C}}$ 上の写像 f に対して、

$|f(z)| > |z|$　　$|z| > R$ となるすべての z について

となって無限遠に吸い込まれてしまう領域を定める限界半径 R を見つけることができます。そこで、f の反復によって確実に無限遠に引き込まれる脱出領域 $\mathcal{V}$ を無限遠を含めて

$$\mathcal{V} = \{x \in \mathbb{C} \,|\, |z| > R\} \cup \{infty\} \tag{3.12}$$

と定義します。すると

$$f(\mathcal{V}) \subset \mathcal{V}$$

となることより、次が成立します。

$$\mathcal{V} = \mathcal{V}_0 \subset \mathcal{V}_1 \subset \mathcal{V}_2 \subset \cdots \subset \mathcal{V}_n \subset \ldots$$

脱出時間アルゴリズムによって、無限遠に引き込まれる脱出領域 $\mathcal{V}$ は 1 つ脱出回数が大きい円環状の等脱出領域 $\mathcal{U}_1 = \mathcal{V}_1 \backslash \mathcal{V}_0$ によって囲まれ、$\mathcal{U}_1$ はさらに 1 つ大きな脱出回数が大きい等脱出領域 $\mathcal{U}_2 = \mathcal{V}_2 \backslash \mathcal{V}_2$ によって囲まれ $\ldots$ と続き、$\mathcal{V}$ は徐々に間隔が狭くなるなる円環領域列 $\mathcal{U}_1, \mathcal{U}_2, \ldots$ による脱出時間等高線で囲まれています。その極限 $\mathcal{U}_\infty$ が脱出速度が無限大となって無限遠に吸い込まれない充填ジュリア集合 F_f を包む薄皮になって、$F_f = \hat{\mathbb{C}} \backslash \mathcal{V}_\infty$ となります。

これらの議論から、次の定理が成立します [18]。ここで、弧状連結集合 X とは、X 内のに似の 2 点 $x, y \in X$ を $c(0) = x, c(1) = y$ で結ぶ連続曲線 $c : [0, 1] \to X$ が存在することです。

定理 3.1 $f : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ を 1 次以上の正則有理関数とする。このとき、充填ジュリア集合 F_f とジュリア集合 J_f は $\mathbb{C}$ の非空集合であって f-不変：$f(J_f) = J_f = f^{-1}(J_f)$ おとび $f(F_f) = F_f = f^{-1}(F_f)$. さらに、無限遠へ吸い込まれる吸引鉢 $\mathcal{V}_\infty = \hat{\mathbb{C}} \backslash F_f$ は弧状連結集合である。

3.4.2 マンデルブロー集合

写像 (3.10)

$$f_\lambda(z) = z^2 + \lambda, \qquad \lambda \in \mathbb{C} \tag{3.10}$$

において、パラメータ変化によるジュリア集合の様子は多彩な変化を呈し、その数学的構造を巡って精緻な研究が続けられています [19]。

$\lim_n f_\lambda{}^n(0) \to \infty$ であるときジュリア集合 J_{f_λ} は全非連結集合に、$\lim_n f_\lambda{}^n(0)$ が有界のときにはジュリア集合は連結になることがわかっています [30][13]。パラメータ変化に応じたジュリア集合のグラフ $(\lambda, J_{f_\lambda}) \in \mathbb{C} \times \mathbb{C}$ を視覚化することは簡単ではありません。

マンデルブローは式 (3.10) におけるパラメータ集合

$$M = \{\lambda \in \mathbb{C} \,|\, \lim_n f_\lambda{}^n(0) \text{ が有界}\} \tag{3.13}$$

を考えました。これを**マンデルブロー集合** (Mandelbrot set) と呼んでいます [19]。先の注意から、ジュリア集合が連結になるパラメータ集合をマンデルブロー集合と呼ぶことができます。

図 3.9 に、脱出時間アルゴリズムを使ったプログラム 3.4-2 で描いたマンデルブロー集合（黄色い領域）を示しました。パラメータを拡大してもその微細構造は続きます [19]。

マンデルブロー集合 M については 1980 年以降に急激に研究が進みました。その成果として次の驚くべき定理があります [30]。

定理 3.2 (Douady and Hubbard) M は閉じた単純連結集合である。

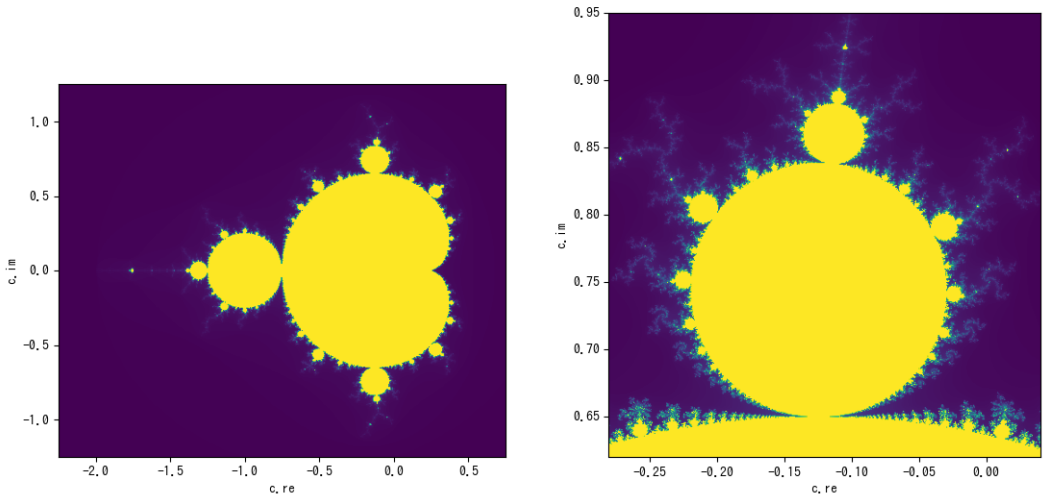


図 3.9 $f(z) = z^2 + c$ $(c = c.\mathrm{re} + i\,c.\mathrm{im} \in \mathbb{C})$ **のマンデルブロー集合 (a) パラメータ** c **の範囲** $c.\mathrm{re} \in [-2.25 - 0.75], c.\mathrm{im} \in [-1.25 - 1.25]$**. (b) パラメータ** c **の範囲** $c.\mathrm{re} \in [-0.28, -0.04], c.\mathrm{im} \in [0.62, -0.95]$**.**

コード 3.4-2　mandelbrot.py

```
import numpy as np
import matplotlib.pyplot as plt

def quadra_map(z: complex, c: complex) -> complex:
    return (z ** 2 + c)

#複素パラメータcの範囲
cRealRange = (-2.25, 0.75) # 図(a)
cImagRange = (-1.25, 1.25)
#cRealRange = (-0.28, 0.04) # 図(b)
#cImagRange = (0.62, 0.95)
matrixSize = (500, 500)
maxIteration = 300
DH_Radius = 2 # Douady-Hubbard半径
```

```
cRe, cIm = np.meshgrid( # パラメータ範囲の格子点グリッド
        np.linspace(*cRealRange, matrixSize[0]),
        np.linspace(*cImagRange, matrixSize[1])
    )
C = cRe + cIm * 1j # 矩形パラメータ領域
Z = np.zeros_like(C) # 初期条件 z_0 = 0 + i0 とする
iteratedMatrix = np.zeros(C.shape, dtype=int)# 反復回数行列の全要素を0で初期化

for i in range(maxIteration):
    Z = np.where(np.abs(Z) <= DH_Radius, quadra_map(Z, C), Z)
    iteratedMatrix += np.abs(Z) <= DH_Radius # 対応する反復行列を +1更新



fig, ax = plt.subplots(figsize=(6.0, 6.0))
ax.imshow(iteratedMatrix / maxIteration, origin='lower',
          extent=[cRealRange[0],cRealRange[1], cImagRange[0],cImagRange[1]])
ax.set(xlabel='c.re', ylabel='c.im')
#fig.savefig('mandelbrot.png')
plt.show()
```

3.5 ロジスティック写像

式 (3.1) で紹介した区間 $[0,1]$ 上のロジスティック写像 f_a を改めて考えてみましょう。

$$f_a(x) = ax(1-x), \qquad x \in [0,1], \quad 0 < a \leqq 4 \tag{3.1′}$$

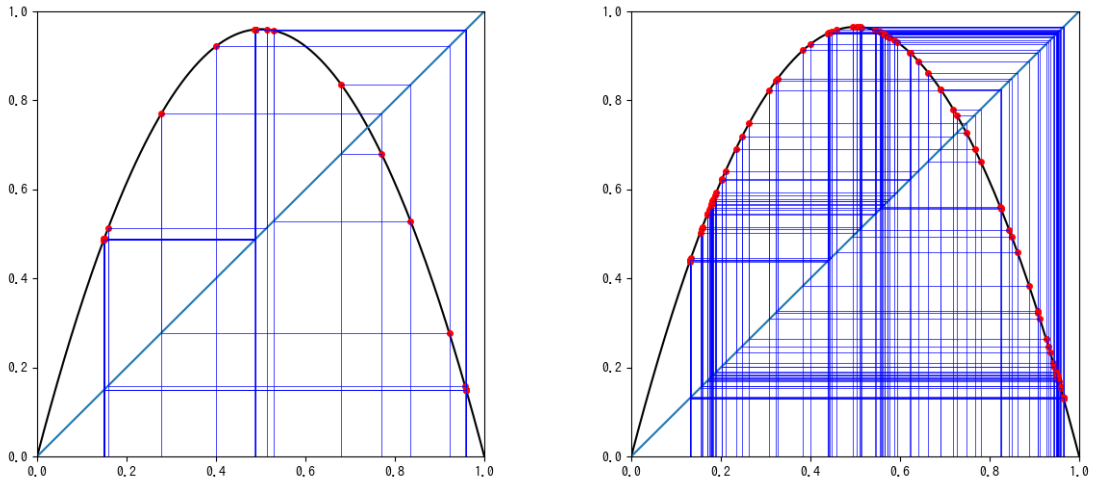
初期値 $x_0 \in [0,1]$ を選び f_a による反復 $f_a^n(x_0)$ によって生成される軌道 $\mathcal{O}_a(x_0)$ は、パラメータ a の選び方によって、その挙動は大変複雑に変化することがわかっています。

1 次元離散力学系の様子を視覚的に表す方法として、節 3.5.1 の蜘蛛の巣図と節 3.5.2 の分岐ダイヤグラムの方法があります。力学系を特徴づける量として節 3.5.4 で**リャプノフ指数**があります [17]。

3.5.1 クモの巣図

1 次元力学系では、その軌道を計算して追跡するだけでは様子がつかめません。そこで、初期点 x_0 から f_a によって次々と区間上で写される有り様を図 3.1 で見たように**クモの巣図** (cobweb plot) として表すことがあります [17, p.5]。

まず、x, y-平面で f_a で移される初期点 x_0 を x-軸上に $(x_0, 0)$ と置いて関数のグラフ値 $(x_0, f_a(x_0))$ と線で結び、写された $f_a(x_0$ を x-軸上に見出すために、そこからの水辺に対角線 $y = x$ とぶつかる点 $(f_a(x_0), f_a(x_0))$ まで線を引いて、そこから x-軸におろした点 $(f_a(x_0), {f_a}^2(x_0))$ まで線を引きます。この操作を軌道点列 $f_a^n(x_0)$ に施して得られる連続な渦巻状の折れ線がクモの巣図です。



(a) $a = 3.84$　　(b) $a = 3.86$

図 3.10　ロジスティック写像のクモの巣図。ロジスティック写像 $f_a(x) = ax(1-x)$ の初期点 $x_0 = 0.4$ から出発した 100 回反復の蜘蛛の巣図。(a) パラメータ $a = 3.84$. 蜘蛛の図はアトラクタとなった 3 周期点 0.14940700876432014, 0.488004689266247, 0.9594474720783386 に巻き込まれていく。(b) パラメータ $a = 3.86$. 1 つの軌道は区間 $[0, 1]$ 上を複雑に動き回る。

図 3.10 は、ロジスティック写像を同じ初期値 x_0 から 100 回反復した蜘蛛の巣図で、同じ初期点 x_0 からの軌道はパラメータ a の僅かな違いで劇的な変化があることが観察できます。軌道点列に対する関数のグラフ値を丸（●）で描いています。図 (a) は $a = 3.84$ と選んだときで、蜘蛛の巣はは 3 周期点 0.14940700876432014, 0.488004689266247, 0.9594474720783386 に巻き付くように引き込まれていていきます。軌道を引き込むようなそうした点集合を**アトラクタ** (attractor) と呼びます。

図 (b) は $a = 3.86$ と選んだときで、初期値 $x_0 = 0.4$ からの軌道は（すなくとも）低い周期の周期点に引き込まれるようなことがなく、区間内で複雑な挙動をしています。初期点 x_0 のごく近くの点 $x_0 + \delta$ の軌道を追跡するとき、パラメータ a の選び方によっては写像

f_a によって

$$f_a(x_0+\delta x) \approx f_a(x_0) + \frac{df_a(x_0)}{dx}\delta x$$

とその差異が $|df_a(x_0)/dx|$ 倍に拡大または縮小され、さらに反復を繰り返していくと初期点のほんのわずかな差異が反復につれて明白な違いとなってしまうことがある。そのとき、最初は隣接していた初期点からの軌道のその後の振る舞いは遂にはまったく違ったようにお互いの挙動が無相関になります。写像の定義域の “多くの領域” でこうした状況が生じるならばもはやその状況が例外とは言えません。

初期点の僅かな差異がその後の振る舞いの大きな違いとなる様子を初期条件鋭敏性 (sensitive dependence on initial conditin) と呼んでいます。節 3.5.3 で検討するように、初期条件鋭敏性を有する軌道点列は写像の反復によって過去の記憶を失っていきます。3.5.4 で紹介するリャプノフ指数は軌道の性質を計量的に特徴づけます。ある現象を実験測定したとき、その結果が毎回異なるからと言ってその現象がランダムな確率事象であるとも、測定方法・精度に問題があると必ずしも断定できません。

力学系の基本問題の 1 つに、初期点 x_0 から出発する軌道 $\mathcal{O}(x_0)$ の**漸近的な振舞い** (asymptotic behaviour) の研究があります。十分先 N 以降の点列 $x_N, x_{N+1}, x_{N+2}, \ldots$ が領域内をどのように移動するのか、それは初期点 x_0 の選び方によってどう変化するのか、どんな集合がアトラクタになっているのかなどが問題になります。

図 3.10 は、プログラム 3.5-1 `cobweb_logistic_map.py` を使って、初期点 x_0 からのクモの巣図です。

コード 3.5-1　cobweb_logistic_map.py

```
import numpy as np
import matplotlib.pyplot as plt

def logistic(x: float, a: loat) -> float:
    x1 = a * x * (1 - x)
    return(x1)

divnum: int = 200 # 分割数
points = np.linspace(0, 1, divnum)
#a = 3.84# パラメータ
a = 3.86

fig, ax = plt.subplots(figsize=(6.0, 6.0))
ax.plot(points, logistic(points, a), 'k-')# 写像のグラフ
```

```
ax.plot([0,1], [0,1],'-')# 対角線
ax.set(xlim = (0, 1), ylim = (0, 1))


iterate = 100 # 反復回数
x0 = 0.4# 初期点
for k in range(iterate): # クモの巣図
    fx = logistic(x0, a)
    ax.plot(x0, logistic(x0, a), 'r.', markersize = 8)
    f2x = logistic(fx, a)
    ax.plot([x0, x0, fx, fx], [0, fx, fx, f2x], '-b', linewidth = 0.5)
    x0 = fx
#fig.savefig('cobweb_logistic_map.png')
fig.show()
```

3.5.2 写像の ω-極限集合と分岐ダイヤグラム

ロジスティック写像 (3.1) の軌道は図 3.10 のようにパラメータ a に鋭敏に依存し、大変複雑な様相を呈することがわかりました。初期値 x_0 から出発した軌道は十分な時間な経過時間を過ぎるとある集合（周期点や連続区間など）に吸い込まれてしまったり、区分的に連続な区間を埋め尽すような振る舞いをします。

このことをはっきりさせるために、点 x_0 を出発した写像 T の軌道 $\mathcal{O}_T(x_0)$ の**漸近的挙動** (asymptotic behaviour) として、十分に大きい n 以降の点列 $x_n, x_{n+1}, x_{n+2}, \ldots$ を $N \to \infty$ を考えることがあります。写像 f の初期点 x_0 から定まる次の集合

$$\omega(f, x_0) = \bigcap_{n=0}^{\infty} \overline{\{f^k(x_0) \,|\, k \geqq n\}} \tag{3.14}$$

を f の x_0 に関する **ω-極限集合**と呼びます（節 3.3.1）。集合 $\{f^k(x_0) \,|\, k \geqq n\}$ の集合の共通部分をとることによって、途中にある推移的点 $x_0, x_1, x_2 \ldots, x_{n-1} (n \to \infty)$ が取り除いていることに注意してください。たとえば、写像 f の周期点がまわりの軌道を引き込むアトラクタになっている場合、引き込まれていく途中点列は推移的軌道となるので ω-極限集合は周期点になります。

ロジスティック写像 (3.1) のように写像がパラメータ a に依存して f_a であるとき、a ごとに定まる軌道の ω-極限集合 $\omega(T_a, x_0)$ を

$$\{a \times \omega(T_a, x_0) \,|\, a \in I_a\}$$

と表すことを**分岐ダイヤグラム** (bifurcation diagram) と呼びます。

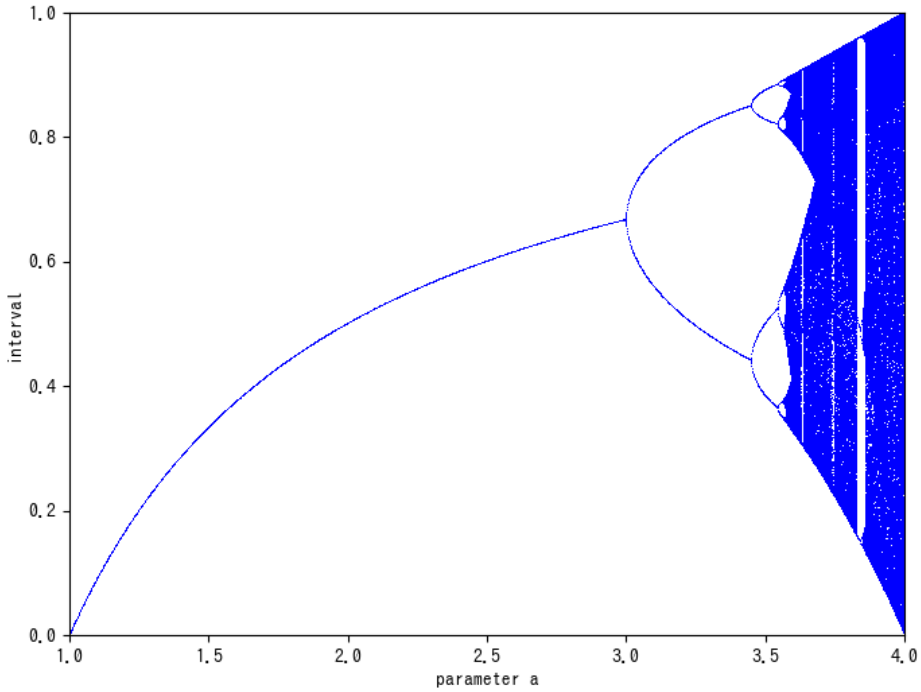
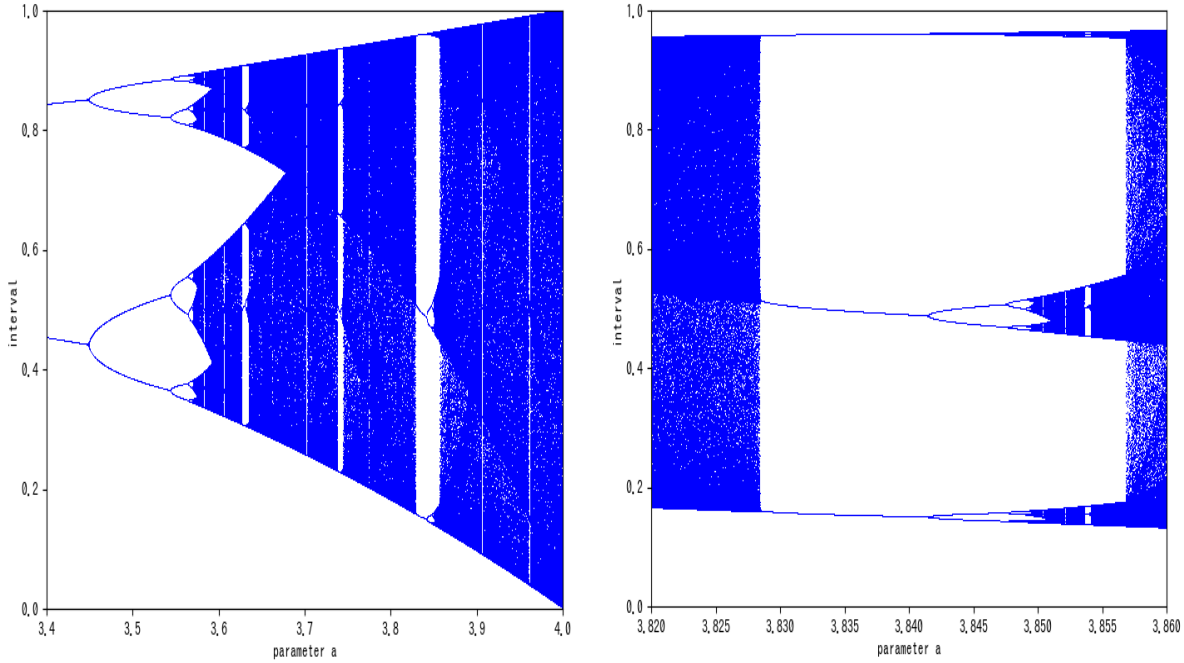


図 3.11　ロジスティック写像 (3.1) の分岐ダイヤグラム。パラメータ $1 \leqq a \leqq 4.0$ の各値で初期点 $x_0 = 0.4$ から 500 回反復させ、300 回目までを推移的点列とみなしそれ以降の 200 点を近似的な ω-極限集合として描いた（プログラム 3.5-2）。

ロジスティック写像のパラメータ区間 $I_a = [1, 4]$ の分岐ダイヤグラムは図 3.11 のように大変複雑な構造を持ちます。また、図 3.12 ではパラメータ区間を拡大し、(a) は区間 $[3.4, 4.0]$、さらに (b) は区間 $[3.82, 3.86]$ の分岐ダイヤグラムを示しました。



(a) 区間 (3.4, 4.0)　　(b) 区間 (3.82, 3.86)

図 3.12　ロジスティック写像 (3.1) の分岐ダイヤグラムの拡大図。パラメータ区間約 $3.828 \lesssim a \lesssim 3.841$ で 3 周期アトラクタを観察できる。パラメータ区間を拡大しても微細構造が続く。

ロジスティック写像の分岐ダイヤグラムはプログラム 3.5-2 `logistic_diagram.py` できました。関数 `get_orbit`) は反復写像で得られる軌道点列を返します。

ここでは、パラメータ区間の各 a ごとの写像 f_a の軌道を計算するために `get_orbit` をユニバーサル関数化して `uget_orbit` を定義しています（節 1.5)。パラメータ範囲 `arange` を与えると、`diagram = uorbit(x0, arange, iterate)` は shape (aNum,) の 1 次元配列となります（各配列要素は軌道点列です)。i 番目のパラメータ値 `arange[i]` の反復写像における推移軌道列以降の軌道点列はスライして `diagram[i][transit:]` です。

y-方向にプロットする推移軌道以降の点列の数に合わせて長さを揃えた同じパラメータ値 `arange[i]` を持つ 1 次元配列を `alist = np.full(iterate-transit, arange[i])` で用意して、分岐ダイヤグラムを作成いしています。図 3.12 はパラメー区間を拡大したダイヤグラム図です。

コード 3.5-2　logistic_diagram.py

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def logistic(x: float, a: float) -> float:
    x1 = a * x * (1 - x)
    return(x1)

def get_orbit(x0: float, a: float, iterate) -> np.ndarray:
    norbit = []
    for i in range(iterate):
       norbit.append(x0)
       x0= logistic(x0, a)
    return(np.array(norbit))

uget_orbit = np.frompyfunc(get_orbit, 3, 1) # ユニバーサル化

aStart, aEnd = 1, 4
aNum = 2000 # パラメター区間の分割数
arange = np.linspace(aStart, aEnd, aNum)
iteration = 1200 # Num of iteration
transit = 200 #transit回数
x0 = 0.4 # 初期点

diagram = uget_orbit(x0, arange, iteration)
fig, ax = plt.subplots()
for i in range(aNum):
    alist = np.full(iteration-transit, arange[i])
    ax.plot(alist, diagram[i][transit:], ',b', markersize=1)
ax.set(xlabel='parameter a', ylabel='interval',
      xlim = (aStart, aEnd), ylim = (0, 1))
fig.show()
```

3.5.3 軌道の数値精度

ロジスティック写像の分岐ダイヤグラムはパタメータ値に応じて大変複雑な ω-極限集合の構造があることがわかりました。そもそも、写像の軌道ははどの程度正しく計算されるのでしょうか。計算した数値結果はどの程度信用できるか、たいへん悩ましい問題です。

ごく簡単な場合で考えてみましょう。区間 $[0,1)$ 上の点 x を 2 倍してその小数部分 $\{2x\} = 2x - [2x]$ を返す写像 $T : [0,1) \to [0,1)$ は次のように定義されます（実数 x の x を

超えない最大の整数を $[x]$ で表すと、x の小数部分 $\{x\}$ は $x-[x]$ となります)。

$$T(x) = 2x - a(x). \tag{3.15}$$

ここで、$a(x)$ は次のような 0 または 1 の整数です。

$$a(x) = \begin{cases} 0, & x \in [0, 1/2) \\ 1, & x \in [1/2, 1). \end{cases} \tag{3.16}$$

初期点 x_0 とわずか δx だけ異なる別の初期点 $x_0{}' = x_0 + \delta x$ からの 2 つの軌道を考えると、T の反復のたびに差異は 2 倍に拡大されるために有限回で $2^n \delta x$ と無視できない程度に離れてしまいます。このような初期条件の僅かな差異が時間と共に急速に拡大する性質を**初期条件鋭敏性** (sensitive dependence on initial conditions) といいます

式 (3.15) を使うと、$x \in [0,1)$ は x から定まる 01 列 $\{a_n(x)\}$ を使って次のように 2 進展開されます。

$$\begin{aligned} x &= \frac{1}{2}\big(a(x) + T(x)\big) \\ &= \frac{1}{2}\left(a(x) + \frac{1}{2}\Big(a(T(x)) + T^2(x)\Big)\right) = \frac{a(x)}{2} + \frac{1}{2^2}\big(a(T(x) + T^2(x)\big) \\ &= \sum_{n=1}^{\infty} \frac{a_n(x)}{2^n}, \qquad a_n(x) = a(T^{n-1}(x)), \quad n \geqq 1. \end{aligned} \tag{3.17}$$

これより、区間 $[0,1)$ 内の任意の数は 2 進表示

$$x = [a_1, a_2, a_3, \dots,]_2, \qquad a_i \in \{0, 1\}$$

で表すことができます。式 (3.15) と式 (3.17) より明らかなことですが、写像 T を反復した軌道列 $\{T^n(x)\}_{n\in\mathbb{N}}$ の 2 進数表示は x の 2 進数表示から次のように定まります。

$$\begin{aligned} x &= [\underbrace{a_1, \dots, a_n}_{n-1 \text{ 個}}, \dots]_2 \\ T(x) &= [\underbrace{a_2, \dots, a_{n+1}}_{n-1 \text{ 個}}, \dots]_2 \\ &\vdots \\ T^m(x) &= [\underbrace{a_{m+1}, \dots, a_{n+m}}_{n-1 \text{ 個}}, \dots]_2 \end{aligned}$$

コンピュータにおいて初期点 x_0 を有限桁精度を持つ浮動小数点 $\tilde{x}_0$ で表して T による反復計算 $\tilde{x}_n = T^n(\tilde{x})$ を繰り返すと、初期点 $\tilde{x}_0$ が持っていたビット列情報は次々と失なわれ、真の値 $x_k = T^k(x)$ と数値計算値 $\tilde{x}_k = T^k(\tilde{x})$ とは徐々に食い違ってしまします。

こうした事情は傾き ± 2 を持つ単峰な次のテント写像においても同様です。

$$T(x) = \begin{cases} 2x, & 0 \leqq x < 1/2 \\ 2(1-x), & 1/2 \leqq x. \end{cases} \tag{3.18}$$

プログラム 3.5-3(tent_map_web.py) はテント写像の初期点 $x_{01} = \sqrt{2} - 1$ および $x_{02} = \sqrt{2} - 1.02$ から 10 回反復させたクモの巣図 3.13 のを描いてます。初期点のズレ 0.02 は写像反復のたびに 2 倍に拡大されていく様子が観察できます。同時に軌道の計算精度も写像 (3.15) と同様に劣化していきます。

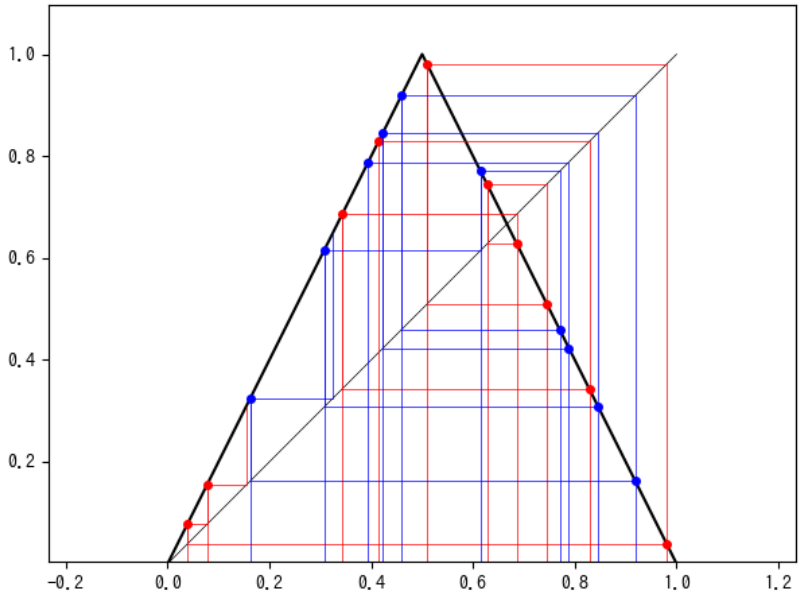


図 3.13　テント写像 (3.18) のクモの巣図。初期点 $x_{01} = \sqrt{2}-1$（赤色）および $x_{02} = \sqrt{2}-1.02$（青色）から 10 回反復させた。初期点のズレ 0.02 は写像反復のたびに 2 倍に拡大されながら、同時に軌道の計算精度も劣化していく。

コード 3.5-3　tent_map_web.py

```
import numpy as np
import matplotlib.pyplot as plt
from typing import Callable

def tent(x: float) -> float:
```

```
    if (0 <= x) and (x < 0.5):
        x1 = 2 * x
    else:
        x1 = -2 * x + 2
    return(x1)

def get_orbit(x0, mapping: Callable[[float], float], n: int)-> np.ndarray:
    orbit = []
    for k in range(n):
        orbit.append(x0)
        x0 = tent(x0)
    return(np.array(orbit))

x01 = np.sqrt(2)-1 # 初期点
x02 = np.sqrt(2)-1.02
n = 10 # 反復回数

orbit1 = get_orbit(x01, tent, n)
orbit2 = get_orbit(x02, tent, n)

fig = plt.figure()
ax = plt.axes()
ax.axis('equal')
ax.plot([0,1/2, 1], [0, 1, 0], 'k-')# 写像のグラフ
ax.plot([0,1], [0,1],'-k', linewidth = 0.5)# 対角線

utent = np.frompyfunc(tent, 1, 1)
def cobweb_plot(orbit: np.ndarray, col: str):# クモの巣図
    ax.plot(orbit, utent(orbit), '.', color = col, markersize = 8)
    for x in orbit:
        fx = tent(x)
        f2x = tent(fx)
        ax.plot([x, x, fx, fx], [0, fx, fx, f2x], '-', color=col, linewidth =
            0.5)

cobweb_plot(orbit1, 'red')
cobweb_plot(orbit2, 'blue')
ax.set(xlim = (0.0, 1.0), ylim = (0, 1.1) )#,title = 'Tent map')
```

```
#fig.savefig('tent_map.png')
fig.show()
```

3.5.4 リャプノフ指数

区間 [0.1] 上の写像 f_a において、点 $x+0$ とその近接点 $x_0+\delta x$ とは一回の反復でその差は $f_a(x_0+\delta x)-f(x_0)\approx \frac{df(x_0)}{dx}\delta x$ 程度になります。傾きの絶対値 $|df_a/dx|$ は写像 f_a による引き伸ばし情報を表しています。f_a を n 回反復したときに x_0 の周りがどの程度引き伸ばされるかは、微分の鎖規則を使って

$$L_n(x_0)\equiv\left|\frac{d^n f_a(x_0)}{dx^n}\right|=\left|\frac{df_a(x_0)}{dx}\right|\cdot\left|\frac{df_a(x_1)}{dx}\right|\cdots\left|\frac{df_a(x_{n-1})}{dx}\right|,\quad x_{i+1}=f_a(x_i)$$

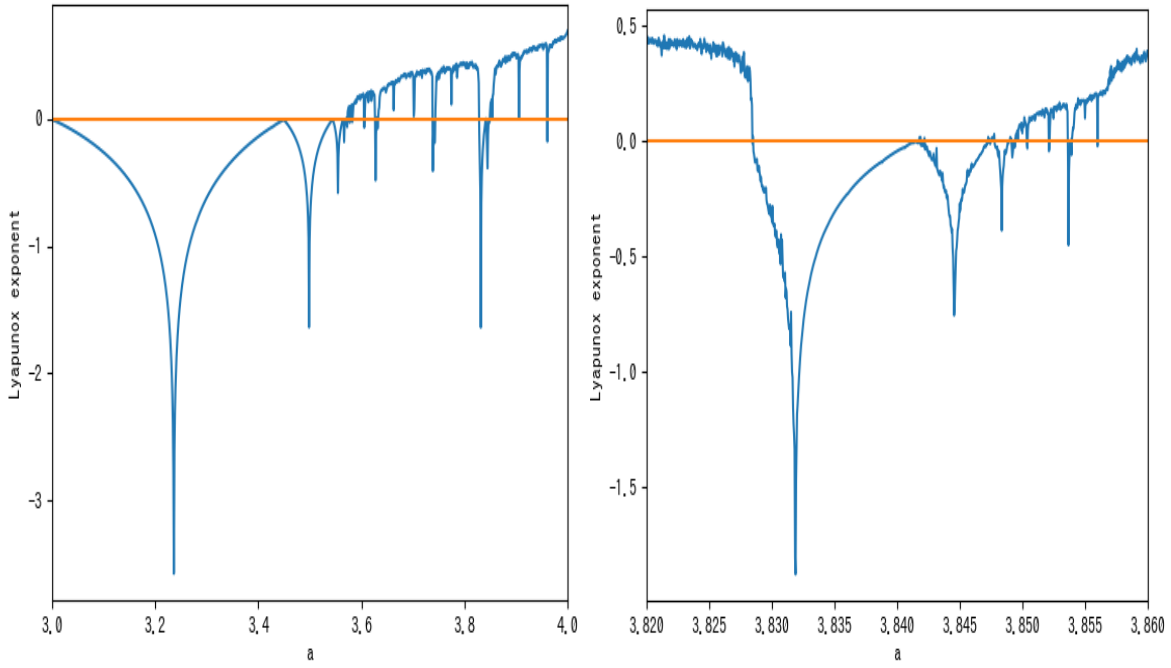
になります。この対数をとった平均値

$$L(f_a,x_0)=\lim_{n\to\infty}\frac{1}{n}\sum_{k=0}^{n}\log\left|\frac{df_a(x_k)}{dx}\right|,\qquad x_{k+1}=f_a(x_k)\tag{3.19}$$

を写像 f_a の点 x_0 に関する**リアプノフ指数** (Lyapunov exponent) といいます。

リアプノフ指数は x_0 を出発する近接軌道の振る舞いに関する情報を与えます。x_0 が吸引的な周期軌道であるとき、その近傍軌道は反覆のたびに点列が近接してリアプノフ指数は負値になます。軌道が初期値鋭敏性を持ち、近接した初期値から出発する軌道が最数的に遠く離れるようなときにはリアプノフ指数は正値となる。

図 3.14 は、プログラム 3.5-4 を使ってプロットしたロジスティック写像 (3.1) のリアプノフ指数の様子です。初期値 $x_0- = 04$ としてパラメータ区間を調べました。リアプノフ指数が正値をとるような反復軌道はランダム挙動を呈することに対応しています。分岐ダイヤグラム図 3.12 と同様に、パラメータに関する微細構造を有しています。



(a) パラメータ区間 $(3.0, 4.0)$　　(b) パラメータ区間 $(3.82, 3.86)$

図 3.14　ロジスティック写像 (3.1) のリアプノフ指数（初期値は $x_0 = 0.4$）。パラメータ区間を拡大しても微細構造が続く（分岐ダイヤグラム図 3.12 参照）。

コード 3.5-4　lyapunov_exp_losistic

```
import numpy as np
import matplotlib.pyplot as plt

def logistic(x: float, a: float) -> float:
    x1 = a * x * (1 - x)
    return(x1)

# derivative of logistic(x, a) w.r.t. x
def dlogistic(x: float, a: float) -> float:
    return(a *(1- 2 * x))

aStart = 3. #  from R,Shaw, Strange Attractors, Chaotic Behavior, and
    Information Flow(1981)
aEnd = 4.0 # [3.82, 3.86]
aNum = 1000 # Num of parameter a in [aStrat, aEmd]
aValues: np.ndarray = np.linspace(aStart, aEnd, aNum) # parameters
```

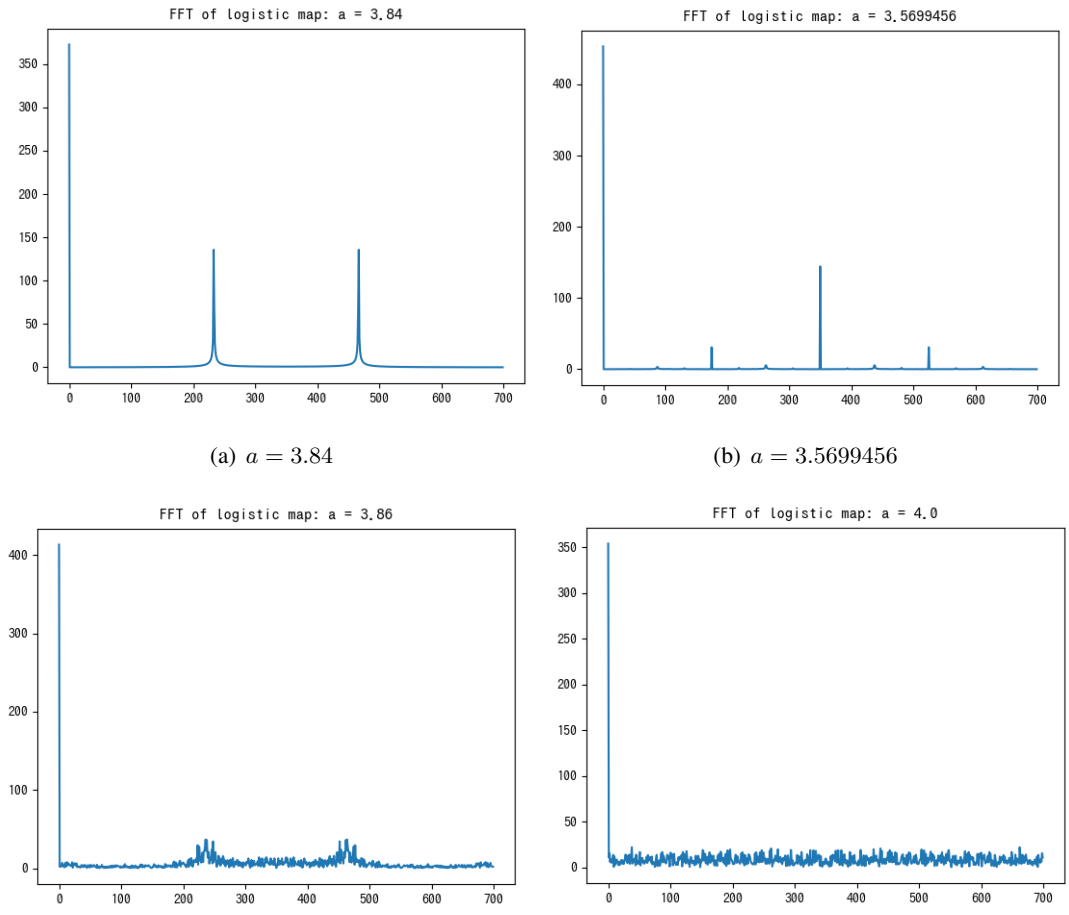
```
iteration =1000 # Num of iteration
LyapunovExp =[]
orbit: np.ndarray = np.empty((iteration,)) # array of logistic orbit
for a in aValues:
    x: np.float_ = 0.4 # initial point
    for i in range(iteration):
        orbit[i] = x
        x = logistic(x, a)
    LyapunovExp.append(np.mean(np.log(np.abs(dlogistic(orbit,a)))))

fig, ax = plt.subplots()
ax.plot(aValues, LyapunovExp, markersize=1)
ax.plot([aStart, aEnd],[0,0])
ax.set(xlabel='a', ylabel='Lyapunox exponent',
    xlim = (aStart, aEnd))
#fig.savefig('lyapunov_exp_losistic.png')
fig.show()
```

3.5.5 反復軌道の離散 Fourier 変換 `scipy.fftpack.fft`

区間 $[0,1]$ 上のロジスティック写像 (3.1) はパラメータ a に応じて初期値 x_0 からのロジスティック軌道 $\mathcal{O}_a(x_0)$ は大変複雑に挙動することがわかりました。節 3.5.2 ではパラメータ a と軌道の ω-極限集合との関係をダイヤグラムとして図 3.11 に、また節 3.5.4 ではリアプノフ指数として図 3.14 に表しました。

軌道の特徴づけには他にも様々な方法があります。ここではデータ解析にしばしば用いられる手段として Fourier 変換を取りげてみましょう。対象データが離散的であるとき、離散 Fourier 変換といい、デジタル信号などその適用が広範に及び重要度が高く高速アルゴリズムが研究・考案されています。今日で離散 Fourier 変換を**高速 Fourier 変換** (FFT) と呼んでいます。



(a) $a = 3.84$　(b) $a = 3.5699456$

(c) $a = 3.86$　(d) $a = 4$

図 3.15　ロジスチック写像 (3.1) のパラメータ a ごとに定まる軌道（初期点 $x_0 = 0.4$）の ω-極限集合的点列の FFT 結果。

Python では FFT 関連関数は SciPy モジュールの `scipy.fftpack` で提供されています。プログラム 3.5-5: `fft_logstic.py` は、関数 `get_orbit` からさだまる軌道点列において、初期点から `transit` 番目以前の点列を推移的軌道として取り去った点列 `orbit[transit:]` を `fftpack.fft()` を使って FFT を施してそのスペクトルを求めています。ここでは、`np.abs()` でその Fourier 係数の絶対値を計算して振幅スペクトルをプロットします（図 3.15 参照）。

コード 3.5-5　fft_logstic.py

```
import numpy as np
from scipy import fftpack
import matplotlib.pyplot as plt
```

```
def logistic(x: float, a: float) -> float:
    x1 = a * x * (1 - x)
    return(x1)

def get_orbit(x0: float, a: float, iteration: int) -> np.ndarray:
    xseq = []
    for i in range(iteration):
        xseq.append(x0)
        x0 = logistic(x0, a)
    return(np.array(xseq))

a = 3.5699456 #ファイゲンバウム#3.84# 吸引的3周期
x0 = 0.4# 初期値
iteration = 1000# 反復回数
transit = 300# 推移的反復回数
orbit = get_orbit(x0, a, iteration)
ft = fftpack.fft(orbit[transit:])

fig = plt.figure()
ax = plt.axes()
ax.plot(np.abs(ft))
ax.set(title = 'FFT of logistic map: a = '+str(a))
#fig.savefig('fft_logistic.png')
fig.show()
```

離散 Fourier 変換の結果は、離散データの周波数成分として複素 Fourier 係数のリストが得られます。その絶対値を**振幅スペクトル**と呼び、周波数成分の大きさを表します。図 3.15 が示すように、FFT の振幅スペクトルはグラフ中央を中心に線対称になります。

1000 回反復した軌道点列から始めの 300 点を推移的軌道として取り去った 700 点について FFT した Fourier 係数の絶対値をプロット。

ロジスティック写像の分岐ダイヤグラム（図 3.12）を仔細に検討すると、パラメータ値 $a = 3.84$ では多くの初期値からの軌道はある周期 3 の軌道に吸引され ω-極限集合は 3 点になります。パラメータ値 $a = 3.86$ では、ω-極限集合は区間 $[0, 1]$ 内の多くの領域を占めるようになりますが、訪問頻度は均等ではなく周期 3 の近傍近くの密度が高いことが観察できます。

こうした状況に対応して、FFT の振幅スペクトルはパラメータ値 $a = 3.82$ では図 3.15(a) のようにシャープな 2 つのピーク（線スペクトル）を持ち、$a = 3.86$ では図 3.15(b) のよう

にシャープさが損なわれ 2 つのピークの残滓が残った帯スペクトルになります。$a = 4$ では図 3.15(c) のようにスペクトルは一様になり、特別なピークを観察できなくなります。

第 4 章　pandas 入門

Pythion モジュール pandas は SQL を含むデータ処理や機械学習などのデータサイエンスのツールとして広く利用されています。科学技術計算においても、計算過程をファイルとして保存したり、巧みなデータ処理手法を援用することができます。

pandas で主に取り扱われるオブジェクトは *Series* と *DataFrame* で、NumPy 配列構造に準拠した操作が可能です。また、グラフ描画モジュール *Matplotlib* とも内的連携があり幅広い利用が可能です。

Series は 1 次元配列、*DataFrame* は一般化 2 次元配列（多次元配列として拡張可）です。pandas が取り扱う中心的なデータオブジェクトはデータフレームで直感的にはスプレッドシートにおける表データで、表計算で行えるデータ操作や計算はすべて pandas で行うことができます [**?**]。

ここでは、まず節 4.1 で与えられた表データを読み込んでデータフレームとしての取扱いを紹介します。その後、節 4.5 では *Series* と *DataFrame* オブジェクトのプログラム的生成を取り上げ、最後にデータ解析の簡単な事例として節 4.6 でテキストに出現する単語頻度に関する Zipf の法則の再発見を試みます。

4.1　表データをデータフレームとして読み込む

pandas のインポートは次のように `pd` を短縮名として記すのが慣例です。

```
import pandas as pd
```

ここではまず、既存の **CSV ファイル** (comma separated valued: 各行のフィールド（項目）カンマ「`,`」で区切られているテキストファイル）をデータフレームとして読み込んでみましょう。いま、次のような CSV ファイル `fruits.csv`、`number_unit.csv`、`districts.csv` がフォルダ `data` に用意されているとします。この例では、表データの各行のデータ並びの意味がわかるように CSV ファイルの先頭行に表データ各列の列名（データフレームの世界ではカラムと呼ぶことになります）を与えています。

```
$ cat data/fruits.csv
name,color,shape
apple,red,sphere
orange,orange,sphere
banana,yellow,finger
raspberry,red,cluster
```

```
grape,purple,cluster
lemon,yellow,spindle
pineapple,orange,spiny
```

```
$ cat data/districts.csv
name,district
apple,China
peach,Spain
cherry,China
pineapple,Costa Rica
banana,India
lemon,Mexco
papaya,India
fig,Turkey
```

```
$ cat data/number_unit.csv
number,unit
100,3
600,6
400,8
600,3
300,5
300,2
150,5
```

```
$ cat data/lights.csv
color,wave length(nm)
purple,380-450
blue,450-495
green,495-570
yellow,57-590
orange,590-620
red,620-750
```

演習 4.1CSV ファイル `fruits.csv`, `number_unit.csv`, `districts.csv` および `lights.csv` をテキストエディタで作成してみなさい。

では、pandas の関数 `read_csv()` を使ってこれらの CSV ファイルをつぎのようにデータフレームとして読み込みます。

```
>>> import pandas as pd
>>> fruits = pd.read_csv('data/fruits.csv')
>>> fruits
        name   color    shape
0      apple     red   sphere
1     orange  orange   sphere
2     banana  yellow   finger
3  raspberry     red  cluster
4      grape  purple  cluster
5      lemon  yellow  spindle
6  pineapple  orange    spiny
```

`fruits` を表示させてわかるように、各行データの左端には（この場合のように指定がなければ自動的に）0 から始まる**インデックス**と、（今の場合は明示的に）各列データには列ラベル名として**カラム**が割り当てられています。ただし、CSV ファイルの先頭行をカラムとしているだけで、列ラベル名を判別しているわけではありません。明示的にカラムを指定する方法は、節 4.3 で取り上げます。

データフレームに割当てられているインデックス属性およびカラム属性は次のようにその属性値 `.index` および `.columns` から確かめることができます。

```
>>> fruits.index
RangeIndex(start=0, stop=7, step=1)
>>> fruits.columns
Index(['name', 'color', 'shape'], dtype='object')
```

今の例では、データを区別する行ラベルとして整数値（0 から 6）が、列ラベルとして文字列（'name'、'color','shape'）が割り当てられています。

このインデックス属性値 .index およびカラム属性値.columns の結果を Python のリスト関数 list() に渡すと、リストにすることができます。pandas オブジェクトでは、節 4.2 で紹介するように、これらの属性を使ったデータ選択（スライス）が可能になりますが、これらの属性値のリスト化は節 4.5.1 で紹介するような使い方を可能に増す。

```
list(fruits.index)
Out[52]: [0, 1, 2, 3, 4, 5, 6]

In [53]: list(fruits.columns)
Out[53]: ['name', 'color', 'shape']
```

演習 4.2関数 pd.read_csv()を使って、CSV ファイル fruits.csv, number_unit.csv および districts.csv をそれぞれデータフレーム fruits, numbers および districts として読み込んでみなさい。

演習 4.3pandas には **Excel ファイル**（拡張子:.xlsx, .xls）を読み込む関数 pd.read_excel() が用意されています（ただし、パッケージ xlrd がインストールされている必要があります)。オプション sheet_name で読み込むシートを番号・シート名で指定したり、header や index_col でヘッダー、インデックスを指定したり、usecols, skiprows で読み込む列、読み込まない行を指定することができます。でみなさい。関数 pd.read_excel() の使い方を調べてみなさい。

4.1.1 データフレームの書き出し

プログラム内で得られたデータフレーム（や *Series*）df を CSV ファイル hoge.csv として書き出すには関数 to_csv() を使います。

```
df.to_csv('hoge.csv')
```

オプションとして、index で行名を出力するかどうか（デフォルト True)、header でカラムがあるときヘッダを付けるか（デフォルト True）や columns で特定の列のみ書き出したりできます。また、sep で区切り文字を指定することができます (デフォルトはカンマ','）。

次はデータフレーム fruits を、行番号をつけてそのカラム（列名）'color','shape' の列だけをヘッダとして CSV ファイル data/pfruits.csv として書き出した例です。

```
>>> fruits.to_csv('data/pfruits.csv', columns=['color', 'shape'])
```

演習 4.4pandas には **Excel ファイル**（拡張子:.xlsx, .xls）を書き出すインスタンスメソッド.to_excel() が用意されています（ただし、パッケージ xlwt, openpyxl がインストールされている必要があります)。オプション sheet_name で書き出すシート名を指定したり、index の有無や特定カラムを columns で指定できます。

また複数のデータフレームを Python の with ブロック文を使って 1 つの Excel ファイルに書き出す pandas 関数 pd.ExcelWriter() が用意されています。

メソッド pd.read_excel() や関数 pd.ExcelWriter() の使い方を調べてみなさい。

4.2 データフレームからのデータ選択

データフレームはその values 属性値を見ると NumPy 配列の構造を持っていることがわかります。

```
>>> fruits.values
array([['apple', 'red', 'sphere'],
       ['orange', 'orange', 'sphere'],
       ['banana', 'yellow', 'finger'],
       ['raspberry', 'red', 'cluster'],
       ['grape', 'purple', 'cluster'],
       ['lemon', 'yellow', 'spindle'],
       ['pineapple', 'orange', 'spiny']], dtype=object)
>>> fruits.values[1:4]
array([['orange', 'orange', 'sphere'],
       ['banana', 'yellow', 'finger'],
       ['raspberry', 'red', 'cluster']], dtype=object)
>>> fruits.values[1:4,1:3]
array([['orange', 'sphere'],
       ['yellow', 'finger'],
       ['red', 'cluster']], dtype=object)
```

データフレームには、配列形式でデータを取り出すために、loc、iloc および ix という 3 つのインデックス属性を使うことができ、NumPy スタイルで [行, 列] を指定してその要素を取り出すことができます。

行インデックス、列カラムとして、loc はデータフレームのラベルで、iloc はデータフレームの番号（先頭は 0）で、そして ix はラベルまたは番号（先頭が 0）のいずれかで

指定します。ただし、その結果にはデータフレームのインデックスとカラムが保持されていることに注意します。

```
>>> fruits.loc[1:4]
        name   color    shape
1     orange  orange   sphere
2     banana  yellow   finger
3  raspberry     red  cluster
4      grape  purple  cluster
>>> fruits.loc[1:4,'color':'shape']
    color    shape
1  orange   sphere
2  yellow   finger
3     red  cluster
4  purple  cluster
>>> fruits.iloc[1:4,2]
1     sphere
2     finger
3    cluster
Name: shape, dtyp
>>> fruits.loc[:,'color']
0       red
1    orange
2    yellow
3       red
4    purple
5    yellow
6    orange
Name: color, dtype: object
```

演習 4.5データフレーム `fruits` から、指定した範囲にあるデータを取り出して表示しなさい。

4.3 インデックス、カラムの変更

いままで、インデックスは 0 から始まる整数値に設定されていました。データフレームのあるカラム、たとえば'name' を、次のように.set_index() を使ってインデックスを付け替えることができます。

```
>>> nfruits = fruits.set_index('name')
>>> nfruits
            color    shape
name
```

```
apple          red   sphere
orange     orange   sphere
banana     yellow   finger
raspberry     red  cluster
grape      purple  cluster
lemon      yellow  spindle
pineapple  orange    spiny
>>> nfruits.index
Index(['apple', 'orange', 'banana', 'raspberry', 'grape', 'lemon',
       'pineapple'],
      dtype='object', name='name')
```

あるいは、CSV ファイルをデータフレームとして読み込む際、インデックスを指定するオプション `index_col` を使って、（先頭行を）列ラベルとして持つ `name` をデータフレームのインデックスに指定することもできます。

```
>>> pd.read_csv('data/fruits.csv', index_col='name')
            color    shape
name
apple          red   sphere
orange     orange   sphere
banana     yellow   finger
raspberry     red  cluster
grape      purple  cluster
lemon      yellow  spindle
pineapple  orange    spiny
```

演習 4.6データフレーム `fruits` から、指定した列ラベルをインデックスとするデータフレームを生成しなさい。

列番号によるデータ操作はプログラムの可読性を極端に低下させます。データフレームの各列にはふさわしい名称のカラムを付けるように心がけると良いでしょう。データフレームでは、カラムを次のようにラベル名を指定するだけで 1 列すべてを取り出すことができます。

```
>>> fruits['color']
0       red
1    orange
2    yellow
3       red
4    purple
5    yellow
6    orange
Name: color, dtype: object
```

```
>>> fruits['shape']
0     sphere
1     sphere
2     finger
3    cluster
4    cluster
5    spindle
6      spiny
```

いま、次のように CSV ファイルを読み込んだデータフレームを見てみましょう。CSV ファイルの 1 行目をカラム要素'meron' と'fruit' とするデータフレームになっていることが確認できます。

```
>>> food = pd.read_csv('data/food.csv')
>>> food
     meron      fruit
0   carrot  vegetable
1    peach      fruit
2     clam      shell
3   cherry      fruit
4     pork       meet
5     beef       meet
6   pepper  vegetable
7  sardine       fish
8   poteto  vegetable
>>> food.columns
Index(['meron', 'fruit'], dtype='object')
```

読み込む CSV ファイルに対し、データフレームのカラムを名称の並びリストとして指定することができます。

```
>>> cfood = pd.read_csv('data/food.csv', names=['name', 'class'])
>>> cfood
      name      class
0    meron      fruit
1   carrot  vegetable
2    peach      fruit
3     clam      shell
4   cherry      fruit
5     pork       meet
6     beef       meet
7   pepper  vegetable
8  sardine       fish
9   poteto  vegetable
```

```
>>> cfood.columns
Index(['name', 'class'], dtype='object')
```

pandas 関数 pd.rename() を使って現在のデータフレームのカラム名称を、Python ディクショナリ型のようにコロン (:) で「旧ラベル : 新ラベル」と挟んだ組の集合で再指定して新たなデータフレームを生成することも可能です。

```
>>> food.rename(columns = {'meron' : 'name', 'fruit' : 'class'})
      name      class
0   carrot  vegetable
1    peach      fruit
2     clam      shell
3   cherry      fruit
4     pork       meet
5     beef       meet
6   pepper  vegetable
7  sardine       fish
8   poteto  vegetable
```

ただし、現在のデータフレーム fruits では'meron' と'fruit' が既にカラム名称になっているため、それらを変更したとしても CSV ファイルの読み込み時のカラム指定の結果とは一致しないことに注意してください（'melon' を含む行が消えています)。

演習 4.7CSV ファイル food.csv を作成し、適切なカラムを指定してデータフレームとして読み込みなさい。

4.4 データフレームの結合とマージ

2 つ以上のデータフレームを結合したり、合併して新たなデータフレームを生成することを考えましょう。pandas を利用したデータ解析の活用に関する中心的話題ですが、ここでは簡単な例を取り上げるだけにとどめます。データフレームの結合や合併においては、生成されたデータフレームのカラム（列ラベル）や行数に注目してください。

節 4.1 の最初に戻って、別の CSV ファイル number_unit.csv および districts.csv を次のようにしてデータフレームとして読み込みます。それぞれカラム ['number', 'unit'] および ['name', 'district'] を持つことに注意してください。

```
>>> numbers = pd.read_csv('data/number_unit.csv')
>>> numbers
   number  unit
0     100     3
1     600     6
```

```
2     400     8
3     600     3
4     300     5
5     300     2
6     150     5
>>> districts = pd.read_csv('data/districts.csv')
>>> district
        name    district
0      apple       China
1      peach       Spain
2     cherry       China
3  pineapple  Costa Rica
4     banana       India
5      lemon       Mexco
6     papaya       India
```

4.4.1 データフレームの結合

pandas 関数 `pd.concat()` を使うと複数のデータフレームが結合されます。ここで注意すべきは結合の方向で、デフォルト（軸指定 `axis=0` は省略できる）は行方向に積み重ねる結合ですが、軸指定を `axis=1`（列）とし列方向に並べて結合することもできます。

デフォルトの行方向の結合は次のようになります。

```
>>> fn_concat = pd.concat([fruits, numbers])
    color       name  number    shape  unit
0     red      apple     NaN   sphere   NaN
1  orange     orange     NaN   sphere   NaN
2  yellow     banana     NaN   finger   NaN
3     red  raspberry     NaN  cluster   NaN
4  purple      grape     NaN  cluster   NaN
5  yellow      lemon     NaN  spindle   NaN
6  orange  pineapple     NaN    spiny   NaN
0     NaN        NaN   100.0      NaN   3.0
1     NaN        NaN   600.0      NaN   6.0
2     NaN        NaN   400.0      NaN   8.0
3     NaN        NaN   600.0      NaN   3.0
4     NaN        NaN   300.0      NaN   5.0
5     NaN        NaN   300.0      NaN   2.0
6     NaN        NaN   150.0      NaN   5.0
>>> pd.concat([fruits, numbers], axis=1)
        name   color    shape  number  unit
0      apple     red   sphere     100     3
1     orange  orange   sphere     600     6
```

```
2     banana  yellow   finger     400     8
3  raspberry     red  cluster     600     3
4      grape  purple  cluster     300     5
5      lemon  yellow  spindle     300     2
6  pineapple  orange    spiny     150     5
```

ここで NaN(Not a Number) は pandas が表す**欠損値**の一種で、結合に際して該当するデータが存在しない（null）場合に pandas がそのエントリに挿入する特殊値です。pandas では他にデータ欠損を示す標準の Python オブジェクト None も使われます。欠損値を一般に NA(Not Available) と呼んでいます。

データフレームに欠損値があるかどうかの検出はインスタンスメソッド isnull() を使って行えます。

```
>>> fnfn_concat.loc[0]
  color   name  number   shape  unit
0   red  apple     NaN  sphere   NaN
0   NaN    NaN   100.0     NaN   3.0
>>> fnfn_concat.loc[0].isnull()
   color   name  number  shape   unit
0  False  False    True  False   True
0   True   True   False   True  False
```

実世界から得られるデータには入力漏れを含む様々なデータ欠損が生じます。データ分析においては欠損データの取扱いをどうするかは大きな問題になります（求めるべき平均値に必要なデータが無いなど計算式で使うべき値が欠損している場合に式の評価結果の取扱はどうすべきでしょう）。pandas では、欠損値が存在するデータについて概ね満足できる処理を行う事ができます。

演習 4.8データフレームの行方向と列方向への結合の差異は一般にどうなるのかを説明してみなさい。

演習 4.9データフレーム fruits と districts を行方向および列方向に結合してみなさい。

pandas 関数 pd.concat() はみてきたように、連結方向のラベル (行連結の場合は index、列連結の場合は columns) に対して保持されるため、演習 4.9 のデータフレーム fruits と districts を列方向への結合のように 連結結果に同じラベル名が重複してしまうことがあり得ます。

連結結果にラベル名が重複する場合、そのラベル名を区別して管理することができます。連結の際、次のように連結元のデータフレームごとにオプション keys にラベル名を付与すると、連結されたデータフレームは多重インデックス構造 **MultiIndex** を待ちます。

```
>>> multiDF = pd.concat([fruits, districts], axis = 1, keys=['fruits', 'product
>>> multiDF
      fruits                    product
        name   color    shape       name    district
0      apple     red   sphere      apple       China
1     orange  orange   sphere      peach       Spain
2     banana  yellow   finger     cherry       China
3  raspberry     red  cluster  pineapple  Costa Rica
4      grape  purple  cluster     banana       India
5      lemon  yellow  spindle      lemon       Mexco
6  pineapple  orange    spiny     papaya       India
7        NaN     NaN      NaN        fig      Turkey
>>> multiDF['fruits', 'color']
0       red
1    orange
2    yellow
3       red
4    purple
5    yellow
6    orange
7       NaN
Name: (fruits, color), dtype: object
```

この例のように、データフレームの結合では対応する行データエントリをマッチングさせるわけではないことに注意してください。たまたま'apple'と'lemon'を含む行だけ（0行と5行）がマッチしているに過ぎません。

```
>>> multiDF['fruits', 'name'] == multiDF['product', 'name']
0     True
1    False
2    False
3    False
4    False
5     True
6    False
7    False
dtype: bool
```

4.4.2 データフレームの合併

pandasは**関係データベース** (RDB: Relational Data Base) のSQL演算機能を提供します[5]。ここで、関係演算とは、ある属性を含むデータセットAと同じ属性を含むことなるデータセットBがあるとき、その属性をキーとして関連し合う2つのデータセットを合併

して新たなデータセットを生成したり、異なるデータセットから共通する属性を持つデータセットを生成するような操作です。

データセット集合に対する関係演算を組み合わせて、複雑なデータ操作を実現することができるようになリます。こうしたデータ操作についてはたくさんの話題がありますが、ここでは pandas の合併メソッド `pd.merge()` のごく基本だけを紹介します。

合併 (merge) とは単純にデータセットを（行または列方向に）並べるのではなく、異なるデータセットをつなげる**キー属性**となるデータ値を探しながら新しいデータセットとして結合する操作です。`pd.merge()` によるデータフレーム同士の合併を具体的に見てみましょう。

データフレーム `fruits` のカラム `['name', 'color', 'shape']` と `districts` のカラム `(['name', 'district']` は

$$\{\text{name}, \text{color}, \text{shape}\}_{\text{fruits}} \cap \{\text{name}, \text{district}\}_{\text{district}} = \text{name}$$

と共通要素`'name'` を持ち、これが 2 つのデータフレームのキー属性となって次の合併操作が成立します。

```
>>> pd.merge(fruits, districts)
        name   color    shape    district
0      apple     red   sphere       China
1     banana  yellow   finger       India
2      lemon  yellow  spindle       Mexco
3  pineapple  orange    spiny  Costa Rica
```

今の場合、キー属性`'name'` におけるデータ値の並びはそれぞれのデータフレームでは 1 回しか登場しておらず、キー属性`'name'` による合併は 1 対 1 です。

合併のためのキー属性（列名）の解決を `pd.merge()` に走査させるのではなく、意図を明確にしプログラムの可読性を向上させるために、次のようにオプション `on` を使って明示的に列名を指定するとよいでしょう。

```
>>> pd.merge(fruits, districts, on = 'name')
```

次にデータフレーム `lights` を用意します。

```
>>> lights = pd.read_csv('data/lights.csv')
>>> lights
    color wavelength(nm)
0  purple        380-450
1    blue        450-495
2   green        495-570
3  yellow        570-590
```

```
4  orange         590-620
5     red         620-750
>>> lights.columns
Index(['color', 'wavelength(nm)'], dtype='object')
```

データフレーム fruits と lights とのカラムの共通ラベルが'color' であること

$$\{\text{name}, \text{color}, \text{shape}\}_{\text{fruits}} \cap \{\text{color}, \text{wavelength(nm)}\}_{\text{lights}} = \text{color}$$

であることを確かめた上で、fruits と lights とを合併すると次のようになります。

```
>>> pd.merge(fruits, lights, on = 'name')
        name   color    shape wavelength(nm)
0      apple     red   sphere        620-750
1  raspberry     red  cluster        620-750
2     orange  orange   sphere        590-620
3  pineapple  orange    spiny        590-620
4     banana  yellow   finger        570-590
5      lemon  yellow  spindle        570-590
6      grape  purple  cluster        380-450
```

この場合、キー属性'color' におけるデータ値の並びはそれぞれのデータフレームにおいても複数回登場しており、キー属性'color' による合併は多対多担っています。

データフレーム同士の合併 pd.merge() では、以上のような 1 対 1、多対多だけでなく、多対 1 による合併があります。

演習 4.10 2 つのデータフレームを合併して、上述のことを確認してみなさい。

4.4.3 データフレームの合併時の問題

データフレームの合併に際して pd.merge() は 2 つの入力データフレーム間でカラムを走査してキー属性を見つけようとします。目的とするキー属性（列名）が一致しない場合もありえます。たとえば、合併対象となる一方のデータフレーム sdistricts のキー属性（列名）が'name' ではく、'species' となっている場合を考えてみましょう。

```
>>> sdistricts = districts.rename(columns={'name': 'species'})
>>> sdistricts
     species    district
0      apple       China
1      peach       Spain
2     cherry       China
3  pineapple  Costa Rica
4     banana       India
5      lemon       Mexco
```

```
6     papaya        India
7        fig       Turkey
```

このとき、データフレーム fruits と sdistricts の合併を試みると、pandas は次のようなエラーメッセージを表示します。

```
>>> pd.merge(fruits, sdistricts)
...
pandas.errors.MergeError: No common columns to perform merge on. Merge
options: left_on=None, right_on=None, left_index=False, right_index=False
```

fruits と sdistricts とでキー属性（列名）が異なっていたとしても、上のエラーメッセージあるように次のようにオプション left_on および right_on によってそれぞれのデータフレームのキーを指定することで通常の合併が可能です。

```
>>> pd.merge(fruits, sdistricts, left_on = 'name', right_on = 'species')
        name   color    shape    species    district
0      apple     red   sphere      apple       China
1     banana  yellow   finger     banana       India
2      lemon  yellow  spindle      lemon       Mexco
3  pineapple  orange    spiny  pineapple  Costa Rica
```

演習 4.11データフレーム fruits と districts の合併に際して、オプション指定 left_on および right_on をすると次のようになる。

```
>>> pd.merge(fruits, district, left_on = 'name', right_on = 'district')
Empty DataFrame
Columns: [name_x, color, shape, name_y, district]
Index: []
```

この結果を説明しなさい。

合併演算において、一般的には一方のデータフレームのキー属性（列名）にデータ値があり、他方のデータフレームの対応するキー属性（列名）には相当するデータ値がないという状況が生じます。

合併に際して合併演算をどのように行うか (how) について、pd.merge() のデフォルト動作は**内部連結**として、つまり 2 つのデータフレームのキー属性（列名）に関する入力データセットの共通集合、2 つのデータフレームのキー属性（列名）に両方共データ値があるように合併したデータフレームが返るように動作します。

実際、pd.merge() は合併演算の仕方をオプション how によって明示的に指定でき、次のようにデフォルト値は how='inner' となっています。

```
>>> pd.merge(fruits, districts, on = 'name', how = 'inner')
```

```
        name   color    shape    district
0      apple     red   sphere       China
1     banana  yellow   finger       India
2      lemon  yellow  spindle       Mexco
3  pineapple  orange    spiny  Costa Rica
```

how キーワードには'inner' 以外にも、他に'outer'（**外部結合**）、'left'（**左結合**）、'right'（**右結合**）を持つことができます。

how = 'outer' により外部結合を指定すると、合併に際してデータフレームのキー属性（列名）の入力列の和集合を返し、欠損値を NA で埋めます。

```
>>> ofruits = pd.merge(fruits, districts, on = 'name', how = 'outer')
         name   color    shape    district
0       apple     red   sphere       China
1      orange  orange   sphere         NaN
2      banana  yellow   finger       India
3   raspberry     red  cluster         NaN
4       grape  purple  cluster         NaN
5       lemon  yellow  spindle       Mexco
6   pineapple  orange    spiny  Costa Rica
7       peach     NaN      NaN       Spain
8      cherry     NaN      NaN       China
9      papaya     NaN      NaN       India
10        fig     NaN      NaN      Turkey
```

how = 'left' により左結合を指定すると、合併に際して左データフレームにあるキー属性（列名）のデータ列を使い、欠損値を NA で埋めます。

```
>>> pd.merge(fruits, districts, on = 'name', how = 'left')
        name   color    shape    district
0      apple     red   sphere       China
1     orange  orange   sphere         NaN
2     banana  yellow   finger       India
3  raspberry     red  cluster         NaN
4      grape  purple  cluster         NaN
5      lemon  yellow  spindle       Mexco
6  pineapple  orange    spiny  Costa Rica
```

how = 'right' により左結合を指定すると、合併に際して右データフレームにあるキー属性（列名）のデータ列を使い、欠損値を NA で埋めます。

```
>>> pd.merge(fruits, districts, on = 'name', how = 'right')
        name   color    shape    district
0      apple     red   sphere       China
```

```
1     banana  yellow   finger       India
2      lemon  yellow  spindle       Mexco
3  pineapple  orange    spiny  Costa Rica
4      peach     NaN      NaN       Spain
5     cherry     NaN      NaN       China
6     papaya     NaN      NaN       India
7        fig     NaN      NaN      Turkey
```

データフレームの合併の仕方 `how` の指定をどうするかは、合併後のデータ操作など、高価い立場からの判断に基づいて決定される必要があります。

4.4.4 欠損値の除外と値設定

欠損値[1)] を除外するメソッド `dropna()` があります。

欠損値を含む次の *Series* インスタンス `ser` を見ましょう。

```
>>> ser = pd.Series([np.nan, 1, np.nan, 'apple', None, np.nan, 3.14, None])
>>> ser
0      NaN
1        1
2      NaN
3    apple
4     None
5      NaN
6     3.14
7     None
dtype: object
>>> ser.isnull()
0     True
1    False
2     True
3    False
4     True
5     True
6    False
7     True
dtype: bool
```

Series インスタンスに `dropna()` を適用すると、NA 値要素が除外されます。

```
>>> ser.dropna()
```

1） NumPy や pandas で無限大を意味する値 `inf` は欠損値ではありません。NumPy 配列を 0 除算すると実行時警告を出しながらを結果を返しますが、pandas の Series 配列を 0 除算すると `float64` 型の浮動小数点として扱われます。

```
1        1
3    apple
6     3.14
dtype: object
```

ただし、データフレームに欠損値除外メソッド dropna() を適用すると、デフォルトでは 1 つでも NA 値を含む行全体を除外します。たとえば、データフレーム ofruits に対して実行すると次のようになります。

```
>>> ofruits.dropna()
        name   color    shape    district
0      apple     red   sphere       China
2     banana  yellow   finger       India
5      lemon  yellow  spindle       Mexco
6  pineapple  orange    spiny  Costa Rica
```

データフレームに適用する dropna() にオプション axis='columns' を設定すると、NA 値を含む全ての列全体を除外します。

```
>>> ofruits.dropna(axis='columns')
         name
0       apple
1      orange
2      banana
3   raspberry
4       grape
5       lemon
6   pineapple
7       peach
8      cherry
9      papaya
10        fig
```

欠損値 NA を持つデータフレームの処理において、NA 値を除外するのでなく、なにか適切な値で置き換えることが望ましい場合もあり得ます。pandas の NA 値を一斉に指定した値に置き換えるメソッド fillna() があります。次は NA 値をすべて 0 に置き換えています。

```
>>> ser.fillna(0)
0        0
1        1
2        0
3    apple
4        0
```

```
5         0
6      3.14
7         0
dtype: object
```

メソッド `fillna()` のオプションに NA 要素の 1 つ前の値で埋める `method='ffill'` (forward fill) と NA 要素の 1 つ後ろの値で埋める `method='bfill'` (back fill) 次の結果を返します。

```
>>> ser.fillna(method='ffill')
0       NaN
1         1
2         1
3     apple
4     apple
5     apple
6      3.14
7      3.14
dtype: object
>>> ser.fillna(method='bfill')
0         1
1         1
2     apple
3     apple
4      3.14
5      3.14
6      3.14
7      None
dtype: object
```

この例のように、forward fill 時に前の値が利用できないときや back fill 時に後ろの値が使えないときには NA 値が残ることに注意してください。

4.5 *Series* と *DataFrame* データの生成

これまで pandas の特にデータフレームの取扱を紹介してきました。ここでは最初に戻って、pandas が提供するオブジェクトを作成することを紹介します。

pandas には主なオブジェクトとして、インデックス付きの 1 次元的データである *Series*、インデックス付きの多次元的（一般化 2 次元的）データである *DataFrame*、そして *Series* や *DataFrame* が保持するデータを参照できる仕組みを提供する *index* オブジェクトがあります。

プログラム内で出力される諸データを *Series* や *DataFrame* データに変換して CVS（あるいは Excel 形式）で書き出して、Python 以外のソフトウエアで処理するという連携が考えられます[2)]。

4.5.1 *Series* の作成

4.5.2 *DataFrame* の作成

4.5.3 *Series* データの更新

pandas データからインデックス情報をうまく利用すると、わかり易く効率的なデータの取り出しが可能になります。しかしながら、pandas の *Series* やデータフレームの生成にはインデックス情報の構造体管理などに相応の計算資源が必要なため、for 文などで生成や追加操作を多数回繰り返すような処理には適していません。pandas データの生成には、必要とするセータセットを用意しておき、適切な段階で pandas データへと変換するようにすると高速で便利な処理を達成することができることを紹介してみます。

プログラム 4.5-1(`appending_series.py`) は、引数に NumPy リストを与えて、そのリストデータを有する *Series* オブジェクトを 2 つの方法で作成するプログラムです。

関数 `list2series_appending(list)` は空の *Series* オブジェクト `ser` を用意しておき、`list` の先頭から要素 `data` を取り出してその 1 つだけのデータからなる *Series* オブジェクト `s` をメソッド`.append()` を使って次々に追加して `ser` を更新しながら求める *Series* オブジェクトを返します。

関数 `list2series(list)` は空の *NumPy* のリスト `lst` をを用意しておき、`list` の先頭から要素 `data` を取り出して Python リストのメソッド`.append()` を使ってリスト `lst` を更新し、最後に pandas 関数 `pd.Series()` を使って *Series* オブジェクトにして、その結果を返します。

コード 4.5-1　appending_series.py

```
import numpy as np
import pandas as pd

def list2series_appending(list):
    ser = pd.Series([])
    for data in list:
```

2)　Python プログラムで得られた諸データを逐次 CVS 形式を含むテキストファイルに書き出すことはもちろん可能です。たとえば CSV 形式に書き出す場合には、データ以外に区切り文字（通常はカンマ,）も併せて書き出し書式を整える必要があります。

```
        s = pd.Series([data])
        ser = ser.append(s, ignore_index = True)
    return(ser)

def list2series(list):
    lst = []
    for data in list:
        lst.append(data)
    ser = pd.Series(lst)
    return(ser)

length = 100
lst = np.random.randint(0, 100, size = length)
lser = list2series_appending(lst)
```

次はIPythonでプログラムappending_series.pyを実行してnp.random.randint()で長さ100の（0から100未満の）整数乱数リストlstを生成した上で、そのデータからなる*Series*の計算時間を%timeitを使って（節1.2参照）計測した結果です。

1次元リストから一回で*Series*オブジェクトに変更する関数list2series()の実行時間は、毎回*Series*を生成して、*Series*追加する関数関数list2series_appending()に比べて100倍以上早いことがわかります。

```
In [1]: %run adding_series.py
....

In [2]: %timeit list2series_appending(lst)
32.3 ms ś 692 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)

In [3]: %timeit list2series(lst)
281 ţs ś 1.06 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

インデックス情報（この例では、行数番号としています）を保持する*Series*オブジェクトの生成や追加操作の実行はデータ量にかかわらず計算時間を消費することがわかりました。次に示すように、DataFrameにおいても事情は同様です。プログラミングにおいては、プログラム可読性とのバランスを取りながら重い処理を要する箇所（回数）を減らす工夫をすると良いでしょう。

演習 4.12プログラムappending_series.pyを実行して、リストサイズを変えて、2つの関数list2series()とlist2series_appending()の実行時間を比較してみなさい。

演習 4.13プログラム `appending_series.py` の 8 行目にあるメソッド `append()` のオプション `ignore_index = True` を削除した場合、関数 `list2series_appending()` はどのようにな *Series* オブジェクトを返すかを確かめてみなさい。

節 3.5 で取り上げた 1 次元区間 $[0,1]$ 上のロジスチック写像 $f_a(x) = ax(1-x), 0 < a \leqq 4$ の反復によって得られる軌道をもう一度考えます。初期点 x_0 からの軌道 $\mathcal{O}_a(x_0)$

$$\mathcal{O}_a(x_0) = \{x_0, x_1, x_2, \ldots, \}, \qquad x_{n+1} = f_a(x_n), \quad n \geqq 0$$

を求め、反復回数を横軸に区間 $[0,1]$ 上にある点列をプロットしてみましょう。

プログラム 4.1（`pandas_logistic.py`）はロジスチック関数の反復回数とその軌道点列をプロットします。5 行目でロジスチック関数 `logistic(x, a)` を定義し、12 行で空リスト `orbit` を用意しておき、15 行目までで写像反復で得られる点列をリストメソッド `append()` でリストに追加します。そうして得られた軌道点列リスト `orbit` を 16 行目で `Series` オブジェクト `oseries` に一括変換します。同じ結果は、12 行目から 16 行目までを次で置き換えた

```
oseries = pd.Series([])
for k in range(iteration):
    s = pd.Series([x0])
    oseries = oseries.append(s, ignore_index = True)
    x0  = logistic(x0, a)
```

によっても計算できますが、先に見たように毎回 *Series* オブジェクトを生成して更新する処理方法は実行時間上のボトルネックとなるため、プログラム 4.1 のように求めるリストデータを生成してから一度で `Series` オブジェクトに変換します。

反復回数と区間 $[0,1]$ 内の点位置とをプロットするために、21 行目で `oseries` のインデックス属性`.index` からデフォルトの整数値インデックス $0, 1, \ldots, 199$ からリスト `[0,1,...,199]` が得るために Python のリスト化関数 `list()` を `list(oseries.index)` のように使っています（節 4.1 参照）。ここでは、初期点からの反復回数 50 を `transit` とし、`[transit:]` とスライスすることによって 50 回以降の点をプロットしています。

コード 4.5-2　pandas_logistic.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


def logistic(x, a):
```

```
    x1 = a * x * (1 - x)
    return(x1)

a = 3.5
x0 = 0.4
iteration =  200
orbit = []
for k in range(iteration):
    orbit.append(x0)
    x0 = logistic(x0, a)
oseries = pd.Series(orbit)

transit = 150
fig = plt.figure()
ax = plt.axes()
ax.plot(list(oseries.index)[transit:], oseries[transit:])
fig.savefig('logistic_a04.png')
fig.show()
```

図 4.1 はプログラム 4.1 で得られる反復回数と軌道移転列をプロットした結果です。パラメータを $a = 0.4$ を持つロジスチック関数は $p_1 = f_a(p_1), p_1 = f_a(p_2)$ であるような周期 2 のアトラクタを持ち、区間 $(0, 1)$ から出発するすべての軌道は $\{p_1, p_2\}$ に漸近することがわかっています。実際、反復回数 150 から 199 までの点列軌道をプロットした図 4.1 は反復のたびに周期点 $\{p_1, p_2\}$ に交互に写像されている様子を示しています。

演習 4.14プログラム 4.1pandas_logistic.py を実行し、インデックス属性値 `orbit2series.index` とそのリスト化結果 `list(orbit2series.index)` を確かめてみなさい。

4.5.4 *DataFrame* データの更新

4.6 Zipf の法則の再発見

pandas を使って読み取ったデータを表形式にまとめながら可視化するプログラム例として、Zipf の法則を再発見することを考えてみます。言語学者 G.K. Zipf が言語データ（コーパス）における単語の出現順位 k とその出現頻度 $p(k)$ との間に**べき法則**、ある正定

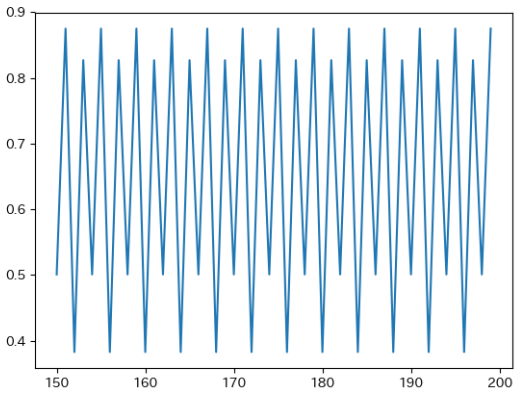


図 4.1　区間 $[0,1]$ 上のロジスチック写像 $f_a(x) = ax(1-x)$ の軌道。パラメータ $a = 0.4$ で初期点 $x_0 = 0.4$ から写像を反復して得られる軌道列を計算し、最初の 150 回を省いた得られる点列 $\{f_a^{150}(x_0), \ldots, f_a^{199}(x_0)\}$ をプロット。周期 2 の軌道に漸近している様子が観測できる。

数 $s >$ があって

$$p(k) \sim k^{-s}$$

が観察できるという指摘に基づいた経験法則です。Zipf の法則は言語減少だけでなく、順位と規模に関する自然現象や社会現象でもしばしば観察される現象として知られています。

ここでは、コマンドラインで指定したローマ字テキスト（英語やラテン語の文章）を対象に Zipf の法則の再発見を試みます。テキストに登場する単語を切り出し、テキストに登場する全ての単語を重複のある単語リストとして取り出してから、単語をキー、その登場回数を値とするディクショナリを作成し、登場回数についてディクショナリ要素を降順に並べ替えて、順位（最頻出単語が 1 位となります）とその出現頻度（出現回数を総単語数で割った値）の関係を調べます。

プログラム 4.6-1 の関数 `modify_line(string)` は、テキストから 1 行ずつ読み込んだ文字列 `string` に含まれる単語を切り出してリストとして返します。引数で指定した文字列を文字列メソッド `.lower()` で小文字化して `modefied_str` に代入し、さらに改行 `\n` やタブ `\t` をメソッド `.replace()` を使って空白文字列に置き換えます。さらに、単語の前後にあるカンマやピリオドなど単語を構成しない記号を文字列 `removing_character_string` として設定し、これらも空白文字として置き換えてしまいます。

コード 4.6-1　文字列に含まれる単語の切り出し

```
def modify_line(string):# 行から不要な記号を見つけて空白に置き換える
```

```
    removing_character_string = "'’ ()<>[]{}&/?!=—_:;,."
    modefied_str = string.lower()
    modefied_str = modefied_str.replace('"',' ')
    modefied_str = modefied_str.replace('\n','').replace('\t',' ')
    for s in removing_character_string:
        modefied_str = modefied_str.replace(s,' ')
    return(modefied_str)
```

結局、 `modify_line(string)` は引数で指定した文字列 `string` を空白で区切られた単語からなる文字列を返します（空白で区切られた文字列を「単語」とみなすということです)。ここでは簡単のために単数や複数形や動詞の活用などは異なる単語とみなしてしまいます。

プログラム 4.6-2 では、コマンドラインで指定したテキストファイルが `sys.argv[1]` にセットされ、文字符号 UTF-8 としてファイルを開いて 1 行ずつ `line` として読み込んでいます。空白で区切られた文字列 `modify_line(line)` をメソッド `.split()` によって単語として切り出された 1 行分の単語リストをリストとして `word_list` に結合しながらテキストに登場する全単語リスト `word_list` を得ています。

コード 4.6-2　テキストに登場する全ての単語からなるリスト word_list を得る

```
import sys

def modify_line(string):
    ....
    ....

fh = open(sys.argv[1], 'r', encoding = 'utf-8')
word_list = []
line = fh.readline()
while line:
    wlist = modify_line(line).split()
    word_list.extend(wlist)
    line = fh.readline()
fh.close()
```

プログラム 4.6-3 の関数 `make_word_frequency_dictionary(word_list)` は単語リスト `word_list` から、単語をキーにその登場回数を値とするディクショナリを返します。単語リストから 1 つづつ取り出した単語 `word` が現在のディクショナリのキーになっていれば値を+1 だけ更新、そうでなければ（キーになっていなければ）新たにディ

クショナリのキーを登録してその値を 1 に設定することを繰り返します。

コード 4.6-3　全単語リストから単語とその出現回数からなるディクショナリを作成

```
def make_word_frequency_dictionary(word_list):
    word_frequency_dict = {}
    for word in word_list:
        if word in word_frequency_dict:
            word_frequency_dict[word] += 1
        else:
            word_frequency_dict[word] = 1
    return(word_frequency_dict)
```

プログラム 4.6-4 は、コマンドラインからテキストファイルを読み込んで、登場する単語の出現順位とその出現頻度とを log-log プロットするプログラムです。ここでは、pandas を使って単語頻度ディクショナリ `word_frequency_dict` を Series `word_series` とし（27 行目）、さらに登場回数に列名 '`frequency`' を付けて DataFrame `word_df` としています（28 行目）。29 行目で pandas のメソッド `.sort_values` で'`frequency`' に関して降順で並べ替え、30 行目で DataFrame のインデックスを'`word`' に名称変更し、31 行目で 0 から始まる行インデックスに 1 を加えた数を列名 '`rank`' として追加して DataFramen `word_rank_df` としています。

32 行目では、メソッド `.to_csv()` を使って、単語 (word)、出現回数 (frequency)、順位 (rank) を列名とし、最左端に 0 からのインデックス番号を持つ 4 列からなる（データ区切りをカンマとする）CSV ファイルを書き出しています。最後に、40 行目で DataFrame `word_rank_df` の'`rank`' 列と'`frequency`' 列を NumPy の log 関数を使って両 log プロットしています。

コード 4.6-4　zipf_law_pandas.py

```
import sys
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def modify_line(string):
    ....
    ....

def make_word_frequency_dictionary(word_list):
    ....
```

```
    ....
#------  main script  -------
fh = open(sys.argv[1], 'r', encoding = 'utf-8')
word_list = []
line = fh.readline()
while line:
    wlist = modify_line(line).split()
    word_list.extend(wlist)
    line = fh.readline()
fh.close()

total_word_number = len(word_list)
print('Number of appeared words = ', total_word_number)
word_frequency_dict = make_word_frequency_dictionary(word_list)

word_series = pd.Series(word_frequency_dict)
word_df = pd.DataFrame(word_series, columns = ['frequency'])
sorted_df = word_df.sort_values(by = 'frequency', ascending=False)
wdf = sorted_df.reset_index().rename(columns={'index':'word'})
word_rank_df = wdf.assign(rank = wdf.index + 1)
print(word_rank_df)
word_rank_df.to_csv(sys.argv[1].replace('.txt','')+'_word_rank.csv')

fig = plt.figure()
ax = plt.axes()
ax.axis('equal')
ax.set(xlabel='log word rank', ylabel='log word frequency',
   title="Ranks vs their frequencies of " + sys.argv[1])
ax.plot(np.log(word_rank_df['rank']), np.log(word_rank_df['frequency'] /
    total_word_number), 'x', markersize=1)
#fig.savefig(sys.argv[1].replace('.txt','')+'_zipf_plot.png')
plt.show()
```

図 4.2 は、Project Gutenberg からチャールス・ディケンズの小説『オリバー・ツイスト』(https://www.gutenberg.org/ebooks/730）の本文を対象に調査して結果を示している。単語登場累計約 161,500 語について、登場単語の順位 k とその登場頻度 $p(k)$ の両対プロットはおおむね負の傾きを持つ直線上にあり、関係 $\log p(k) \sim s \log k \ (s > 0)$ であることが観察できます。

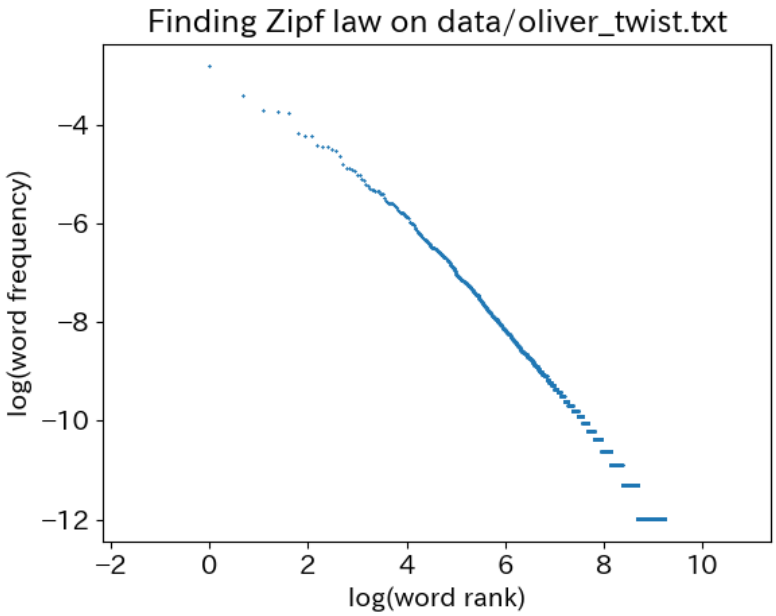


図 4.2　ディケンズの『オリバー・ツイスト』の登場する単語の順位 k とその登場頻度 $p(k)$ の両対数プロット。単語登場累計約 161,500 語について、プロット点はおおむね負の傾きを持つ直線上にある関係 $\log p(k) \sim s \log k$ $(s > 0)$ であることが観察できる。

第 5 章　微分方程式

5.1　連続曲線と接線

5.1.1　連続曲線

空間内の連続曲線 C を考えましょう。連続曲線とは、たとえば紙の上でペンを紙から離さず（ペンアップしないて切れ目なく）描いた曲線です。1 次元空間（直線）内の連続曲線は直線上を行きつ戻りつして描いた点の軌跡、2 次元空間（曲面）内の連続曲線は紙の上に描いた点の軌跡、3 次元空間内の連続曲線はジェットコースターのレール（周回軌道）を思い浮かべてください。。一般に n 次元空間 $\mathbb{R}^n$ 内で鉛筆の先が時刻と共に動きながら連続曲線 C が描かれると考えればよいでしょう。

連続曲線は時刻 t_a で描き始めた点 $C(a)$ から出発し、位置 $C(t)$ を変えながら時刻 $t = b$ で点 $C(b)$ に達するように描かれていると考えると、連続曲線 C を時間区間 $I = [a, b]$ から $\mathbb{R}^n$ への連続関数 $C : [a, b] \to \mathbb{R}^n$ と考えることができます。

$\mathbb{R}^n$ における時刻 t での連続曲線の位置 $C(t)$ を表すために、n 個の成分（それらの t の連続関数）を使って

$$C(t) = \begin{bmatrix} x_1(t) \\ \vdots \\ x_n(t) \end{bmatrix}$$

と表記します。このような曲線 C の表し方を時刻 t とパラメータとする表示と称します。

曲線（群）はパラメータ表示だけでなく、他の方法によっても表すことができます。平面曲線では位置を表す 2 つの座標、直交座標系 (x, y) または極座標系 (r, θ) に関する関係式で与えることもできます。図 5.1 は平面曲線が関係式

$$x^4 - 2a^2x^2 - 2b^2y^2 + y^4 - c = 0 \tag{5.1}$$

で与えられる平面曲線を示しています（ただし、$c = -b^4, a^2 > b^2$）。

図 5.1 は式 (5.1) のような関係式 $F(x, y) = 0$ を解いて陽な関係式 $y = f(x)$ を求めて関数のグラフとして描いたのでなく、平面上の実数値関数 $z = F(x, y)$ を高さ $z = 0$ での**等高線**

としてプロットしました。等高線を描くには matplotlib.pyplot の `contour` を使います。

プログラム 5.1-1: `implicit_cureve.py` では、x, y-平面上の格子点を `np.meshgrid` を使って x, y-格子行列 `xg, yg` として求め（節 2.2.3）、高さについての格子行列 `zg` に関して $z = 0$ の等高線を `contour` で描いています。

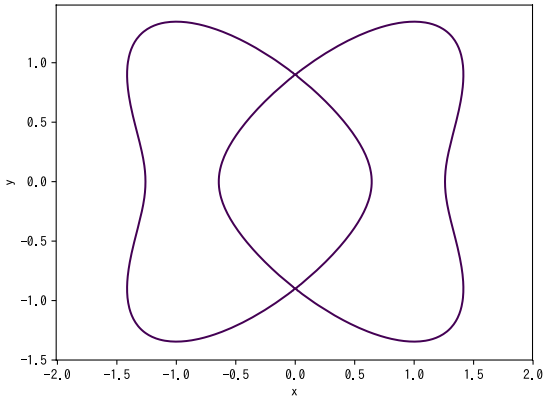


図 5.1　xy-平面上の曲線 $C : x^4 - 2a^2x^2 - 2b^2y^2 + y^4 - c = 0$（$c = -b^4, a^2 > b^2$）．曲線 C を陰関数 $F(x, y) = 0$ のままで描画するために matplotlib の等高線描画 `contour` を使って $z = F(x, y)$ の $z = 0$ での切り口としてプロット（プログラム??: `implicit_cureve.p`）

コード 5.1-1　implicit_cureve.py

```
import numpy as np
import matplotlib.pyplot as plt

delta = 0.015
width = 1.5
xRange = np.arange(-width, width, delta)
yRange = np.arange(-width, width, delta)
xg, yg = np.meshgrid(xRange, yRange)

# implicit curves
a = 1
b = 0.9
c = - b ** 4
zg = xg**4 - 2*a**2 * xg**2 - 2*b**2 * yg**2 + yg**4 - c

fig = plt.figure()
ax = plt.axes()
```

```
ax.axis('equal')
plt.contour(xg, yg, zg, [0]) # zg = 0 での等高線
ax.set(xlabel='x', ylabel='y')
#fig.savefig('implicit_cureve.eps')
fig.show()
```

5.1.2 滑らかな曲線

曲線を構成する点が途中で途切れることなくつながっていれば連続曲線となりますが、図 5.2 はブラウン運動のシミュレーションの様子です。水中浮遊する花粉が膨らんで破裂して流れ出た微粒子が時間経過に際して不規則に動くことを発見した Brown にちなんで名付けられた花粉の動きをブラウン運動と称します。描かれた曲線は確かに 1 本の連続曲線ですが、ジグザクを成した曲線の連なりで滑らかな曲線とはいえません。

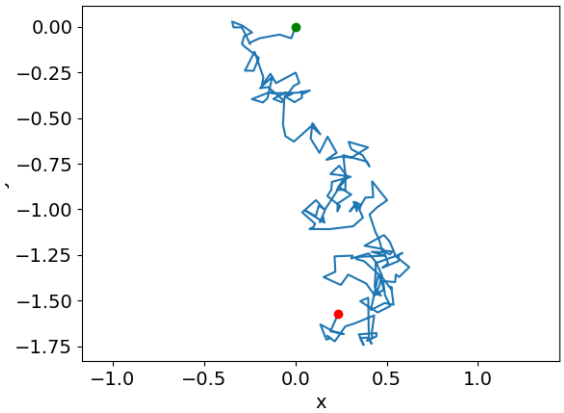


図 5.2　xy-平面上のブラウン運動のシミュレーション。液体に浮遊する微粒子が液体分子の熱運動に由来する衝突によって時間と共に不規則にジグザグ運動する様子を、乱数を使って出発点の丸印から直線で結ばれた連続曲線としてプロット。

滑らかな連続曲線とは何でしょうか。曲線の滑らかさは曲線が**微分可能** (differentiable) であることです。以下、さらに詳しい議論は微積分学の書籍を参照してください。連続曲線 $C : [a, b] \to \mathbb{R}^n$ が $t \in [a, b] \subset \mathbb{R}$ で微分可能であるとは次のような極限が成立することです。

$$\lim_{\substack{h\to 0\\ h\neq 0}} \frac{C(t+h) - C(t)}{h}, \qquad t \in [a, b]. \tag{5.2}$$

式 (5.2) で h は正負どちらかからでも 0 に近づいても極限を持たねばならないことに注意してください。極限 (5.2) が存在するとき、それを連続曲線 $C(t)$ の t における**微分**といい

$$\frac{dC}{dt}(t)$$

と表記します。曲線 C が定義区間 $[a,b]$ の各点で微分可能であるとき、関数 $\dfrac{dC}{dt} : I \to \mathbb{R}^n$

$$t \mapsto \frac{dC}{dt}(t)$$

が定義できます。これを C の**導関数** (derivative) といい、座標成分で書くと次のように表されます。

$$\frac{dC}{dt}(t) = \begin{bmatrix} \dfrac{dx_1(t)}{dt} \\ \vdots \\ \dfrac{dx_n(t)}{dt} \end{bmatrix}. \tag{5.3}$$

さて、連続曲線 $C : I \to \mathbb{R}^n$ が t で微分可能なとき、その導関数 $\dfrac{dC}{dt}(t)$ を時刻 t における曲線の位置 $C(t)$ における**接ベクトル**と見ることができます。平面曲線の場合、$\dfrac{dC}{dt}(t_0)$ は点 $C(t_0)$ を通る傾き $\frac{dx_2}{dt}/\frac{dx_1}{dt}$ を有する**接線** (tangent) の式 $\ell(s)$

$$\ell(s) = C(t_0) + \frac{dC}{dt}(t_0)s, \qquad s \in [a,b] \tag{??'}$$

を与えます。図 5.3 は平面上の曲線の点 $C(t_0)$ における接ベクトルを矢印で表しています。

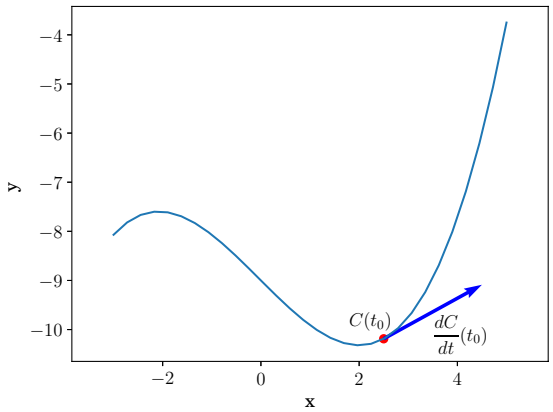


図 5.3　xy-平面上の曲線 $C(t)$ の場所 $C(t_0)$ における接ベクトル $\dfrac{dC}{dt}(t_0)$.

図 5.4 は、x, y-成分をそれぞれ $x(t) = a\cos k_x t$ と $y(t) = b\sin k_y t$ とするリサージュ曲線 $C(t) = (x(t), y(t))$ の様子を表しています。定数 k_x と k_y の比が非有理数（$k_x/k_y \notin \mathbb{Q}$）のとき、リサージュ曲線は長方形領域を稠密に埋め尽くします。リサージュ曲線の接ベクトルは次のようになります。

$$\begin{cases} \dfrac{dx}{dt} = \dfrac{k_x a}{b} y \\ \dfrac{dy}{dt} = -\dfrac{k_y b}{a} y \end{cases} \tag{5.4}$$

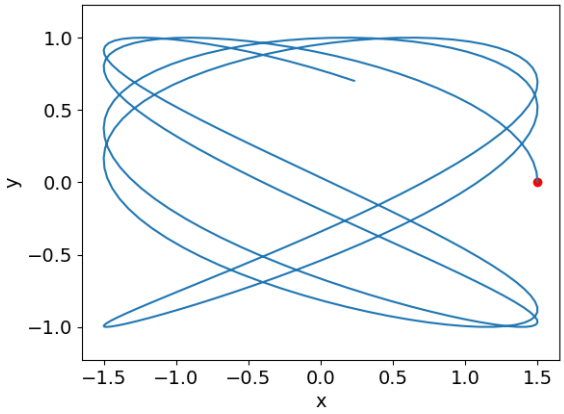


図 5.4　リサジュー曲線 $C(t) = (xa\cos k_x t, b\sin k_y t)$ の軌跡（$a = 1.5, b = 1, \omega_1 = 1, \omega_2 = 1/\sqrt{2}$)。時刻 $t = 0$ での出発点は赤丸で示した。この例のように比 k_x/k_y が無理数のとき、リサジュー曲線は時間経過につれて長方形領域を稠密に埋め尽くしていく。

5.2　微分方程式系と相空間

いささか形式的ですが、まず用語を準備しておきましょう。具体例は次の節で紹介します。

時刻 t の n 個の関数 $x_1(t), x_2(t), \ldots, x_n(t)$ が次のような関係式

$$\begin{aligned} \frac{dx_1}{dt} &= v_1(x_1, \ldots, x_n, t), \\ \frac{dx_2}{dt} &= v_2(x_1, \ldots, x_n, t), \\ &\vdots \\ \frac{dx_n}{dt} &= v_n(x_1, \ldots, x_n, t), \end{aligned} \tag{5.5}$$

$$x_1(t_0) = x_{10}, x_2(t_0) = x_{20}, \ldots, x_n(t_0) = x_{n0} \tag{5.6}$$

を満たしているとき、式 (5.5) を変数 $x_1, \ldots, x_n$ についての**常微分方程式系** (system of ordinary differential eqatuons) と称し、式 (5.5)′ を式 (5.5) の時刻 t_0 における**初期値** (initial values) あるいは初期条件といいます。変数の組 $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ が動きまわる空間を**相空間** (phase space) といい、相空間内の 1 点を定めるために必要な成分数（ここでは $x_1, \ldots, x_n$ の n 個）を相空間の**次元**といいます。

式 (5.8) を関数 $x_1(t), \ldots, x_n(t)$ が満たすべき方程式、つまり、ある関数群が存在して式 (5.8) を満たしていると見るのでなく、ある曲線 $\boldsymbol{x}(t) = (x_1(t), \ldots, x_n(t))$ の接ベクトルが式 (5.8) の右辺で与えられているとみなすのが幾何学的な立場です。$\boldsymbol{x}(t)$ を微分方程式式 (5.8) の解曲線または**解軌道**といいます。

式 (5.5) において、次にように実数値関数 $x_i(t), v_i(\boldsymbol{x}, t)$ を列ベクトル

$$\boldsymbol{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix}, \quad \boldsymbol{V}(\boldsymbol{x}, t) = \begin{bmatrix} v_1(\boldsymbol{x}, t) \\ v_2(\boldsymbol{x}, t) \\ \vdots \\ v_n(\boldsymbol{x}, t) \end{bmatrix}, \quad \boldsymbol{x}(t_0) = \begin{bmatrix} x_{10} \\ x_{20} \\ \vdots \\ x_{n0} \end{bmatrix} \tag{5.7}$$

で表して、常微分方程式系 (5.5) を次のように簡素化して表します。

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{V}(\boldsymbol{x}, t), \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{5.8}$$

このとき、$\boldsymbol{V}(\boldsymbol{x}, t)$ を常微分方程式系 (5.8) を定める**ベクトル場** (vector field) と呼びます。

式 (5.8) をベクトル場 $\boldsymbol{V}(\boldsymbol{x}, t)$ に接する（接ベクトルがベクトル場で与えられている）解曲線を定義しているとみなすと、相空間 $\mathcal{M}$ 上の点 $\boldsymbol{x}_0$ を $t = 0$ で出発し、時間 t 経過後に点 $\boldsymbol{x}_t$ に達する $\mathcal{M}$ 上の時間発展 $F_t : \boldsymbol{x}_0 \mapsto \boldsymbol{x}_t$ を自然に考えることができます。相空間 $\mathcal{M}$ 上の時間発展 $F_t : \mathcal{M} \to \mathcal{M}$ は、$\boldsymbol{x}_0$ から s 経過後にさらに t 経過した場所が $\boldsymbol{x}_0$ から時刻 $t + s$ 経過した場所に等しいことから

$$F_t \circ F_s = F_{t+s}, \quad F_0 = \text{恒等写像}$$

の性質を持ちます。$\mathcal{M}$ 上の時間発展 F_t をベクトル場 (5.8) で定められる**流れ** (flow) と称します。相空間 $\mathcal{M}$ 上にベクトル場が定められていると、$\mathcal{M}$ に置いた 1 点 $\boldsymbol{x}_0$ はベクトル場に沿って流されていくと捉えるのです。ベクトル場上の流れについて節 5.3 で改めて説明します。

式 (5.8) で定まる流れ $\boldsymbol{x}(t, \boldsymbol{x}_0)$ は初期値 $\boldsymbol{x}_0$ に応じてその後の振る舞いが異なります。

式 (5.8) の解軌道 $\boldsymbol{x}(t, \boldsymbol{x}_0)$ において、

$$\omega(\boldsymbol{x}_0) = \{\boldsymbol{z} \in \mathbb{R}^n \mid \text{増加点列 } \{t_n\} \text{ が存在して、} \lim_{t_n \to \infty} \boldsymbol{x}(t_n, \boldsymbol{x}_0) = \boldsymbol{z}\} \tag{5.9}$$

が存在するとき、$\omega(\boldsymbol{x}_0)$ を $\boldsymbol{x}(t, \boldsymbol{x}_0)$ の **ω-極限集合**といいます（節 3.5.2 も参照）。時間の向きを取り替えて、同様に α-極限集合も定義されます。

ベクトル場（すなわち微分方程式）が与えられたら、まず定点（時間とともに動かない点）やその周辺部分の流れの様子を調べます。さらに周期軌道が存在するかや ω-極限集合についてはどうなのかも探ります（これらは初期条件にも依存しますから簡単な作業ではありません）。

解曲線を求めることを微分方程式を**解く**（solve）、その手続きを**求積**（quadrature）と言います。残念ながら、ベクトル場が**線形**（$\boldsymbol{V}(\alpha\boldsymbol{x} + \beta\boldsymbol{y}, t) = \alpha\boldsymbol{V}(\boldsymbol{x}, t) + \beta\boldsymbol{V}(\boldsymbol{y}, t)$ であるなどの例外的な場合を除いて、一般に微分方程式の解を数学的に求めることは不可能で、その軌道の振る舞い、特に長時間の時間経過にわたる大域的な振る舞いを知ることは数学的にはたいへん困難です。そこで、本来は連続曲線である微分方程式の解を離散的な数値列として求める**数値解法**が実用的にも理論的探求にも重要な意味を持つことになります。

5.2.1 線形微分方程式

常微分方程式を定めるベクトル場は一般的には式 (5.8) のように位置 $\boldsymbol{x}$ だけでなく時刻 t の関数 $\boldsymbol{V}(\boldsymbol{x}, t)$ です。

ベクトル場がその座標成分 $x_1, \ldots, x_n$ に関して高々 1 次関数であるとき、つまり正方行列 $A(t)$ を使って

$$\frac{d\boldsymbol{x}}{dt} = A(t)\boldsymbol{x} + \boldsymbol{b}(t) = \begin{bmatrix} a_{11}(t) & \ldots & a_{1n}(t) \\ \vdots & \ddots & \vdots \\ a_{n1}(t) & \ldots & a_{nn}(t) \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1(t) \\ \vdots \\ b_n(t) \end{bmatrix} \tag{5.10}$$

と表されるとき、**線形非斉次微分方程式** (linear non-homogeneous differential equation) といいます。$\boldsymbol{b}(t) = \boldsymbol{0}$ のとき、

$$\frac{d\boldsymbol{x}}{dt} = A(t)\boldsymbol{x} \tag{5.11}$$

を線形斉次方程式 (LH) または単に**線形微分方程式**といいます。とくに、$A(t) =$ 定数 の場合を定常線形微分方程式といいます。

行列 $\Phi(t)$ の n 個の列が線形微分方程式 (5.11) の線形独立な解となっているとき、行列

方程式

$$\frac{d\Phi(t)}{dt} = A(t)\Phi(t)$$

を満たしています。このような $\Phi(t)$ を線形微分方程式の**基本行列**といいます。

定理 5.1行列 $\Phi(t)$ は線形微分方程式の基本解であるための必要十分条件は

$$\det \Phi(t) \neq 0$$

となることである。

斉次微分方程式の基本行列が知られているとき、定数変化法によって非斉次微分方程式の解を求めることができます。

定理 5.2　行列 $\Phi(t)$ を基本解とすると

$$\Psi(t) = \Phi(t) \int_{\tau}^{t} \Phi^{-1}(s)\boldsymbol{b}(s)\, ds$$

は $\Psi(\tau) = 0$ を満たす線形非斉次微分方程式 (5.10) の解行列である。

行列 $A(t)$ が定数行列 A の場合、線形微分方程式 (5.11) は

$$\frac{d\boldsymbol{x}}{dt} = A\boldsymbol{x} \tag{5.11′}$$

であることが成り立ちます。このとき、基本行列は

$$\Phi(t) = \mathrm{e}^{At} \tag{5.12}$$

になります。ここで、行列 At の指数関数 e^{At} は

$$\mathrm{e}^{At} = I + At + \frac{A^2t^2}{2!} + \cdots + \frac{A^nt^n}{n!} + \ldots$$

で定義されます。これより、初期条件 $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ を満たす (5.11)′ の解は

$$\boldsymbol{x}(t) = \mathrm{e}^{A(t-t_0)}\boldsymbol{x}_0$$

で与えられます。正方行列 A が与えられたときの具体的な数値計算および記号計算の実際については行列の指数関数（節 2.6）を参照してください。

線形微分方程式の流れ e^{At}（式 (5.12)）は行列 A の固有値によって完全に決定してしまうことが知られています。固有値が純虚数（実部を持たない）となる場合、複素共役となる関係を成す一組の固有として表れます。その 2 次元部分空間で軌道は原点のまわりを回転し、原点を**楕円不動点** (ellipticd fixed point) といいます。もっとも一般的な流れは、A のすべての固有値が 0 でない実部を持つ場合で**双曲的な流れ** (hyperboloc flow) です。このと

き、原点を**双曲不動点** (hyperbolic fixed point) といいます。A が実標準形になるようなベクトル空間 $E = \mathbb{R}^n$ の基底をとって、基底の順序をうまく選んで標準形の行列がはじめに負の固有値に対応するブロックを、その後に性の実部を持つ固有値に対応するブロックが並ぶようにできます。

定理 5.3　e^{At} が双曲的な流れであるとき、E は A-不変な E_s を安定部分空間および E_u を不安定部分空間とする直和分解

$$E = E_S \oplus E_u$$

を持つ。e^{At} を E_s に制限して得られる流れは縮小的流れ（$t \to -\infty$ のとき任意の軌道が $-\infty$ に向かう）になり、e^{At} を E_u に制限して得られる流れは拡大的流れ（$t \to \infty$ のとき任意の軌道が ∞ に向かう）となる。

線形方程式の詳しい議論およびベクトル場の不動点（$\boldsymbol{V}(\boldsymbol{x}^*) = \boldsymbol{0}$ となる $\boldsymbol{x}^*$）の周りの流れについて是非 [12][14] を参照してください。ベクトル場の不動点については節 6.2 で改めて取り上げます。

5.2.2　1 変数高階微分方程式の取扱い

光学や物理学などに登場する微分方程式では、求める解 $x(t)$ が高階の導関数を含むような次の形の 1 変数高階微分方程式を扱う場合があります。

$$\frac{d^n x}{dt^n} + a_1(t)\frac{d^{n-1}x}{dt^{n-1}} + a_2(t)\frac{d^{n-2}x}{dt^{n-2}} + \cdots + a_{n-1}(t)\frac{dx}{dt} + a_n(t)x = 0. \tag{5.13}$$

こうした場合、新しい変数 $x_1, x_2, \ldots, x_n$

$$x_1 \equiv x, x_2 \equiv \frac{dx}{dt}, \ldots, x_n \equiv \frac{d^{n-1}x}{dt^{n-1}}$$

を導入することによって、式 (5.5) または (5.8) の形、n 変数 1 階線形常微分方程式系に帰着させることができます。

$$\begin{aligned}
&\frac{dx_1}{dt} = x_2, \\
&\frac{dx_2}{dt} = x_3, \\
&\vdots \\
&\frac{dx_{n-1}}{dt} = x_n, \\
&\frac{dx_n}{dt} = -a_n(t)x_1 - a_{n-1}(t)x_2 - \cdots - a_1(t)x_n
\end{aligned} \tag{5.14}$$

式 (5.14) は次の n 次正方行列

$$A(t) = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & & 0 \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & 1 \\ -a_n & -a_{n-1} & \dots & & -a_1 \end{bmatrix} \tag{5.15}$$

を使って、次のように書き直すことができます。

$$\frac{d\boldsymbol{x}}{dt} = A(t)\boldsymbol{x}, \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. \tag{5.16}$$

行列 (5.15) の特性多項式 $p(\lambda)$ は

$$p(\lambda) = \lambda^n + a_1\lambda^{n-1} + a_2\lambda^{n-2} + \cdots + a_n \tag{5.17}$$

となります。微分方程式 (5.13) の係数 $\{a_i\}$ が定数のとき、その特性多項式 (5.17) の固有値を求めることによって、線形微分方程式 (5.16) の解を書き下すことができます [14]。

5.3 ベクトル場が定める解曲線

5.3.1 ベクトル場

ベクトル場がどんなものかを把握するために、まず xy-平面上の線形なベクトル場として調和振動子（バネ定数 $k = \omega^2$）のバクトル場

$$\boldsymbol{V}(\boldsymbol{x}) = \begin{bmatrix} y \\ -\omega^2 x \end{bmatrix} \tag{5.18}$$

を考えてみます。図 5.5 は、$\omega = 0.8$ として平面上の格子点 $\boldsymbol{x} = (x, y)$ を足場とする矢印を接ベクトルとして描くことでベクトル場 $\boldsymbol{V}(\boldsymbol{x})$ を表しています（リサージュ曲線を与える式 (5.4) 参照）。たとえば点 $(0.4, 1.2)$ における接ベクトルは $(1.2, -0.8^2 \times 0.4)$ となります。

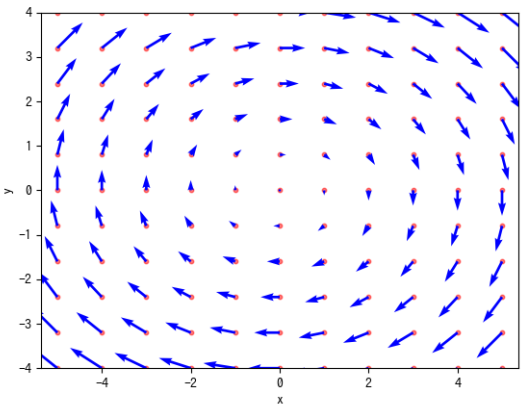


図 5.5　調和振動子ベクトル場 $\boldsymbol{V}(\boldsymbol{x})$。平面上の各点 $\boldsymbol{x}$ で接ベクトル $\boldsymbol{V} = (y, -\omega^2 x)(\omega = 0.8)$ を x, y-区間 $[-4, 4]$ をそれぞれ 10 分割した格子点（赤）での接ベクトルを矢印（青）として描いた。

与えられたベクトル場 $\boldsymbol{V}(\boldsymbol{x})$ に接する曲線 $C(t)$ を考えてみましょう。節 5.1.2 で説明したように、$C(t)$ は時刻 t_0 で点 $\boldsymbol{x}_0$ を出発して時刻 t で点 $\boldsymbol{x}(t)$ に到達したとき、その接ベクトル $dC(t)/dt$ が $\boldsymbol{V}(\boldsymbol{x}(t))$ となっている曲線です。言い換えれば、曲線 $C(t)$ 上の位置をベクトル $\boldsymbol{x}(t)$ で表したとき、次の曲線は次の関係を満たします。

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{V}(\boldsymbol{x}(t)), \qquad \boldsymbol{x}(t_0) = \boldsymbol{x}_0.$$

これは式 (5.8) に他なりません。つまり、ベクトル場で定まる微分方程式の解は、初期値から出発してベクトル場に接する解曲線となっています。

図 5.6 は、図 5.5 で示されたベクトル場において原点中心に楕円

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1, \quad \text{ただし } b = \omega a \tag{5.19}$$

を 2 つ描いたもので、いずれもベクトル場に接している様子が見て取れます。実際、この楕円は式 (5.18)（ばね定数 $k = \omega^2$ を持つ調和振動子の微分方程式 (5.3.2.1)）の解曲線になっています。実際、$E(x, y) = \dfrac{x^2}{a^2} + \dfrac{y^2}{b^2}$ を時間で微分して式 (5.18) を使うと

$$\frac{d}{dt}E(x, y) = xy\left(\frac{1}{a^2} - \frac{\omega^2}{b^2}\right) = 0$$

となり、$E(x, y) =$ 定数 であることが成り立ちます。これより、楕円はベクトル場 (5.18) で定まる解曲線であることが確かめられました。

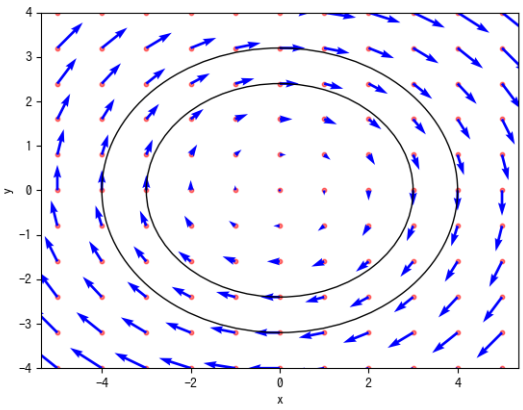


図 5.6　ベクトル場 $\boldsymbol{V}(\boldsymbol{x})$ に接する曲線の例。2 つの楕円 $x^2/a^2 + y^2/b^2 = 1 (b = \omega a)$ は図 5.5 で表されるベクトル場 $\boldsymbol{V} = (y, -\omega^2 x)$ に接している。実際、楕円は式 (5.18) で定まるベクトル場の解曲線になっている。

5.3.2　ベクトル場の様子を描画する matplotlib.pyplot.quiver

ベクトル場とは、各点でベクトル（向きと大きさ）が付与されている空間でした（節 5.3.1）。matplotlib.pyplot.quiver は、点 (x, y) を起点とする矢印付きの 2 次元ベクトル (u, v) または (x, y, z) を起点とする 3 次元ベクトル (u, v, w) を描きます。

```
matplotlib.pyplot.quiver(x,y,z, u,v,w, options)
```

以下では、ベクトル場の様子をグリッドの各点における接ベクトルとして描いています。

5.3.2.1　調和振動子

ばね定数 $k = \omega^2$ を持つ調和振動子の微分方程式は

$$\begin{cases} \dfrac{dx}{dt} = y \\ \dfrac{dy}{dt} = -\omega^2 x \end{cases} \tag{5.20}$$

です（節 5.3 参照）。調和振動子はその解を厳密に解くことができる簡単な方程式ですが（解軌道は節 5.3.1 で確かめたように楕円を描きます）、物理学的には大変重要で随所に現れます。調和振動子の微分方程式が定めるベクトル場は、プログラム 5.3-1: `harmonic_vectorfield.py` を使って図 5.5 のようになります。

プログラム 5.3-1: `harmonic_vectorfield.py` では、格子点の x, y-格子点行列 `xg` と `yg` から、2 要素（のベクトル場成分）を返す関数 `harmonic` の引数にそのまま与えて、

ブロードキャストによって接ベクトルを成す x, y-格子点行列 vx と vy を計算しています（すべて同じ shape を持ちます）。quiver にそのままこれらの格子点行列を各点での接ベクトルをしています。

コード 5.3-1　harmonic_vectorfield.py

```
import numpy as np
import matplotlib.pyplot as plt

def harmonic(x, y: float, w: float) -> tuple:
    vx = y
    vy = - w**2 * x
    return(vx, vy)
w = 0.8

xwid = 5
ywid = 4
meshSize = 11
gx, gy = np.meshgrid(
    np.linspace(-xwid, xwid, meshSize),
    np.linspace(-ywid, ywid, meshSize)) # shape (meshSize, meshSize)
vx, vy = harmonic(gx, gy, w) # vectors shape (meshSize, meshSize)

fig = plt.figure()
ax = plt.axes()
ax.axis('equal')
ax.scatter(gx, gy, s=10, c= 'red',alpha=0.5)# 格子点
ax.quiver(gx, gy, vx, vy, color='b', units='width')# 格子点からの接ベクトル
ax.set(xlim=(-xwid, xwid), ylim=(-ywid, ywid),
       xlabel='x', ylabel='y')
#fig.savefig('harmonic_vectorfield.pdf')
fig.show()
```

5.3.2.2　Van der Pol の微分方程式

2 次元ベクトル場の様子はプログラム 5.3-1 において接ベクトル場を与える関数を（調和振動子の harmonic から）取り替えることで同様にして描くことができます。電気回路の発振現象でしばしば登場する Van der Pol 方程式はパラメータ $\mu > 0$ を含む非線形方

程式

$$\begin{cases} \dfrac{dx}{dt} = y \\ \dfrac{dy}{dt} = -x + \mu(1 - x^2)y, \qquad \mu > 0 \end{cases} \tag{5.21}$$

で表されます。Van der Pol の微分方程式が定めるベクトル場は図 5.7 のようになります。

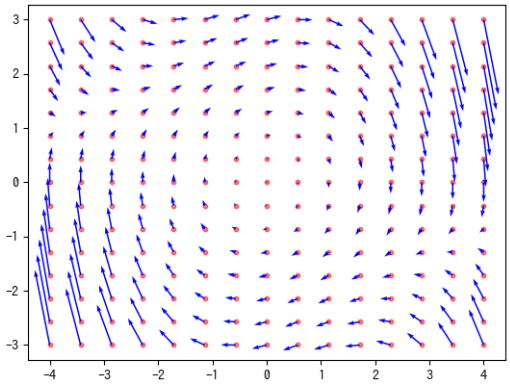


図 5.7　Van der Pol の微分方程式 (5.21) で与えられるベクトル場（$\mu = 0.25$）の様子。

5.3.2.3　Lotka-Volterra の微分方程式

捕食-被捕の様子を記述する Lotka-Volterra の微分方程式は非線形で

$$\begin{cases} \dfrac{dx}{dt} = x - xy \\ \dfrac{dy}{dt} = -y + xy \end{cases} \tag{5.22}$$

と表されます。Lotka-Volterra の微分方程式が定めるベクトル場の様子は図 5.8 のようになります。

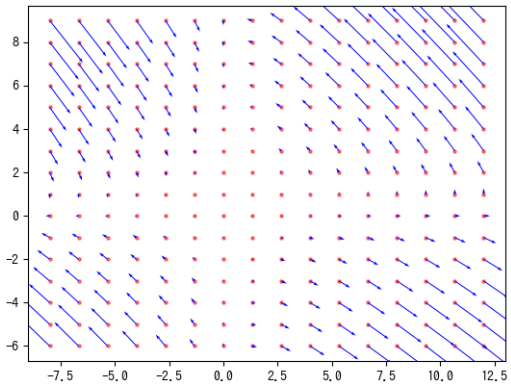


図 5.8　Lotka-Volterra の微分方程式 (5.22) で与えられるベクトル場の様子。

5.3.2.4　Lorenz 系のベクトル場

第**??**章で詳しく紹介する Lorenz 方程式は最も深く研究されてきた非線形常微分方程式で次で与えられます。

$$\begin{cases} \dfrac{dx}{dt} = -\sigma x + \sigma y, \\ \dfrac{dy}{dt} = rx - y - xz, \\ \dfrac{dy}{dt} = -bz + xy. \end{cases} \tag{5.23}$$

通常、$\sigma = 10, b = 8/3$ としぃぇ固定し、$r > 0$ をパラメータとして考えます。

コード 5.3-2　lorenz_vectorfield.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# parameters of Lorenz system
p =10
b = 8.0 / 3.0
r = 28
def lorenz(x, y, z: float, p, b, r: float)-> tuple:
    vx = -p * x + p * y
    vy = -x * z + r * x - y
    vz = x * y - b * z
```

```
    return(vx, vy, vz)

x = np.linspace(-50, 50, 9)
y = np.linspace(-50, 50, 9)
z = np.linspace(0, 60, 8)
xg, yg, zg = np.meshgrid(x, y, z)
vx, vy, vz = lorenz(xg, yg, zg, p, b, r)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
#ax.axis('equal')
scale = 0.01# scaling of vectors
ax.quiver(xg, yg, zg, scale * vx, scale * vy, scale * vz)
ax.scatter3D(xg, yg, zg, s=5, c= 'red',alpha=0.5)
ax.set(xlabel='x', ylabel='y', zlabel='z')
#fig.savefig('lorenz_vectorfield.png')
fig.show()
```

プログラム 5.3-2 は、$r = 28$ としたときの 3 次元ベクトル場の様子を矢印を図 5.9 のように描くためのものです。微分方程式の軌道はベクトル場に接するように移動した曲線となりますが、第**??**章でわかるように長時間に渡る軌道の様子をベクトル場から推測することは大変むつかしく、初期点のわずかな差異がその後の挙動に大きな影響を及ぼすという**初期条件鋭敏性**を呈します。

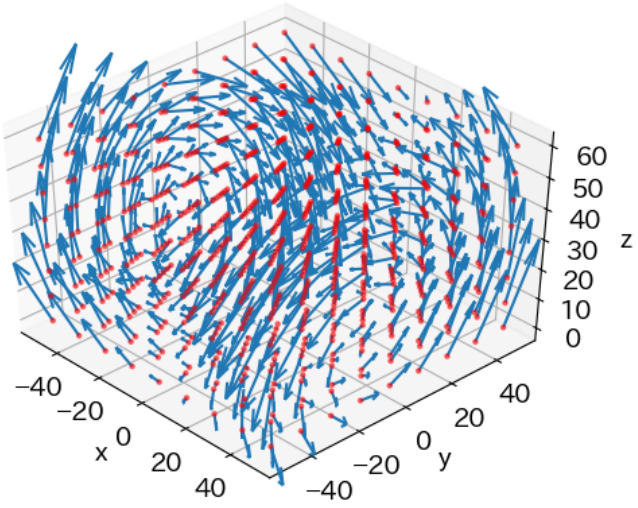


図 5.9　Lorenz の微分方程式 (5.23) で与えられる 3 次元ベクトル場の様子。$\sigma = 10, b = 8/3, r = 28$**.**

5.4　微分方程式の数値解法

これまで、ベクトル場が与えられたとき、それが定める微分方程式の解曲線とはベクトル場に接している曲線であることを説明してきました。しかしながら、与えられたベクトル場に接する曲線を完全に決定することは実際には大変困難で、線形方程式や求積可能な特別な場合を除いて一般的には数値的に近似解を計算するしか方法がありません。

ここでは、次の微分方程式系

$$\begin{aligned}
\frac{dx_1}{dt} &= v_1(x_1, \ldots, x_n, t), \\
\frac{dx_2}{dt} &= v_2(x_1, \ldots, x_n, t), \\
&\vdots \\
\frac{dx_n}{dt} &= v_n(x_1, \ldots, x_n, t)
\end{aligned} \tag{5.5′}$$

または、ベクトル表記 $\boldsymbol{x}(t) = (x_1, \ldots, x_n)$ して

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{V}(\boldsymbol{x}, t) \tag{5.8′}$$

を初期値

$$\boldsymbol{x}(t_0) = \boldsymbol{x}_0 = (x_{10}, x_{20}, \ldots, x_{n0}) \tag{5.6'}$$

のもとで、離散的な数値列を求める数値解法を考えます。

時間区間 $I = [a, b]$ における数値解 $\boldsymbol{x}(t)$ を求めるために、I を次のように N 分割（等分割でなくてもよい）

$$a = t_0 < t_1 < t_2 < \cdots < t_{N-1} < t_N = b$$

して、与えられた微分方程式の初期値 $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ のもとで各離散時間 t_k における数値列 $\{\boldsymbol{x}_k\}_{k=1,2,\ldots,N}$ を計算する方法を微分方程式の初期値問題に関する**数値解法**といいます。数値列 $\{\boldsymbol{x}_k\}$ が真の解 $\boldsymbol{x}(t_k)$ をどのように近似しているかが問題となります。ここでは、時間区間を等分割して時間刻み Δt を

$$h = \frac{b-a}{N}, \qquad t_k = a + kh, \tag{5.24}$$

とし、次の $\boldsymbol{x}_{k+1}$ は現在位置 $\boldsymbol{x}_k$ から差分方程式

$$\boldsymbol{x}_{k+1} = \boldsymbol{F}(\boldsymbol{x}_k, h) \tag{5.25}$$

によって逐次的に求める方法を考えます。$\boldsymbol{x}_{k+1}$ を現在位置 $\boldsymbol{x}_k$ から過去にさかのぼって $\boldsymbol{x}_{k-\ell}$ までを使って $\boldsymbol{x}_{k+1} = \boldsymbol{F}(\boldsymbol{x}_k, \ldots, \boldsymbol{x}_{k-\ell}, h)$ を考えることも可能です。いずれにせよ、いかにして真の解に近い数値解列 $\{\boldsymbol{x}_i\}$ を見出すかが数値解法の課題となります。

任意の初期値から出発した流れが時間経過後に相空間内でどのような挙動をしていくのかかという**大域的性質**は力学系の理論の中心的テーマの 1 つです。非線形微分方程式では例外的な場合を除いて一般的には真の解を求積することができないため、工夫をこらして得られた数値解が真の解のどんな性質をどの程度近似しているかを知ることが大変重要になります。

初期値を出発した数値軌道と真の解軌道との誤差を時間の関数として評価する一般理論は知られておらず、限られた時間区間や時間刻み幅のとり方に関する計算精度の向上は興味深い研究対象になります [10][11]。数理モデルとして微分方程式で表される現象は自然科学だけでなく社会科学を含む広範な領域に及ぶため、その数値解が大きな意味をもつ場合が少なくありません。

5.4.1 Euler 法

式 (5.8)′ の左辺の微分

$$\frac{d}{dt}\boldsymbol{x}(t) \equiv \lim_{h \to 0} \frac{\boldsymbol{x}(t+h) - \boldsymbol{x}(t)}{h}$$

において極限操作 $h \to 0$ をとらずに、十分小さい有限の $h > 0$ に留めると近似的に次が成立します。

$$\boldsymbol{x}(t+h) \approx \boldsymbol{x}(t) + h\boldsymbol{V}(\boldsymbol{x}(t), t) \tag{5.26}$$

離散時間列 $\{t_k\}$ に対してこの近似が立しているとすると次の関係式を得ます。

$$\boldsymbol{x}(t_{i+1}) \approx \boldsymbol{x}(t_i) + h\boldsymbol{V}(\boldsymbol{x}(t_i), t_i). \tag{5.27}$$

このような素朴な考察から、時刻 t_k での値 $\boldsymbol{x}(t_k)$ を $\boldsymbol{x}_k$ と記したとき、初期値 $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ からの時刻 t_{k+1} での値 $\boldsymbol{x}_{k+1}$ を逐次反復して

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h \cdot \boldsymbol{V}(\boldsymbol{x}_k, t_k), \qquad k = 0, 1, 2, \ldots \tag{5.28}$$

によって定める計算法を **Euler 法**といいます。ただし、ここで取り扱う数値解全体に言えることですが、初期値 $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ 以降の点 $\boldsymbol{x}_k$（$k > 0$）は初期値 $\boldsymbol{x}_0$ だけから決まっており、式 (5.27) が成立しているからといって真の解 $\boldsymbol{x}(t_k)$ と $\boldsymbol{x}_k$ とが近い $\boldsymbol{x}(t_k) \approx \boldsymbol{x}_k$ とは限りません。

節 5.3.2.1 の調和振動子の微分方程式 (5.20) を Euler 法を使ったプログラム 5.4-1: `simple_euler_diff_method.py` によって描いた軌道の様子を図 5.10 に示しました。調和振動子の解軌道は節 5.3.1 の式 (5.19) で示したように、任意の初期値を載せる楕円になります（図 5.6）。図 5.10 では、初期値 $(1, 0)$ を出発した軌道は時間経過と共に時計回りに原点から離れるように渦巻状をなし、真の解軌道から逸脱していく様子を示していています。Euler 法を使って計算した数値軌道は真の解軌道の様子を表しているとはいえません（こうした断定は真の解の性質が完全にわかっているために言えるわけです）。

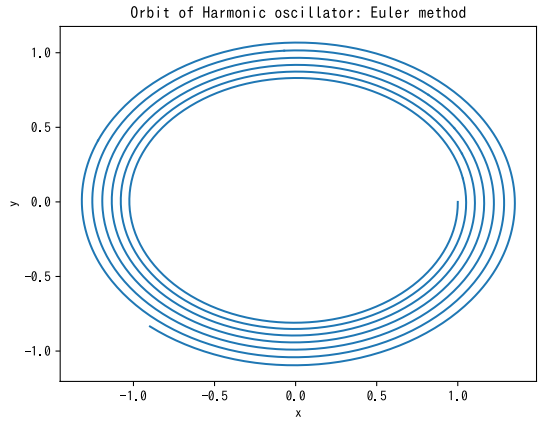


図 5.10　Euler 法を使って数値計算した調和振動子の微分方程式 (5.20)（$\omega = 0.8$）の解軌道。時間刻み幅 $h = 0.02$ で時間区間 $[0, 100]$ までの軌道を初期条件 $x_0 = 1, y_0 = 0$ から計算。Euler 法では時間刻みを細かくしても計算誤差が積み重なり、時間経過と共に真の解軌道から逸脱していく。

コード 5.4-1　simple_euler_method.py

```
import numpy as np
import matplotlib.pyplot as plt
from typing import Callable # 関数引数の型宣言

def harmonic_force(x: float, w: float) -> float:
    vy = -w ** 2 * x
    return(vy)

def euler_method(x, y: float, f: Callable[[float,float],float], dt, w: float
    )->tuple:
    x1 = x+ dt * y
    y1 = y + dt * f(x, w)
    return(x1, y1)

w = 0.8 # ばね定数 k = w **2
x0, y0 = 1.0, 0.0 # 初期値
dt = 0.02 # 時間刻み
t = 0.0   # 経過時間
T = 50    # 時間幅
orbit = np.empty((0,2), float)
while t < T:
    orbit = np.append(orbit, [[x0, y0]], axis = 0)
    x0, y0 = euler_method(x0, y0, harmonic_force, dt, w)
    t += dt

fig, ax = plt.subplots()
#ax.axis('equal')
ax.set(xlabel='x', ylabel='y',
       title='Orbit of Harmonic oscillator: Euler method')
ax.plot(orbit[:,0],orbit[:,1])
#fig.savefig('simple_euler_method.png')
fig.show()
```

調和振動子の数値解軌道を計算するプログラム 5.4-1: `simple_euler_diff_method.py` では、関数引数 `f` を持つ関数 `euler_method(x, y, f, dt, w)` を定義しておき、`f` に調和振動子の復元力を与える関数 `harmonic_force` を代入して、時刻 t_k における点 (x_k, y_k) から次の時刻 t_{k+1} の点 $x_{k+1}, y_{k+1})$ を Euler 法で計算します。

初期点 $[x_0, y_0]$ から Euler 法を反復適用して得られる $[x_k, y_k]$（$k = 0, \cdots N-1$）を追加して得られる shape (N,2) の数値軌道配列 `orbit` を得るために、ここでは、まず 0 行 2 列の未初期化配列 `orbit = np.empty((0,2), float)` で用意してから、NumPy 配列 `orbit` に `np.append` によって軌道要素の追加を繰り返します。`orbit` への追加には、shape(1,2) の要素 `[[x0,y0]]` を `axis=0` 方向（行方向）を指定して追加していることに注意してください。

目的の `orbit` は、`orbit` として Python の空リストを用意し、そのリスト append を反復計算し、最後に NumPy 配列化する次の計算によっても得られます。

```
orbit = []
while t < T:
    orbit.append([x0, y0])
    x0, y0 = euler_method(x0, y0, harmonic_force, dt, w)
    t += dt
orbit = np.array(orbit)
```

プログラム 5.4-1 を実行させ、NumPy 配列である軌道配列 `orbit` を 10 行目まで表示すると次のようになります。各行に軌道成分である x, y-要素のリスト `[x0, y0]` が並んでいます。

```
In [1]: %run simple_euler_method.py
In [2]: orbit[:10]
Out[2]:
array([[ 1.        ,  0.        ],
       [ 1.        , -0.0128    ],
       [ 0.999744  , -0.0256    ],
       [ 0.999232  , -0.03839672],
       [ 0.99846407, -0.05118689],
       [ 0.99744033, -0.06396723],
       [ 0.99616098, -0.07673447],
       [ 0.99462629, -0.08948533],
       [ 0.99283659, -0.10221655],
       [ 0.99079226, -0.11492485]])
```

求めた軌道行列の x-成分を `orbit[:,0]`、y-成分を `orbit[:,1]` として `plot` に渡してプロットして図 5.10 を得ます。

5.4.2 NumPy 計算をつかったプログラムの改良

微分を素朴な階差として置き換えた Euler 法では誤差が蓄積してしまいました。Euler 法の改良を考える前に、プログラミング上の課題を対策しておきましょう。

先のプログラム 5.4-1: `simple_euler_diff_method.py` は 2 変数 x, y に関する微分方程式に関するもので、Euler 法にしたがう関数 `euler_method(x, y, f, dt, k)` が `x,y` から次の点 `x1, y1` を返してます。ここで課題とすべきことは、軌道行列 `orbit` を計算するプログラムの中心部が変数の数（今の場合、変数 x と y の 2 つ）に依存している

ことです。

変数の数に依らずベクトル表記の式 (5.28) をそのまま使って、微分方程式を定義するベクトル場関数 $\boldsymbol{V}(\boldsymbol{x},t)$ を関数引数として Euler 法で次の時間の軌道要素を与える関数 `euler_method()` が定義できるとプログラムの汎用性が高まります。

ベクトル場関数の引数を座標成分ごとには与えずに 1 次元 NumPy 配列 `x` を与え、その `x[0],x[1],x[2]...` が $x,y,z,...$-成分であるとみなして関数の返り値が NumPy 配列であるようにプログラムします。Euler 法を使って次の点を返す関数 `euler_method` の引数として、現在の点を表す引数に 1 次元 NumPy 配列 `x`、ベクトル場の関数引数を `vecf` であたえるようにすると、Euler 法の定義式 (5.28) をそのまま使って次の点を

```
x1 = x + dt * vecf(x, t, k)
```

と計算すことができます。

プログラム 5.4-2: `euler_diff_method.py` は、このように書き直した微分方程式の次元に依らない汎用的な Euler 法による数値解法のプログラムです。ここでは調和振動子のベクトル場を表す関数は `vector_filed` として定義しています。

コード 5.4-2　euler_diff_method.py

```
import numpy as np
import matplotlib.pyplot as plt
from typing import Callable

def vector_filed(x: np.ndarray, t, w: float) -> np.ndarray:
    vx = x[1]
    vy = -w ** 2 * x[0]
    return(np.array([vx, vy]))

def euler_method(x: np.ndarray, t: float, vecf: Callable[[np.ndarray,float,
    float],np.ndarray], dt, k: float) -> np.ndarray:
    x1 = x + dt * vecf(x, t, k)
    return(np.array(x1))

k = 0.8 # ばね定数 k = w **2
x0 = np.array([1.0, 0.0]) # 初期値
dt = 0.02 # 時間刻み
t = 0.0   # 経過時間
T = 50    # 時間幅
orbit =np.empty((0,2), float)
```

```
while t < T:
    orbit = np.append(orbit, [x0], axis = 0)
    x0 = euler_method(x0, t, vector_field, dt, k)
    t += dt

fig = plt.figure()
ax = plt.axes()
ax.plot(orbit[:,0],orbit[:,1])
ax.set(xlabel='x', ylabel='y')
#fig.savefig('euler_diff_method.png')
fig.show()
```

Euler 方による微分方程式の軌道行列を計算するには、ベクトル場を定める関数 `vecf` を差し替ええるだけでプログラム全体を書き換える必要がなくなります。Euler 法を改良して軌道要素の制度を向上させるプログラムでも、同様の手法を採用することにします。

Euler 法の数値軌道は正確でないことが、調和振動子 (5.20) の場合に明らかになりました。この調和振動子は質量 $m = 1$ バネ定数 $k = \omega^2$ を持つ力学運動を表しており、その全エネルギー（運動エネルギー $\frac{y^2}{2m}$ と調和バネの位置エネルギー $\frac{\omega^2 x^2}{2}$ の和）

$$E = \frac{y^2}{2} + \frac{w^2 x^2}{2} \tag{5.29}$$

は運動の恒量として時間変化しません。実際、E を時間微分してみると式 (5.20) を使って $\frac{dE}{dt} = 0$ であることがわかります。

この事実を使って Euler 法の計算結果を再度確認してみます。プログラム 5.4-3 は Euler 法によって計算した解軌道とそのエネルギー変化を図**??**のようにプロットします。エネルギー変化をプロットする際に、x, y-軌道要素の 1 次元 NumPy 配列を引数に持つエネルギー関数 `energy` を使って、時間列 `times`（shape (N,) を持つ）と各時間ごとのエネルギー値を一気に描くために `plot(times, energy(orbit.T, w))` と `energy` に `orbit.T` と転置（shape (2, N) を持つ）してエネルギー値列を求めていることに注意してください。

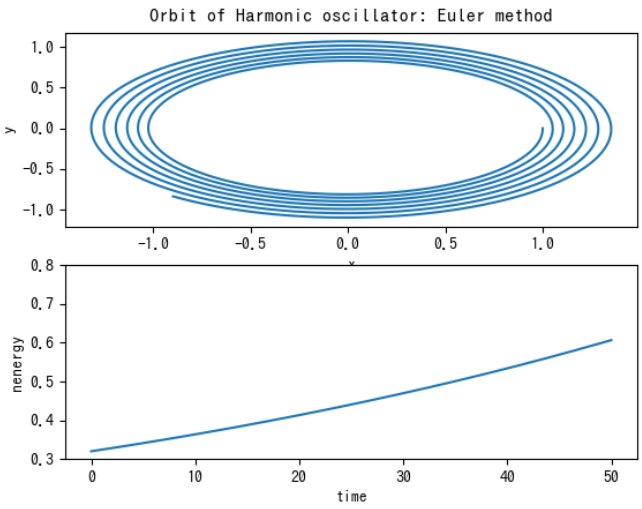


図 5.11　Euler 法を使って調和振動子の微分方程式 (5.20) の数値軌道とそのエネルギー変化。$k = 0.8$、時間刻み幅 $h = 0.02$ で時間区間 $[0, 50]$ までの軌道を初期条件 $x_0 = 1, y_0 = 0$ から計算。本来保存される調和振動子のエネルギーが徐々に増加している。

コード 5.4-3　harmonic_energy_euler.py

```
import numpy as np
import matplotlib.pyplot as plt
from typing import Callable

def vector_field(x: np.ndarray, t, w: float) -> np.ndarray:
    vx = x[1]
    vy = -w ** 2 * x[0]
    return(np.array([vx, vy]))

def euler_method(x: np.ndarray, t: float, vecf: Callable[[np.ndarray,float,
    float],np.ndarray], dt, k: float) -> np.ndarray:
    x1 = x + dt * vecf(x, t, k)
    return(np.array(x1))

def energy(x: np.ndarray, w: float) -> np.float_:
    return(w**2 * x[0]**2 / 2 + x[1]**2 / 2)

w = 0.8
x0 = np.array([1.0, 0.0]) # 初期値
dt = 0.02 # 時間刻み
```

```
t = 0.0   # 経過時間
T = 50    # 時間幅
times = np.arange(t, T, dt)

orbit =np.empty((0,2), float)
while t < T:
    orbit = np.append(orbit, [x0], axis = 0)
    x0 = euler_method(x0, t, vector_field, dt, w)
    t += dt

fig, ax = plt.subplots(2)
ax[0].set(xlabel='x', ylabel='y',
       title='Orbit of Harmonic oscillator: Euler method')
ax[0].plot(orbit[:,0],orbit[:,1]) # 数値軌道
ax[1].set(xlabel='time', ylabel='nenergy', ylim=(0.3, 0.8))
ax[1].plot(times, energy(orbit.T, w)) # エネルギー変化
#fig.savefig('euler_harmonic_energy.png')
fig.show()
```

5.4.3 修正 Euler 法

Euler 法による微分方程式の数値解は、簡単な例について調べてみても実用的な計算精度に問題があることがわかりました。

Euler 法の定義式 (5.28) からわかるように、誤差が時間刻み幅 h に比例することから ($\mathcal{O}(h)$ と記されます)、精度向上のために時間刻みを細かくすることが考えられます (2 桁正しい値を求めるためには 2 桁小さい時間刻みが必要)。しかしながら、こうして対処法では不十分で、計算量の増加をもたらすだけでなく丸め誤差の集積という別種の課題が生じることが知られています。

Euler 法の改良についてさまざまに考案されていますが、誤差が $\mathcal{O}(h^2)$ である次のような**修正 Euler 法**があります。現在時間 t_k におけるベクトル場の値 $\boldsymbol{V}(\boldsymbol{x}_k, t_k)$ と次の時間 t_{k+1} での予測位置 $\tilde{\boldsymbol{x}}_{k+1}$ におけるベクトル場の値 $\boldsymbol{V}(\tilde{\boldsymbol{x}}_{k+1}, t_{k+1})$ との平均を使って、次の時間 t_{k+1} での位置 $\boldsymbol{x}_{k+1}$ を逐次的に決めていくという考え方です。

$$\begin{aligned}
\tilde{\boldsymbol{x}}_{k+1} &= \boldsymbol{x}_k + h \cdot \boldsymbol{V}(\boldsymbol{x}_k, t_k), \\
\boldsymbol{x}_{k+1} &= \boldsymbol{x}_k + h \cdot \frac{\boldsymbol{V}(\boldsymbol{x}_k, t_k) + \boldsymbol{V}(\tilde{\boldsymbol{x}}_{k+1}, t_{k+1})}{2}, \qquad k = 0, 1, 2, \ldots
\end{aligned} \tag{5.30}$$

次の関数 `modified_euler_method` は、先のプログラム 5.4-2 に習って、修正 Euler 法

(5.30) にしたがって 1 次元 NumPy 配列 x、ベクトル場の関数 vecf を引数として、次の軌道点を与える関数です。

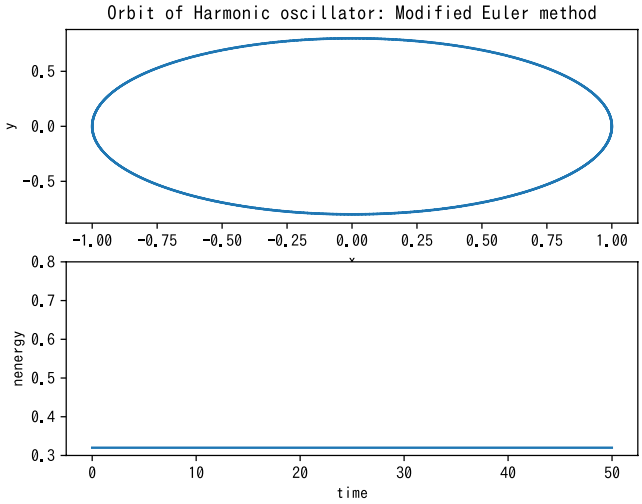


図 5.12　修正 Euler 法を使って調和振動子の微分方程式 (5.20) の数値軌道とそのエネルギーした変化（Euler 法での図 5.11 参照）。

図 5.12 は、修正 Euler 法関数 modified_euler_method を使って、プログラム 5.4-2 と同様に数値軌道とエネルギー変化をプロットしたものです。Euler 法に比べると格段に良好な数値軌道が得られました。

5.4.4　単振子の運動

質量を無視できる棒の一端を固定し他端に質点をつけて鉛直平面内で運動する単振子の微分方程式の数値解を考えてみましょう。x は棒の鉛直下方からの角度、パラメータ k は棒の長さ ℓ と重力加速度 g から決まる質点に依らない定数 $k = g/\ell$ です。

$$\begin{cases} \dfrac{dx}{dt} = y \\ \dfrac{dy}{dt} = -k \sin x \end{cases} \tag{5.31}$$

単振子のベクトル場は $\sin x$ のために非線形ですが楕円積分を使って求積でき、単振子では一般に振動周期は振れ角に依存することがわかっています。式 (5.31) において振れ角が $x = 0$ 付近の微小振動であるとして、$\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots$ において x^3 以上の項を省略した場合、周期運動する調和振動子（$k = \omega^2$）の式 (5.18) となります。

式 (5.31) の第 2 式の両辺に dx/dt をかけて積分すると

$$\left(\frac{dx}{dt}\right)^2 = 2E + 2k\cos x$$

が得られ、全エネルギー $E = \frac{1}{2}y^2 + k(1-\cos x)$ を見出すことができます（位置エネルギーを $U(x) = k(1-\cos x)$ と $x = 0$ のとき $U = 0$ に選んであります)。$E < 2k$ ならば、$\cos\alpha = 1 - E/k$ として $x \in [-\alpha, \alpha]$ の範囲で往復して振子は振動します。$E > 2k$ のときは、x は一方向に増加することになる振子は回転します。全エネルギーが $E = 2k$ のときは、振れ角 $x = \pm\pi$ の鉛直上方で静止した状態です（不安定)。

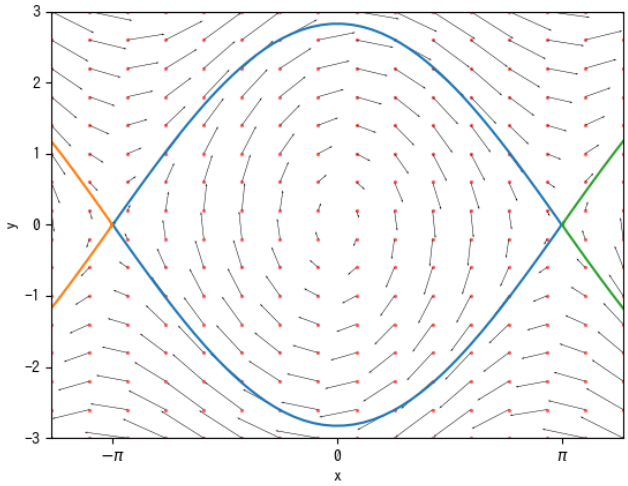


図 5.13　単振動子 (5.31) のベクトル場（$k = 2$)。ベクトル場の不動点として安定な $(0 + 2m\pi, 0)$ 以外に不安定な $(\pm m\pi, 0)$（$m = 1, 2\ldots$）がある。このとき不安定不動点とを結ぶセパラトリックス（分離曲線）は往復運動と回転運動の境目として相空間を分かつ。セパラトリックスの外側から出発した軌道は振子の回転していることを表す。

図**??**は節 5.3.1 で紹介した方法で、単振子のベクトル場 (5.31) の様子をプロットしたものです。ベクトル場の不動点（$\boldsymbol{V}(\boldsymbol{x}) = \boldsymbol{0}$ となる点）は鉛直下方の位置 $(0 + 2m\pi, 0)$（安定な楕円形不動点）以外に鉛直上方の位置 $(-\pi + 2m\pi, 0)$（不安定な双曲不動点）があります。不安定な不動点 $(-\pi + 2m\pi, 0)$ と $(\pi + 2m\pi, 0)$ とは**セパラトリックス**（separatorix: 分離閉曲線）と呼ばれるベクトル場に沿った曲線で結ばれており、往復運動と回転運動の境目をなしています（図 5.13)。

安定な不動点 $(0 + 2m\pi, 0)$ を囲むセパラトリックス内を出発する軌道はベクトル場に沿った閉曲線となって振子が往復振動することを表します（図 5.5 参照)。一方、セパラトリックスの外側から出発した軌道は振子が回転しているすることを表します（軌道がセ

パラトリックスを横断することはありません)。x-座標は回転角であることを考えると、単振子の相空間は $[-\pi, \pi) \times \mathbb{R}$ の円筒（直線 (π, y) と直線 $(-\pi, y)$ を同一視して得られる 2 次元空間）と見るのが自然です。

図 5.14 は、式 (5.31) においてセパラトリックス内の境界の近くにとった初期条件（振れ角を鉛直方向から $x_0 = \pi - 0.01$ とわずかに垂直に足りない角度で静止した（$y_0 = 0$）状態）からの数値軌道を修正 Euler 法で計算した結果です（時間区間は長く先の 4 倍の $T = 200$ に設定しました)。このとき、本来の軌道は閉曲線となってセパラトリック内にとどまり続けます。図のように振れ角 x が区間 $[-\boldsymbol{p}_0, \pi]$ を越えてしまい回転運動となってしたのは、修正 Euler 法で生ずる数値誤差が原因だと考えられます。差分時間区間が長くなると、修正 Euler 法に起因する数値誤差は無視できないくらいに累積してしまうことがわかりました。

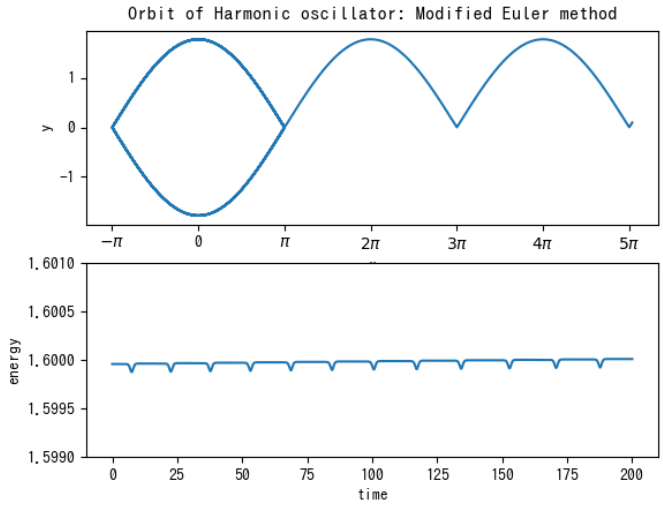


図 5.14　単振子 (5.31)（$w = 0.8$）を修正 Euler 法を使って、セパラトリックス内の境界近くにとった初期条件 $x_0 = \pi - 0.01, y_0 = 0$ からの数値軌道とエネルギー変化（時間刻み幅 $h = 0.02$ で時間区間 $[0, 200]$)。振れ角 x は区間 $[-\pi, \pi]$ を超えて回転運動をしているかのように計算されているが、この初期条件では実際には起こり得ず、この現象は計算誤差によって生じたことがわかる。

5.4.5　Runge-Kutta 法

微分方程式の数値近似解を効率よく求める方法はさまざまな改良が続けられています。ベクトル場 $\boldsymbol{V}(\boldsymbol{x}, t)$ で定まる微分方程式において、現在時間 t_n における値 $\boldsymbol{x}_n$ から次の時間 t_{n+1} での値 $\boldsymbol{x}_{n+1}$ を多段階を経て求める方法として、次の古典的 **Runge-Kutta 法**があり

ます。

$$
\begin{aligned}
\boldsymbol{k}_1 &= \boldsymbol{V}(\boldsymbol{x}_n, t_n) \\
\boldsymbol{k}_2 &= \boldsymbol{V}(\boldsymbol{x}_n + \frac{h}{2}\boldsymbol{k}_1, t_n + \frac{h}{2}) \\
\boldsymbol{k}_3 &= \boldsymbol{V}(\boldsymbol{x}_n + \frac{h}{2}\boldsymbol{k}_2, t_n + \frac{h}{2}) \\
\boldsymbol{k}_4 &= \boldsymbol{V}(\boldsymbol{x}_n + h\boldsymbol{k}_3, t_n + h) \\
\boldsymbol{x}_{n+1} &= \boldsymbol{x}_n + \frac{h}{6}\Big(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_3\Big)
\end{aligned}
\tag{5.32}
$$

微分方程式の数値解法における収束性や安定性に関する詳しい議論は [11] に詳しく説明されています。

次の関数 5.4-4`runge\_kutta` は、Runge-Kutta 法を使って 1 次元 NumPy 配列 `x` とベクトル場を表す関数引数 `vecf`（そのパタメータ引数は `param`）から次の位置を返す関数です。式 (5.32) に従って、途中計算した 4 点 `k1,k2,k3,k4` を使って平均化した点 `x1` ををNumPy 配列として返します。

コード 5.4-4　関数 runge_kutta

```
def runge_kutta(x: np.ndarray, t: float, vecf: Callable[[np.ndarray,float,
    float],np.ndarray], dt, w: float) -> np.ndarray:
    k1 = dt * vecf(x, t, param)
    k2 = dt * vecf(x + k1/2, t + dt/2, param)
    k3 = dt * vecf(x + k2/2, t + dt/2, param)
    k4 = dt * vecf(x + k3, t + dt, param)
    x1 = x + (k1 + 2 * k2 + 2 * k3 + k4) / 6
    return(np.array(x1))
```

図 5.15 は、単振子を古典的 Runge-Kutta 法 (5.32) を使って、セパラトリックス内の境界近くにとった初期条件（$(x_0, y_0) = (\pi - 0.01, 0)$）からの数値軌道とエネルギー変化を示しています。節 5.4.4 の予備的考察の通りに、軌道はセパラトリックス内で閉曲線を描いており、エネルギー変化も目立つような変化がなく Runge-Kutta 法は修正 Euler 方よりも良好な数値軌道を与えます。

しかしながら、挙動が未知の微分方程式において Runge-Kutta 法で計算した数値軌道が長時間に渡って正しい解曲線に近い軌道を与える保証はありません。それでも特異なベクトル場（たとえば、万有引力のように二点間の距離が近づくにつれて力が無限大になる場合）においては格別は取り扱いが必要になるとしても、実用的な要請に応じて解軌道の性質を調べるために Runge-Kutta 法（あるいはそれ以上に注意深く設計された積分法）の利用は十分役に立つと考えられます。

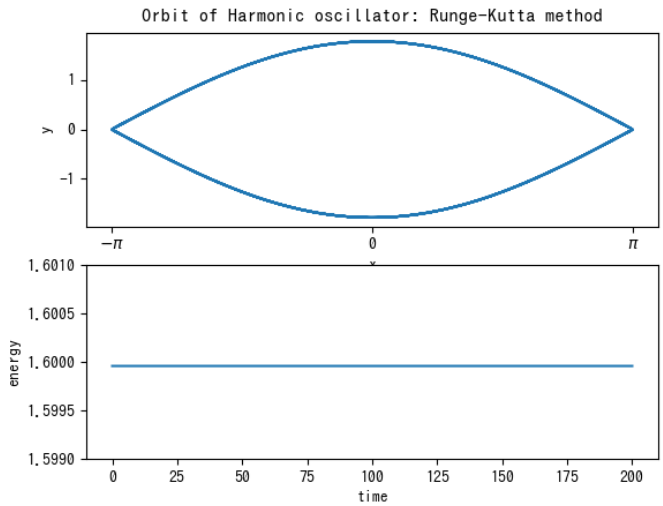


図 5.15　単振子 (5.31)（$w = 0.8$）を古典的 Runge-Kutta 法 (5.32) を使って、セパラトリックス内の境界近くにとった初期条件（$(x_0, y_0) = (\pi - 0.01, 0)$）からの数値軌道とエネルギー変化（時間刻み幅 $h = 0.02$ で時間区間 $T = [0, 200]$）。修正 Euler 法を使った結果とは異なり、この差分時間間隔 T では数値軌道は閉曲線を描いている。

5.4.6　SciPy の数値積分モジュールを使う

ライブラリ SciPy の積分サブパッケージ scipy.integrate には、微分方程式 $dx/dt = f(x,t)$ の初期値問題の数値解を計算するために LSODE(Livermore Solver for Ordinary Differential Equations) [1] として広く知られているルーティンを使った関数 scipy.integrate.odeint が用意されています

微分方程式の数値解法を提供する汎用的なインターフェース scipy.integrate.ode を使って、されたさまざまな数値解法を指定することもできます。時間変化に対して軌道変化が緩やかな箇所と非常に急峻な箇所とが混在する解軌道を持つ硬い (stiff) 微分方程式では、刻み幅 Δt を一定値とせずを急峻な箇所で刻み幅を極めて小さく調整するような計算アルゴリズムを採用する必要があります。odeint の元になっている計算アルゴリズムは Runge-Kutta 系ではなく、Adams-Moulton 法と BDF(backward differentiation formula) 法とを使い分けるように工夫されています（このためには、ベクトル場の Jacobi 行列も与える必要があります)。目的とする数値解法を指定したいときは scipy.integrate.ode を使いますが、格別の問題がない限り、ここで取り上げる odeint を使った数値計算は良好な結果を与えます。詳しい使い方は SciPy リファレンス [3] を参照してください。

`scipy.integrate.odeint` は次のように使います。

1）　Serial Fortran Solvers for ODE Initial Value Problems, https://computing.llnl.gov/casc/odepack/

```
orbit = odeint(vector_filed, x0, times, args)
```

または

```
orbit, info = odeint(vector_filed, x0, times, args, full_output = True)
```

引数の `vector_filed` は要素をリストまたは 1 次元 NumPy 配列として返すベクトル場を与える関数 $\boldsymbol{V}(\boldsymbol{x},t)$（`args` はパラメータの並びを与えるタプル）、`x0` は初期値の並びからなるリストまたは 1 次元 NumPy 配列、`times` は数値軌道を与える時間点列で、この時間点列の先頭時間 t_0 での位置を初期値 $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ となることを想定しています。この時間列の間隔は必ずしも数値解の精度を決めるわけではなく、数値軌道をどの程度の時間間隔で算出するかを決めています。

返り値 `orbit` は軌道行列で、初期値 $\boldsymbol{x}_0$ の要素並びを 1 行目とする軌道点列行列で時間点列の個数分の行数を持つ NumPy 配列です（ベクトル場の成分数を n、`times` の要素数を m とすると、`orbit` は shape (m, n) を有する $m \times n$ 行列です）。後者は、オプション `full_output = True` を与えて、2 つ目の返り値 `info` として計算手段の詳細をディクショナリの形式で確認するときに利用します。

次のコードは単振子 (5.31)（$w = 0.8$）を odeint を時間区間 $[0, 200]$ について差分刻み $dt = 0.02$ で計算した軌道行列 `orbit` を求めるプログラムです。

コード 5.4-5　odeint の利用

```
import numpy as np
from scipy.integrate import odeint
from typing import Callable

def pendlumv(x: np.ndarray, t, w: float) ->np.ndarray:
    vx = x[1]
    vy = -w * np.sin(x[0])
    return(np.array([vx, vy]))

w = 0.8
x0 = [1.0, 0.0] # 初期値
dt = 0.02 # 時間刻み
t0, T = 0.0, 200  # 経過時間
times = np.arange(0, 200, dt)
args = (w,) # ベクトル場パラメータのタプル
orbit = odeint(pendlumv, x0, times, args)
```

図 5.16 は Lorenz 系 (5.23)（パラメータ $\sigma = 10, b = 8/3, r = 28$）の初期値 $(0, 0)$ からの解軌道を odeint を使うプログラム 5.4-6: lorenz.py でプロットしたものです。図 5.9 で表され

るベクトル場からは解軌道がこのようになることは想像できません。

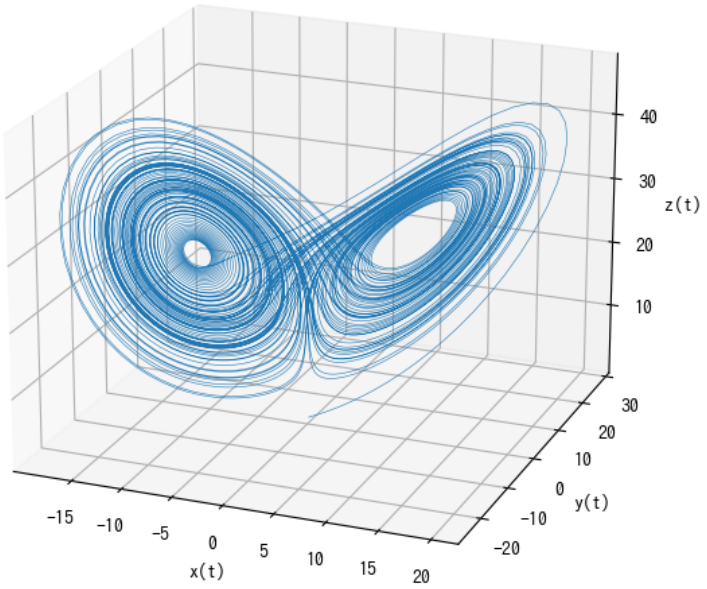


図 5.16　Lorenz の微分方程式 (5.23) で与えられる解軌道。$\sigma = 10, b = 8/3, r = 28$、初期条件 $(0.1, 0.1, 0.1)$.

パラメータ $\sigma = 10, b = 8/3, r = 28$ を固定して初期条件をさまざまに変化させても、一定の推移時間後の軌道の様子はみな図 5.16 と大差ありません。図 5.16 で示される領域（蝶が羽を広げているように見えるためのバラフライと呼んでいます）は 3 次元空間の任意の場所を初期条件として出発する軌道を吸い寄せます。周辺の解軌道を吸い寄せてしまう領域を**アトラクタ** (attractor) といいます。

第**??**章で詳しく調べますが、この図 5.16 のアトラクタは、1 点または周期軌道でもなく非周期的挙動に引き込む**ストレンジ・アトラクタ**と呼んでいます。IPython で描かれるプロットを回転させて観察すると、このアトラクタは 2 次元曲面的な構造を持つように見えますが、垂直方向にカントール集合を呈する複雑な微細構造を持つことがローレンツ方程式の登場 (1963) から 40 年たってようやく数学的な存在証明 [45] がなされました。

Lorenz アトラクタのように、微分方程式が呈するアトラクタの構造は非常に複雑になり得るために、多くの研究者の注意を引きつけ研究が続けられています。

コード 5.4-6　lorenz.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
from scipy.integrate import odeint
```

```
from typing import Callable

# lorenz system
def lorenz(x: np.ndarray, t, p, b, r: float) -> np.ndarray:
    vx = -p * x[0] + p * x[1]
    vy = -x[0] * x[2] + r * x[0] - x[1]
    vz = x[0] * x[1] - b * x[2]
    return(np.array([vx,vy,vz]))
# parameters
p =10
b = 8.0 / 3.0
r = 28

x0 = [0.1, 0.1, 0.1]
t0, T = 0, 100# 時間間隔
dt = 0.01#差分刻
times = np.arange(t0, T, dt)
args = (p, b, r)
orbit = odeint(lorenz, x0, times, args)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(orbit[:, 0], orbit[:, 1], orbit[:, 2], linewidth = 0.4)
ax.set(xlabel='x(t)', ylabel='y(t)', zlabel='z(t)')
#fig.savefig('lorenz.png')
plt.show()
```

第 6 章　非線形微分方程式

n-変数自励微分方程式

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{V}(\boldsymbol{x}) \tag{6.1}$$

において、ベクトル場 $\boldsymbol{V}(\boldsymbol{x})$ が

$$\boldsymbol{V}(\alpha\boldsymbol{x}_1 + \beta\boldsymbol{x}_2) \neq \alpha\boldsymbol{V}(\boldsymbol{x}_1) + \beta\boldsymbol{V}(\boldsymbol{x}_2), \qquad \alpha, \beta \in \mathbb{R}, \boldsymbol{x}_1, \boldsymbol{x}_2 \in \mathbb{R}^n \tag{6.2}$$

であるとき、**非線形ベクトル場**といいます。

非線形ベクトル場を持つ初期値問題の軌道を求積して、その一般解を正確に求めることはたいへん困難です。しかしながら、非線形微分方程式の解の様子を局所的に時間的（ある時間経過の範囲で）および空間的に（ある近傍の範囲で）把握することは可能です。

6.1　変分方程式

式 (6.1) において、いま $\boldsymbol{x}(t)$ をある開区間 J 上で定義された時刻 $t_0 \in J$ で点 $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$ を通過する微分方程式 (6.1) の解だと仮定します。このとき、時刻 t_0 で近くの点 $\boldsymbol{x}_0 + \boldsymbol{\delta}_0$ を通過する解を $\boldsymbol{x}'(t)$ を考えます。

式 (6.1) のベクトル場 $\boldsymbol{V}(\boldsymbol{x})$ の $\boldsymbol{x}$ に関する Jacobi 行列（式 (2.14) 参照）$J_{\boldsymbol{x}}(\boldsymbol{V})$ を使って定まる次の微分方程式を、解 $\boldsymbol{x}(t)$ に沿った**変分方程式** (variational equation) といいます。

$$\frac{d\boldsymbol{\delta}(t)}{dt} = J_{\boldsymbol{x}}(\boldsymbol{V})\boldsymbol{\delta}(t), \qquad \boldsymbol{\delta}(t_0) = \boldsymbol{\delta}_0. \tag{6.3}$$

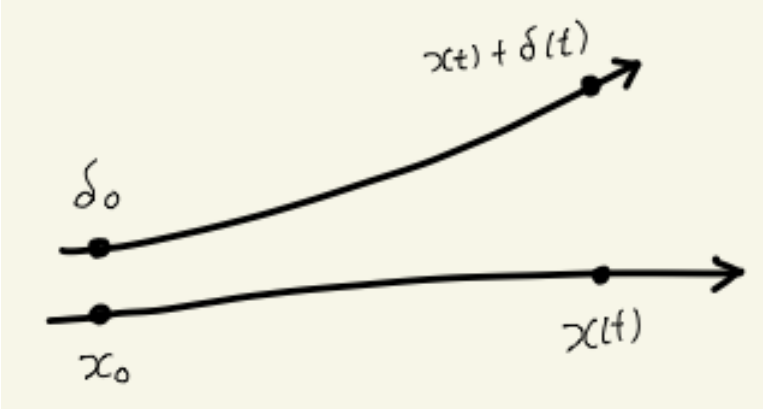


図 6.1　式 (6.1) を満たす $\boldsymbol{x}(t)$ と $\boldsymbol{x}'(t)$ は、時刻 t_0 でそれぞれ点 $\boldsymbol{x}(t_0)=\boldsymbol{x}_0$ と $\boldsymbol{x}'(t_0)=\boldsymbol{x}_0+\boldsymbol{\delta}_0$ を通過する。解 $\boldsymbol{x}'(t)$ は、$\boldsymbol{\delta}(t_0)=\boldsymbol{\delta}_0$ であるような時刻 $\boldsymbol{x}(t)$ に沿った変分方程式 (6.3) の解 $\boldsymbol{\delta}(t)$ を使って $\boldsymbol{x}(t)+\boldsymbol{\delta}(t)$ で近似できる。

変分方程式 (6.3) は、$\boldsymbol{\delta}_0$ が小さいとき、流れ

$$t \mapsto \boldsymbol{x}(t)+\boldsymbol{\delta}(t)$$

は初期値 $\boldsymbol{x}_0+\boldsymbol{\delta}_0$ を持つ式 (6.1) の解のよい近似を与えるという意味があります。

定理 6.1　任意の $\varepsilon>0$ に対してある $\delta>0$ が取れて、$\boldsymbol{x}_0$ からのずれ $\boldsymbol{\delta}_0$ を $|\boldsymbol{\delta}_0| \leqq \delta$ と十分に小さくするれば、変分方程式 (6.3) の解を使った曲線 $\boldsymbol{x}(t)+\boldsymbol{\delta}(t)$ は時間区間 J で $\boldsymbol{x}'(t)$ をよく近似する。

$$\Big|\boldsymbol{x}'(t)-\big(\boldsymbol{x}(t)+\boldsymbol{\delta}(t)\big)\Big| \leqq \varepsilon|\delta_0|, \qquad t\in J$$

■

この結果は、点 $\boldsymbol{V}(\boldsymbol{x}+\boldsymbol{\delta})$ でのベクトル場が本来与える微分方程式を考えると理解できるでしょう。

$$\begin{aligned}\frac{d}{dt}(\boldsymbol{x}+\boldsymbol{\delta}) &= \boldsymbol{V}(\boldsymbol{x}+\boldsymbol{\delta}), \qquad \boldsymbol{x}(t_0)+\boldsymbol{\delta}(t_0)=\boldsymbol{x}_0+\boldsymbol{\delta}_0\\ &\simeq \boldsymbol{V}(\boldsymbol{x})+J(\boldsymbol{V})\boldsymbol{\delta}\end{aligned}$$

これより、$\boldsymbol{x}(t)$ の周りの解 $\boldsymbol{x}'(t)$ は局所的には、時間間隔 J において変分方程式の解 $\boldsymbol{\delta}(t)$ をつかって近似的に $\boldsymbol{X}(t)+\boldsymbol{\delta}(t)$ と表されることがわかります。

6.2　ベクトル場の線形化

式 (6.1) のベクトル場 $V(\boldsymbol{x})$ がゼロとなる点集合 $\mathcal{F}(V)$

$$\mathcal{F}(V)=\{\boldsymbol{x}^* \,|\, V(\boldsymbol{x}^*)=\boldsymbol{0}\} \tag{6.4}$$

を**不動点** (fixed points) または平衡点 (equilibrium point) といいます（ベクトル場の**特異点** (singular point) ということもあります)。点 $\boldsymbol{x}^* \in \mathcal{F}(V)$ は式 (6.1) の定数解になっていることに注意してください。

変分方程式 (6.3) において、基準解 $\boldsymbol{x}(t)$ がベクトル場 $V(\boldsymbol{x})$ の不動点 $\boldsymbol{x}^* \in \mathcal{F}(V)$ である場合、式 (6.3) を不動点 $\boldsymbol{x}^*$ のまわりの線形化方程式と呼び（節 2.7.7 参照)、改めてつぎのように表します。

$$\frac{d\boldsymbol{x}}{dt} = A\boldsymbol{x}, \quad A(\boldsymbol{x}^*) \equiv J_{\boldsymbol{x}^*}(\boldsymbol{V}) \tag{6.5}$$

A はヴェクトル場 $V(\boldsymbol{x})$ のヤコビ行列 $J_{\boldsymbol{x}}(\boldsymbol{V})$ の不動点 $\boldsymbol{x}^*$ における値で定数正方行列です。

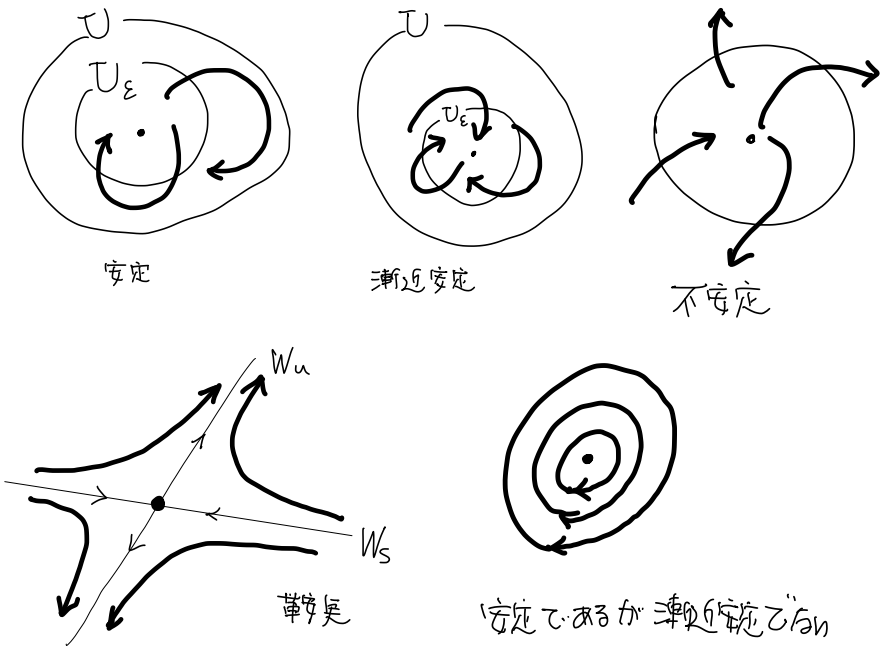


図 6.2　ベクトル場の不動点。

ベクトル場の不動点に注目するのは、不動点の近傍 $\boldsymbol{x} \in U_\varepsilon(\boldsymbol{x}^*)$ から出発する軌道がその後も不動点のそばに留まり続けるかどうかに関心があるためです。不動点の近傍 $U(\boldsymbol{x}^*)$ に対して、$\boldsymbol{x}^*$ の近傍 $U_\varepsilon(\boldsymbol{x}^*) \subset U(\boldsymbol{x}^*)$ をうまく取ると、$U_\varepsilon(\boldsymbol{x}^*)$ 内から出発した軌道はすべての $t \geqq 0$ で $U(\boldsymbol{x}^*)$ 内に留まり続けるときその不動点を**安定** (stable) 、さらに $\lim_{t\to\infty} \boldsymbol{x}(t) = \boldsymbol{x}^*$ をも満たすように $U_\varepsilon(\boldsymbol{x}^*)$ がとれるとき、$\boldsymbol{x}^*$ を**漸近安定** (asympticall stable) といいます。一方、不動点の適当な近傍 $U(\boldsymbol{x}^*)$ があって、それに含まれる $\boldsymbol{x}^*$ の任

意の近傍 $U_\varepsilon(\boldsymbol{x}^*) \subset U(\boldsymbol{x}^*)$ に対して、$\boldsymbol{x}(t_0) \in U_\varepsilon(\boldsymbol{x}^*)$ を出発する解で $U(\boldsymbol{x}^*)$ に留まらない解が少なくとも 1 つは存在する場合、不動点は**不安定** (unstable fixed point) と言います。

定数行列 A から生成される線形な流れ e^{At}（節 5.2.1 の式 (5.12)）において、不動点である原点での安定性は A の固有値だけによって全空間 $\mathbb{R}^n$ にわたって（大域的に）決定され、次元によらず流れの様子は明快に記述することができます [12][14]。しかしながら、非線形ベクトル場では不動点の周りで線形化した式 (6.5) が不動点の周りの流れについて良好な近似を与えるのは不動点近傍のしかも限られた時間の間だけであり（定理 6.1）、全空間にわたる大域的で十分な時間経過後の流れについては何ら情報を与えてはいません。

単振子 (5.31) のベクトル場のヤコビ行列は

$$J_{\boldsymbol{x}} = \begin{bmatrix} 0 & 1 \\ -k\cos x & 0 \end{bmatrix}$$

をベクトル場の不動点 $(0,0)$, $(\pm\pi, 0)$ にごとに求めると、$J_{(0,0)}$ の固有値は複素共役の純虚数 $\pm i\sqrt{k}$ を持り、不動点 $(0,0)$ は楕円形でその近傍で安定である（漸近安定ではない）ことがわかります。実際、$(0,0)$ は鉛直下方の静止点で、その近傍から出発する解曲線は小さな閉曲線を描く往復運動を続けます。$_{(\pm\pi,0)}$ の固有値は実な正負固有値 $\pm\sqrt{k}$ を持ち、不動点 $(\pm\pi, 0)$ は双曲型（鞍点）となって不安定であることがわかります。実際、鉛直上方の静止点 $(\pm\pi, 0)$ は不安定で、そのごく近くから出発する軌道は $(\pm\pi, 0)$ の近傍から離れ去ってしまいます。ただし、円筒 $[-\pi, \pi) \times \mathbb{R}$ を相空間とするとき、往復運動あるいは回転運動によって解軌道はこの不安定不動点近傍に何度でも**回帰** (reccurent) します。

単振子の線形化による局所解析の結果は平面上の方程式という次元の特殊性によって、大域的な性質も決定してしまいます。平面系の微分方程式においては次の有名な定理が知られています [14]。

定理 6.2 (Poincaré-Bendixon)　平面の微分方程式の空でない有界な極限集合が不動点を含まなければ閉軌道である。

単振子の不安定不動点 $\boldsymbol{x}^* = (\pm\pi, 0)$ から伸びているセパラトリックスは、$t \to \infty$ のときに解軌道が $\boldsymbol{x}^*$ に収束する初期集合の集合 $W_s(\boldsymbol{x}^*)$、および $t \to -\infty$ のときに解軌道が $\boldsymbol{x}^*$ に収束する初期集合の集合 $W_u(\boldsymbol{x}^*$ と成っています。一般に不動点の固有値の実部が負に対応して不動点で E_s に接するこのような性質を持つ $W_s(\boldsymbol{x}^*)$ を不動点の**安定多様体** (stable manifold) 、不動点の固有値の実部が正に対応して不動点で E_u に接する $W_u(\boldsymbol{x}^*)$ を**不安定多様体** (unstable manifold) と呼んでいます。

2 次元ベクトル場は、非線形であっても定理 6.2 によって、ベクトル場の不動点とその点におけるベクトル場の線形化による固有値を調べることによって相空間内の流れの様子は定性的に把握することができます。

2 次元以上の非線形なベクトル場、2 次元ベクトル場が時間に依存するような場合や 3 次元以上の相空間における流れは線形解析から全容を掴むことはできません。

6.3 Lorenz 方程式の線形化

いままで何度か登場してきた Lorenz 方程式 (5.23) を改めて以下に書いておきましょう。通常、$\sigma = 10, b = 8/3$ とし、$r > 0$ をパラメータとして扱います。

$$\begin{aligned} \frac{dx}{dt} &= -\sigma x + \sigma y \\ \frac{dy}{dt} &= rx - y - xz \\ \frac{dy}{dt} &= -bz + xy \end{aligned} \tag{6.6}$$

Lorenz 方程式のベクトル場 $\boldsymbol{V}(x, y, z))$ の不動点集合

$$\mathcal{F}(\boldsymbol{V}) = \{(x^*, y^*, z^*) \,|\, \sigma x + \sigma y = 0, rx - y - xz = 0, -bz + xy = 0\} \tag{6.7}$$

を SymPy を使って計算してみましょう。SymPy を読み込んで変数 x, y, z およびパラメータ p, r, b を記号として設定し、ベクトル場の x, y, z-成分からなるリスト `lorenz_vec` を定義します。

```
import sympy as sym
x, y, z = sym.symbols('x y z')
p, r, b = sym.symbols('p r b')
lorenz_vec = [-p * x + p * y, -x * z + r * x - y, x * y - b * z]
```

多次元多変数非線形方程式 `lorenz_vec` $= 0$ を変数リスト `[x ,y, z]` について解く関数 `sympy.nonlinsolve` を使い、集合型のタプルとして 3 つの x, y, z 成分を持つリストを求めます。

```
fix1, fix2, fix3 = sym.nonlinsolve(lorenz_vec, [x ,y, z])
fix1
```

```
(0, 0, 0)
```

```
fix2
\vspace{-4mm}
\begin{lstlisting}[frame=none]
(sqrt(b*r - b), sqrt(b*(r - 1)), r - 1)
```

```
fix3
\vspace{-4mm}
\begin{lstlisting}[frame=none]
(-sqrt(b*r - b), -sqrt(b*(r - 1)), r - 1)
```

これより、Lorenz ベクトル場の不動点として原点 O と C_+ および C_- の 3 つが求まりました。

$$O=(0,0,0),\quad C_+=(\sqrt{b(r-1)},\sqrt{b(r-1)},r-1),\quad C_-=(-\sqrt{b(r-1)},-\sqrt{b(r-1)},r-1) \tag{6.8}$$

次に、Lorenz 系のベクトル場の Jacobi 行列 $J_{\boldsymbol{x}}(\boldsymbol{V})$ を求めて、不動点の周りの線形化微分方程式を計算します。ベクトル場の Jacobi 行列を SymPy で計算するために、ベクトル場の各 x,y,z-成分を要素リストとする 1 行 3 列の SymPy 行列 `lorenz_vecmat` を定義します。

```
lorenz_vecmat = sym.Matrix([[-p * x + p * y], [-x * z + r * x - y],\
                [x * y - b * z]])
```

Lorenz ベクトル場の変数 x,y,z に関する Jacobi 行列は、SymPy の Matrix メソッド `.jacobian((x,y,z))` を使って

```
lorenz_vecmat.jacobian((x,y,z))
```

```
Matrix([
[   -p,  p,  0],
[r - z, -1, -x],
[    y,  x, -b]])
```

と求められますが、変数 x,y,z に不動点 fixedpt $=[x^*,y^*,x^*]$ の値を代入した行列として計算するために次のように関数定義しておきます。

```
def linearized_lorenz(fixedpt):
    jacobi_lorenz = lorenz_vecmat.jacobian((x,y,z))
    sub = jacobi_lorenz.subs([(x, fixedpt[0]), (y, fixedpt[1]),\
           (z, fixedpt[2])])
    return(sub)
```

この関数を使って、先に計算した各不動点 O(`fix1`), C_+(`fix2`) および C_-(`fix3`) における各 Jacobi 行列を次のように求めることができます。

```
linmat1 = linearized_lorenz(fix1)
linmat1
```

```
Matrix([
[-p,  p,  0],
[ r, -1,  0],
[ 0,  0, -b]])
```

```
linmat2 = linearized_lorenz(fix2)
linmat2
```

```
Matrix([
[             -p,              p,               0],
[              1,             -1, -sqrt(b*r - b)],
[sqrt(b*(r - 1)), sqrt(b*r - b),              -b]])
```

```
linmat3 = linearized_lorenz(fix3)
linmat3
```

```
Matrix([
[             -p,              p,               0],
[              1,             -1,  sqrt(b*r - b)],
[-sqrt(b*(r - 1)), -sqrt(b*r - b),              -b]])
```

これより、Lorenz ベクトル場の不動点 O, C_+, C_- のまわりの線形化微分方程式を与える定数行列 L_0, L_{C_+}, L_{C_-} はそれぞれ次のようになることがわかりました。

$$L_0 = \begin{bmatrix} -p & p & 0 \\ r & -1 & 0 \\ 0 & 0 & -b \end{bmatrix}, \tag{6.9}$$

$$L_{C_+} = \begin{bmatrix} -p & p & 0 \\ 1 & -1 & -\sqrt{b(r-1)} \\ \sqrt{b(r-1)} & \sqrt{b(r-1)} & -b \end{bmatrix}, \tag{6.10}$$

$$L_{C_-} = \begin{bmatrix} -p & p & 0 \\ 1 & -1 & \sqrt{b(r-1)} \\ -\sqrt{b(r-1)} & -\sqrt{b(r-1)} & -b \end{bmatrix} \tag{6.11}$$

たとえば、不動点 C_+ における Jacobi 行列 L_{C_+} のパラメータ値 $p = 10, b = 8/3, r = 28$ における固有値は次のように計算できます。

```
vec2 = list(linmat2.eigenvals().keys())
vec2values = sym.Matrix(vec2).subs([(p, 10), (r, 28), (b, 8/3)])
vec2values
```

```
Matrix([
[-13.854577914596],
[0.712100200983433 - 5.26765575653899*sqrt(3)*I + 1.2362891540375/(-1/2 + sqrt
    (3)*I/2)],
[0.712100200983433 + 1.2362891540375/(-1/2 - sqrt(3)*I/2) + 5.26765575653899*
    sqrt(3)*I]])
```

不動点 C_+ の周りの線形化行列 L_{C_+} の固有値 ‘vec2values‘を浮動小数からなる NumPy 配列でとして表すと、次のように 1 つ負の実数と正の実部を持つ 1 組の複素数からなることが確かめられます。

```
import numpy as np

npvec = np.array(vec2values, dtype='complex')
npvec
```

```
array([[-13.85457791 +0.j        ],
       [  0.09395562-10.19450522j],
       [  0.09395562+10.19450522j]])
```

6.4 Lorenz アトラクタの薄い構造

時刻 $t = 0$ で体積 V_0 を持つ領域内の各点が Lorenz ベクトル場 (6.6) にしたがって一斉に流れていくとき、時刻 t での体積 $V(t)$ がどのように変化していくかを調べてみましょう、このために、Lorenz 方程式ベクトル場 $\boldsymbol{V}(x, y, z) = (v_x, v_y, v_z)$（式 (6.6) の右辺を成分とするベクトル値関数）の**発散**（divergence: 湧き出し）を計算してみると次のように負の定数 $-(\sigma + b + 1)$ になることがわかります。

$$\begin{aligned} \mathrm{div}\boldsymbol{V} &= \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \\ &= \frac{\partial}{\partial x}(-\sigma x + \sigma y) + \frac{\partial}{\partial y}(rx - y - xz) + \frac{\partial}{\partial z}(-bz + xy) \\ &= -(\sigma + b + 1) \end{aligned} \tag{6.12}$$

ベクトル場の発散はその点を囲む微小領域面からの湧出する流れを表していることから、式 (6.12) は、Lorenz 系において相空間内の体積変化は次の微分方程式

$$\frac{dV(t)}{dt} = -(\sigma + b + 1)V, \quad V(0) = V_0$$

に従うことになります。これより、Lorenz システムにおける体積変化は $V(t) = V_0 \mathrm{e}^{-(\sigma+b+1)t}$ と指数関数的に急速に縮小し、有限の体積 V_0 は体積ゼロに変形されていくことがわかります。232 ページで説明したように、流れは 1 点に沈み込むのではなくその上で非周期的挙動を呈する Lorenz アトラクタに引き込まれていきます（非周期的であることの意味は節 6.5 で明らかにします）。この事実は引き込み現象を伴う Van der Pole 系 5.21 とは本質的に異なる現象です。Van der Pole 系ではパラメータ $-1 \leqq \mu \leqq 0$ ではすべての流れは原点に漸近しますが、μ が 0 を超えると **Hopg 分岐**によって ω-極限集合である極限周期軌道が出現し、最終的にすべての流れはこの周期解に漸近します。Lorenz 系では非周期運動をするアトラクタに引きまれるのです（式 (6.6) のパラメータ r を変化させた時、アトラクタがどのように変化するかについて膨大な研究が積み上げられてきました）。

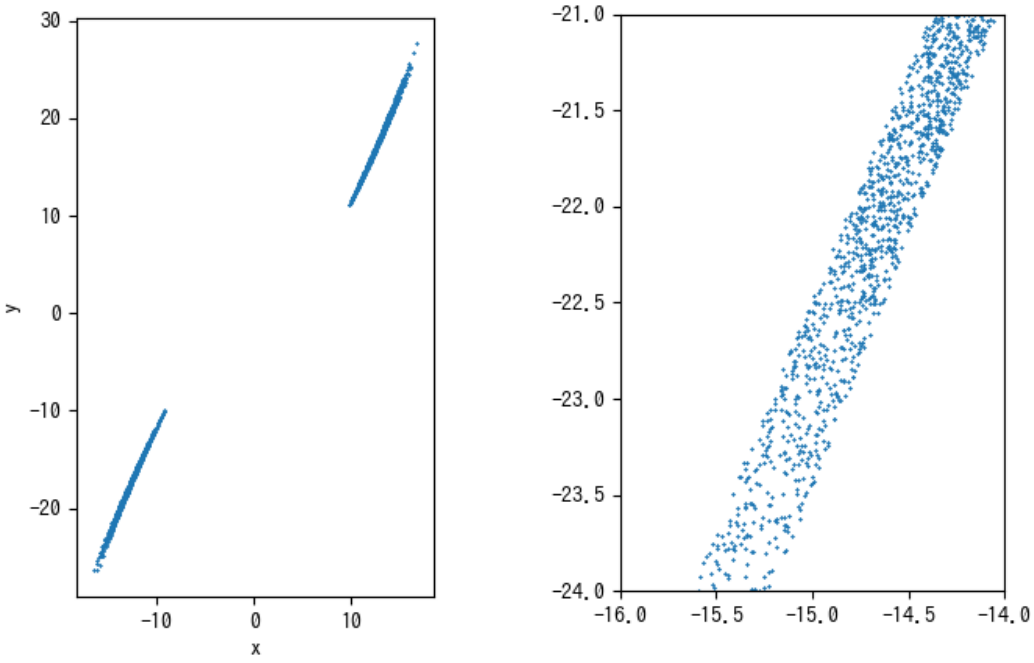


図 6.3　Lorenz の微分方程式 (6.6)（パラメータ $\sigma = 10, b = 8/3, r = 28$）で平面 $z = r - 1$ での Poincaré 断面（`oedint` を刻み幅 $dt = 0.005$ で計算）。(a) アトラクタの全体図 (b) 部分拡大図 (時間区間 $T = 12000$). 初期値 $(0.1, 0.1, 0.1)$ からの軌道が平面 $z = r - 1$ を $dz/dt > 0$ で横切る点列 (x_i, y_i) を初期の推移的軌道を取り除いてプロット。

Lorenz 系において体積が時間とともに指数的に縮小し急速に体積ゼロに漸近する事実

とをあわせて考えると、“**Lorenz アトラクタは体積をもたない複雑に折りたたまれた薄っぺらな構造**” を持つと理解せざるを得ません。この非周期的挙動を呈するアトラクタが 2 次元曲面をなすことはありえません。アトラクタ上で軌道が何度も交差しているように観測される様子は、多項式で定まった Lorenz ベクトル場の滑らかさが保証している微分方程式の解の一意性と連続性と両立しないからです。

このことを詳しく確かめてみるために、プログラム 6.4-1: `lorenz_section.py` を使って Lorenz 系の軌道が 3 次元空間内の平面 $z = r-1$（不動点 $C_\pm$ の z-座標）を $dz/dt > 0$ の方向に横切る点列 $\{(x_i, y_i)\}$ をプロットしてみると図 6.3 が得られます。初期条件をさまざまに変化させた軌道であっても、一定の推移的時間後に平面 $z = r-1$（ちょうど 2 つ組になった不動点 $C_\pm$ の z-座標）を横切る点列 $\{(x_i, y_i)\}$ は図 6.3 と区別がつかず、アトラクタに引き込まれていることを確認できます。IPython の環境で matplotlib で描画した図 5.16（232 ページ）をマウスを使ってさまざまな方向から観察しても（平面 $z =$ 一定 を変えて軌道の切断を観察しても）アトラクはどこも「薄い」ことがわかります。ただし、図 6.3(b) が示すように完全な 2 次元曲面とはなっておらず、垂直方向には微細構造があることが知られており折りたたまれたアトラクタ形状を反映しています。

こうした Lorenz の発見に基づき Hénon は 2 次元平面上でこのような折りたたまれたアトラクタを持つ平面上の写像を提案し [37]、その後の詳細なアトラクタ構造の研究の先鞭をつけました。

コード 6.4-1　lorenz_section.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from typing import List

def lorenz(x: np.ndarray, t, p, b, r: float) -> np.ndarray:
    vx = -p * x[0] + p * x[1]
    vy = -x[0] * x[2] + r * x[0] - x[1]
    vz = x[0] * x[1] - b * x[2]
    return(np.array([vx,vy,vz]))
# parameters
p =10
b = 8.0 / 3.0
r = 28
# initial points
x0 = [0.1, 0.1, 0.1]
# setting
```

```
t0, T = 0, 300# 時間間隔
dt = 0.01#時間間隔
times = np.arange(t0, T, dt)
args = (p, b, r)
orbit = odeint(lorenz, x0, times, args)

# surface of section # 数列 seqがvalue値を正方向に横断するインデックス
def section_id(seq: np.ndarray, value: float) -> List[int]:
    crossed = [(seq[k] - value) < 0 and (seq[k+1] - value) > 0 for k in range(
        len(seq)-1)]
    crossed.append(False)# 末尾は常にFalse
    return(crossed)

transitstep = 50 # 推移時間
asymorbit = orbit[transitstep:]
zorbit = asymorbit[:,2]
sections = asymorbit[section_id(zorbit, r-1)]# 平面z=r-1を正方向に横切る点列


fig = plt.figure()
ax = plt.axes()
ax.set_aspect('equal')
ax.set(xlabel = 'x', ylabel = 'y')
ax.plot(sections[:,0], sections[:,1], "x", markersize=1)
#fig.savefig('lorenz_section.png')
plt.show()
```

6.5 Lorenz 系の非周期性

Lorenz 系（$r = 28$）の流れが非周期的であることを数値軌道から説明する方法として Poincaré の切断面の方法と Lorenz プロットによる方法を紹介します。

6.5.1 ポアンカレ写像

プログラム 6.4-1 で計算した `sections` には平面 $z = r-1$ を横切る軌道点列 $\{(x_i, y_i, z_i)\}$ ($z_i \approx r-1$)が格納されています。図6.4 は、その y-成分を `ysection = sections[:, 1]` で取り出した点列 $\{y_i\}$ について、平面上に点列 (y_i, y_{i+1}) をプロットしたものです。その

プログラムはプログラム 6.4-1 のプロット部分を修正した 6.5-1: `lorenz_first_return` です。

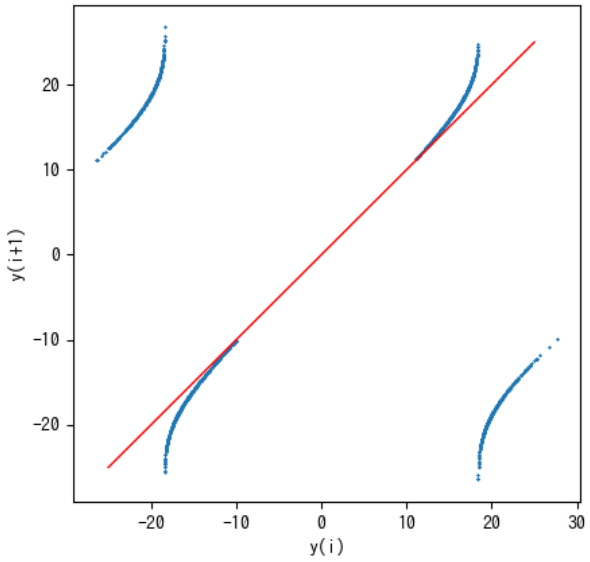


図 6.4　Lorenz の微分方程式 (6.6)（パラメータ $\sigma = 10, b = 8/3, r = 28$）が平面 $z = r - 1$ を $dz/dt > 0$ で横切る軌道の y-成分列 $\{y_i\}$ から (y_i, y_{i+1}) をプロット。点列 $\{y_i\}$ は区間 I 上の 1 次元写像 $f : I \to I$ に従っているように見える。写像 $f(y)$ の傾きが 1 以上 $df/dy > 1$ であることに注意。

図 6.4 は、$r = 28$ のとき、平面 $z = r - 1$ を $dz/dt > 0$ の向きで横断する流れの y-成分について、次の横断点の y-成分を与える写像 $y_i \mapsto y_{i+1}$ のグラフ、つまり 1 次元区間 $I = [-30, 30]$ 上の写像 $f : I \to I$ があって、点列 $\{y_i\}$ がこの写像 f によって、$y_{i+1} = f(y_i)$ というように定まっているように見ることができます。このとき、区間 I の 2 つの連続部分区間 $I_1, I_2 \subset I, I_1 \cap I_2 = \phi$ があって、$I_2 = f(I_1), I_1 = f(I_2)$ であって、さらに各部分区間 I_j 上で写像 f のグラフは傾き $df/dy > 1$ となっていることも観察できます。

このことは、（そのような写像 f が都合よく存在すると仮定するならば）写像 $F = f^2$ を考えると、各部分区間 $I_j (j = 1, 2)$ 上の区分的な連続写像 $F : I_j \to I_j$ は I_j 上で $dF/dy > 1$ であることを強く示唆します。その場合、Lebesgue 測度に絶対連続な不変測度 μ が存在して

$$\mu(F^{-1}A) = \mu(A), \qquad A \subset I_j$$

が成立することが知られています。言い換えれば、節 3.5 で紹介したように、写像 F による $y_0 \in I_j$ の軌道 $y_{i+1} = F(y_i)$ は区間 I_j を非周期的に動き回ることを説明してくれます。このようにして数学的な厳密さはありませんが、Lorenz 系が非周期的挙動を呈しているこ

とは、数値計算による現象論的な Poinca'e の切断面に関する First Returm 写像を通じて理解することができます。

ベクトル場の流れが周期的であるとき、その流れを切断する平面（余次元 1）S をとることができ、周期軌道の近傍にある S 上の点について写像 $P : S \to S$ を考えることができます。これを **Poincaré 写像**と呼びます。図 6.4 は Lorenz 系 $r = 28$ における平面 $z = r - 1$ 上のポアンカレ写像のグラフを表しています。

微分方程式によって記述される力学系の研究において 1 次元写像や 2 次元写像が重要であるのは、連続な流れがもたらす特徴をあたかも流れに埋め込まれた写像があるかのように、写像の性質から流れの様子を良く説明できることがあるからです。

なお、微分方程式系においてポアンカレ写像を精密に計算するための巧妙なアルゴリズムが Hénon[38] によって考案されています [46]。

コード 6.5-1　lorenz_first_return.py

```
.....
sections = asymorbit[section_id(zorbit, r-1)]
ysection = sections[:, 1]
ylist = np.delete(ysection,-1)# 末尾を除去
nylist = np.delete(ysection, 0)# 先頭を除去

fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.set(xlabel = 'y(i)', ylabel = 'y(i+1)')
ax.plot(ylist, nylist, "x", markersize=1)
ax.plot([-25,25], [-25,25], 'r', linewidth=1)
#fig.savefig('lorenz_first_return.png')
plt.show()
```

6.5.2　Lorenz プロット

Lorenz(1963) は、Poincaré 写像とは別の方法によって、軌道が呈する非周期性を示唆することに成功したました。

Lorenz 系の解軌道 $\boldsymbol{x}(t) = (x(t), y(t), z(t))$ の 1 つの成分、たとえば $z(t)$ に注目したときの時間変化は図 6.5(a) のようになります。ある時刻 t_c で $z(t_c)$ が極大、つまり $t' < t_c$ および $t_c < t''$ に対して $z(t') < z(t_c), z(t_c) > z(t'')$ となるとき $z(t_c)$ は極大値を取るといいます。Lorenz[39] は時間変化に関する極大値列 $\{z_i\}$ を計算し、$\{z_i, z_{i+1}\}$ をプロットしました。このプロット法を **Lorenz プロット**と呼びます。

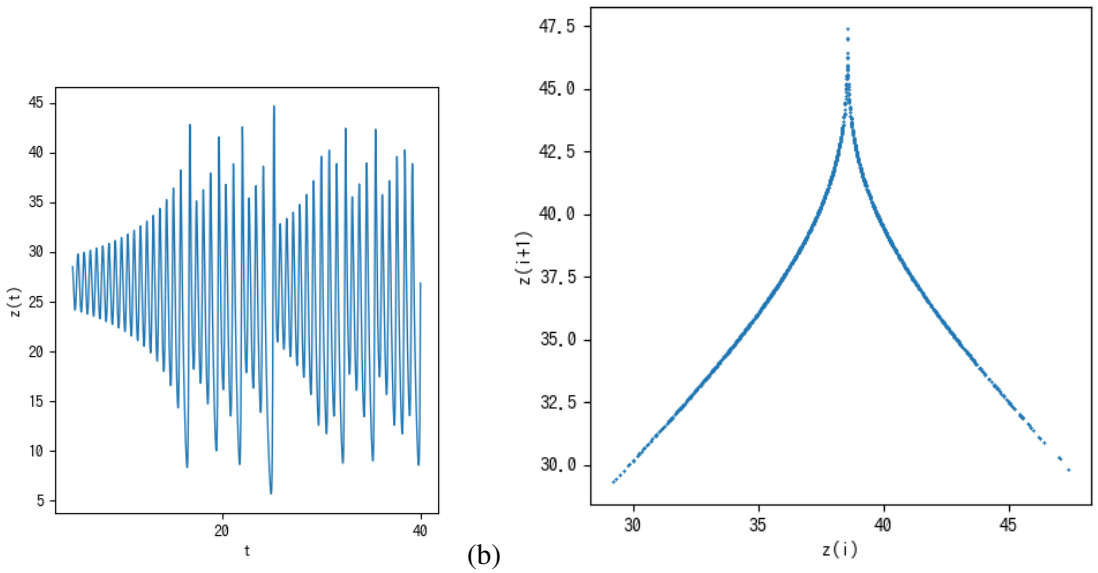
図 6.5(b) はプログラム 6.5-3: `lorenz_plot.py` で描いたした Lorenz プロットの様子で

す（プログラム 6.5-1 と同様に軌道行列 `orbit` を計算するまではプログラム 6.4-1 と同じです)。プログラム**??**では、軌道の z-成分列 `zorbit` の極大値のインデックス `zmaxid` の取り出しに、自前の関数 `maxpeakid` を定義しましたが、SciPy の信号処理モジュール signal の関数 `signal.argrelmax` を使って次のように極大値のインデックス (Fault/True) を取り出すこともできます。

コード 6.5-2　scipy.signal パッケージを使う

```
from scipy import signal
...
zmaxid = signal.argrelmax(zorbit, order=1)
...
```

図 6.5(b) は、節 6.5.1 のポアンカレ写像の場合と同じように、区間 $I = [30, 47.5]$ 上のテント形状の連続写像 $L : I \to I$ が存在して点列 $\{z_i, z_{i+1}\}$ が $z_{i+1} = L(z_i)$ によって逐次的に計算できるとみなすことができと強く示唆します。また同様に、区間 I 上の傾きが $dL/dz > 1$ になっていることから区間 I 上に Lebesgue 測度に絶対連続な不変測度が存在します。このことから、結果、Lorenz は点列 $\{z_i\}$ の非周期性、つまり軌道の非周期性を説明したのです。



(a)　(b)

図 6.5　(a) Lorenz の微分方程式 (6.6)（パラメータ $\sigma = 10, b = 8/3, r = 28$）の $z(t)$-成分の挙動。(b) $z(t)$ に関する Lorenz プロット

コード 6.5-3　lorenz_plot.py

```
.....
```

```
def maxpeakid(seq: np.ndarray) -> List[int]:# 数値列 seq の極大値インデックス
    maxid = [seq[k-1] < seq[k] and  seq[k] > seq[k+1] for k in range(1, len(
        seq)-1)]
    maxid.insert(0, False)# 先頭は常にFalse
    maxid.append(False)# 末尾は常にFalse
    return(maxid)

transitstep = 500 # 推移時刻 transitstep 以降の極大値を計算
zorbit = orbit[transitstep:,2]
zmaxid = maxpeakid(zorbit)
zmaxlist = zorbit[zmaxid]
zlist = np.delete(zmaxlist,-1)# 末尾を除去
nzlist = np.delete(zmaxlist, 0)# 先頭を除去

fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.set(xlabel = 'z(i)', ylabel = 'z(i+1)')
ax.plot(zlist, nzlist, ".", markersize=1)
#fig.savefig('lorenz_plot.png')
plt.show()
```

6.6 非自励微分方程式系

ベクトル場が時間に陽に依存して

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{V}(\boldsymbol{x}(t), t), \quad \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \tag{6.13}$$

と表される微分方程式を**非自励系** (non-autonomous) といいます。

まず自励ベクトル場として、2 重井戸型ダフィング方程式

$$\begin{aligned} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -x^3 + x \end{aligned} \tag{6.14}$$

を考えてみます。系 (6.14) は W 字型をした 2 つの井戸の底を持つ 2 重井戸ポテンシャル $V(x) = \dfrac{x^4}{4} - \dfrac{x^2}{2} + \dfrac{1}{4}$ を持つ力学方程式です。実際、全エネルギー

$$H(x, y) = \frac{y^2}{2} + \frac{x^4}{4} - \frac{x^2}{2} + \frac{1}{4}$$

が保存されることが式 (6.14) を使って $DE/dt = 0$ であることより確認できます（$1/4$ はエネルギーの最小値が 0 となるように調整しています)。

式 (6.14) で定まる運動を知るために、ベクトル場の不動点は原点 $(0,0)$ 以外に $(\pm 1, 0)$ の 2 点があることに注意します。また、式 (6.14) の線形化方程式を定めるヤコビ行列は

$$\begin{bmatrix} 0 & 1 \\ 1-3x^2 & 0 \end{bmatrix}$$

となります。$x = \pm 1$ はポテンシャル $V(x)$ の底になっていることからベクトル場の不動点 $(\pm 1, 0)$ は安定で、実際、そこでの線形化行列の固有値は複素共役な純虚数 $\pm i\sqrt{2}$ です。一方、$x = 0$ の周りでポテンシャルは上に凸になっており、ベクトル場の不動点 $(0,0)$ での固有値は ± 1 となり $(0,0)$ は鞍点となり不安定です。

節 5.3.2 で紹介したように x, y-平面のける接ベクトルの様子を観察することもできますが、ここでは力学の立場から全エネルギー E を与えたときの等高線 $H(x,y) = E$ を調べてみます。図 6.6 はプログラム 6.6-1: `duffing_coutour.py` で描いたエネルギー等高線です。$E = 1/4$ のとき、不安定不動点 $(0,0)$ を 8 の字の交点とするセパラトリックスとなり、左右にある安定不動点 $(\pm 1, 0)$ を囲んでいます。この 2 重井戸の運動ではエネルギー等高線からわかるように、セパラトリックスで囲まれた $H(x,y) < 1/42$ 領域内から出発する軌道は、他方の領域に入ることなく自身が属する領域内部で往復運動をします。$H(x,y) > 1/4$ のセパラトリックスの外部から出発する軌道もまた往復運動します。

セパラトリックス上の点は原点から遠ざかる不安定曲線（多様体）をなし、同時に原点に近づく安定曲線（多様体）ともなっている**ホモクリニック軌道** (homoclinic orbit) になっています。

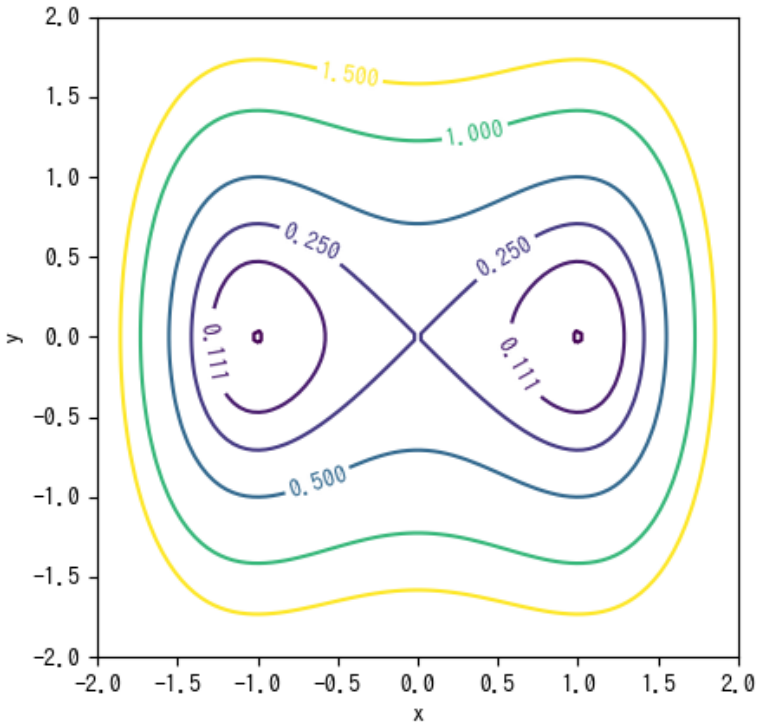


図 6.6　二重井戸ポテンシャル関数 $V(x) = \dfrac{x^4}{4} - \dfrac{x^2}{2} + \dfrac{1}{4}$ を持つ等エネルギー $H(x,y) = E$ 曲線。エネルギー値 1/4 の等高線は双曲不動点 $(0,0)$ を 8 の字の交点とするセパラトリックスで、$(0,0)$ から伸びた不安定曲線は安定曲線として自分自身に入るホモクリニック軌道になっている。

コード 6.6-1　duffing_coutour.py

```
import numpy as np
import matplotlib.pyplot as plt

# double-well
def double_well_vec(x: np.ndarray, t:float) -> np.ndarray:
    vx =  x[1]
    vy = -x[0] ** 3 + x[0]
    return(np.array([vx, vy]))

def double_well_energy(x, y: float) -> np.float_:
    return(x**4 / 4 - x**2 / 2 + y**2 / 2 + 1/4)

xg, yg = np.meshgrid(
    np.linspace(-2, 2, 100),
    np.linspace(-2, 2, 100))
```

```
fig, ax = plt.subplots()
ax.set_aspect('equal')
ax.set(xlabel = 'x', ylabel = 'y')
cont = ax.contour(xg, yg, double_well_energy(xg, yg), levels=[0.001, 1/9, 1/4,
    1/2, 1, 1.5])
ax.clabel(cont, inline=True, fontsize=10)
#fig.savefig('double_well_contour.png')
plt.show()
```

さて、非自励系として次のような減衰強制ダフィング方程式

$$\begin{aligned}\frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -x^3 + x - ky + B\sin t\end{aligned} \tag{6.15}$$

を考えてみましょう。項 $-ky$ は減衰項でパラメータ $k > 0$ によって減衰力を調整します。項 $B\sin t$ は外力としての強制振動を表し、その大きさを振幅 $B \geqq 0$ で調整します。上田 [47] は 1978 年に同様なダフィング方程式 $\frac{d2y}{dt^2} = -x^3 - ky + B\cos t$ を調べて、以下と同様な結果を得ています。

外力項の存在によってベクトル場が時間に陽に依存するため、相空間を x, y-平面的に考えずに、時間軸を入れた拡大相空間 $\mathbb{R}^2 \times \mathbb{R}$ を考えます。今の場合、ベクトル場は強制振動の周期 2π で変化するため時間方向として円周 $S_1 = [0, 2\pi)$ を考え $\mathbb{R}^2 \times S_n$ と考えて拡大相空間を $\mathbb{R}^2 \times S_1$ とするのが自然です。式 6.15 の流れ場もやは x, y-平面的ではなく、ポアンカレ=ベンディクソンの定理 6.2 の制約は受けません。

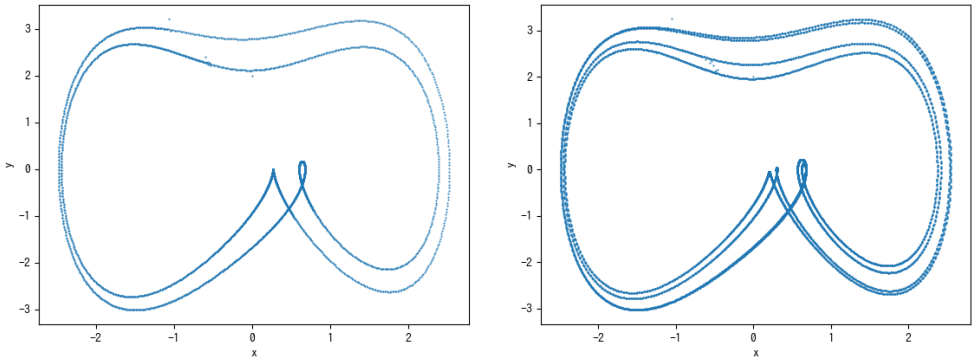
まず、外力がない $B = 0$ の場合を考察してみます。第 2 式の項 $-ky$（$k > 0$）は減衰力を表しており、パラメータ k が大きいほど全エネルギーは軌道に沿って減少し、最終的にはポテンシャルの井戸のどちらかの底 $x = \pm 1$ で運動は止まります。

しかしながら、周期的な外力 $B > 0$ が加わることによってポテンシャルの極小値に落ち着くという単純な動きはしなくなります。ベクトル場が周期 T で変化する場合の軌道の様子を表すために、拡大相空間内の流れ $\boldsymbol{x}(t)$ を周期 $T = 2\pi$ ごとに観察した点列 $\{\boldsymbol{x}(t + kT)\}_{k=0,1,2,\ldots}$ を観察する写像 $S : \mathbb{R}^2 \times S_1 \to \mathbb{R}^2$ を考えます。これを**ストロボ写像** (stroboscopic maping) といいます。

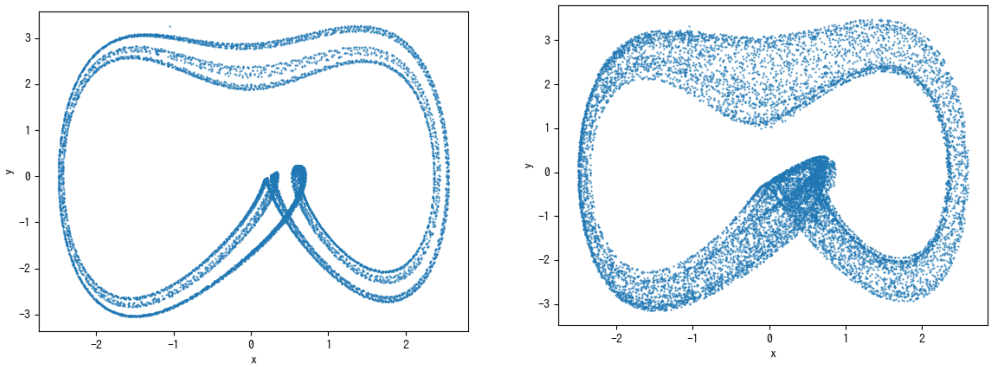
図 6.7 は、式 (6.15) について（$k = 0.29 \sim 0.27 =, B = 3$）、初期条件 $(0.0, 2.0)$ を固定し、減衰力 k を変化させたときのストロボ写像をプロットしたものです。減衰力 k が小さい時には、2 重ポテンシャルの 2 つの安定な井戸の底 $(x = \pm 1)$ を取り囲むような 2 重周期軌道に近傍にある軌道が引き込まれます。(b)$k = 0.28$、(c)$k = 0.27$ と k を増やしていくと倍化

(2 倍、2^2 倍、2^3 倍 ...) した周期軌道に引き込まれるようになります。(d) $k = 0.27$ あたりから 1 つの軌道が 1 つのバンド幅にあるようになり $k = 0.255$ では周期的な軌道要素は見られなくなります。周期軌道の倍化現象は、節 3.5.2 のロジスティック方程式のダイヤグラムで生じていた現象と似ていることに注意してください。

この減衰強制ダフィング方程式では、Lorenz 系の場合（$r = 28$）のように 1 つのアトラクタが周辺の軌道を吸引するのとは異なり、複数のアトラクタが入り混じって初期条件に依存して ω-極限集合は異なります。



(a) $k = 0.29$　　(b) $k = 0.28$



(c) $k = 0.275$　　(d) $k = 0.27$

図 6.7　減衰強制ダッフィング方程式 (6.15) の xy-平面へのストロボ射影（パラメータ $k = 0.29 \sim 0.27 =, B = 3$）。初期条件 $(0.0, 2.0)$ を固定し、減衰力 k を変化。(a) $k = 0.29$ では 2 重ポテンシャルの 2 つの安定な井戸の底を取り囲むような 2 重周期軌道に近傍にある軌道が引き込まれる。(b),(c) k を増やして生じる周期倍化軌道に引き込まれる。(d) $k = 0.255$ になると周期成分は見当たらない。

コード 6.6-2　forced_duffing.py

```
import numpy as np
```

```
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from typing import List

# forced+damped duffing
def forced_duffing(x: np.ndarray, t:float, k, B:float) -> np.ndarray:
    vx =  x[1]
    vy = -x[0] ** 3 + x[0] - k*x[1] + B * np.sin(t)
    return(np.array([vx, vy]))

# cycleごとの軌道indexを求める
def stroboID(T, dt, cycle:float) ->  List[int]:
     shotNum = int(T/cycle)
     cycle_idx = [int(cycle/dt)*k for k in range(shotNum)]
     return(cycle_idx)

k = 0.27 # 減衰力
B = 3.0 # 周期外力
x0 = [0.0, 2.0]
t0, T = 0, 60000 # 時間間
dt = 0.005
times = np.arange(t0, T, dt)
args = (k, B)
orbit = odeint(forced_duffing, x0, times, args)
strobo_orbit = orbit[stroboID(T, dt, 2*np.pi)]

fig, ax = plt.subplots()
#ax.set_aspect('equal')
ax.set(xlabel = 'x', ylabel = 'y')
ax.plot(strobo_orbit[:,0], strobo_orbit[:,1], '.', markersize=1)
#fig.savefig('forced_duffing.png')
plt.show()
```

第 7 章　古典力学の軌道

7.1　Hamilton 系

Newton の第 2 法則「位置 $\boldsymbol{r} = (x_1, x_2, \ldots, x_n)$ に働く力 $\boldsymbol{F} = (f_1, \ldots, f_n)$ はその質量 m と加速度 $d^2\boldsymbol{r}/dt^2$ の積に等しい」は、質量と速度の積を**運動量** (momentum) $\boldsymbol{p} = md\boldsymbol{r}/dt$ を導入して**運動方程式** (equation of motion)

$$\frac{d\boldsymbol{p}}{dt} = \boldsymbol{F}$$

で記述されます。この運動方程式で記述される力学系は自由度 n を持つといいます。

ここでは、運動方程式の右辺の力 $\boldsymbol{F} = (f_1, \ldots, f_n)$ がある実数値関数の勾配

$$\boldsymbol{F} = -\mathrm{grad}V(\boldsymbol{r}) = \begin{bmatrix} -\frac{\partial V}{\partial x_1} \\ \vdots \\ -\frac{\partial V}{\partial x_n} \end{bmatrix} \tag{7.1}$$

で与えられる力学系を**保存系** (conservative system) と呼びます。ここで、$V(\boldsymbol{r})$ は位置 $\boldsymbol{r}$ の実数値関数で**ポテンシャル** (potential) と呼びます。このとき、働く力をポテンシャル力といいます。

保存系の運動方程式として、すでに節 5.3.2 では $V(x) = kx^2/2$ とする調和振動子 (5.20) を、節 5.4.4 では $V(x) = k(1 - \cos x)$ とする単振子 (5.31) を取り扱いました（どちらも自由度 1 でした）。

運動方程式がポテンシャル力によって記述される力学系では、運動量 $\boldsymbol{p} = (p_1, p_2, \ldots, p_n)$ から定まる運動エネルギー

$$T = \frac{\boldsymbol{p}^2}{2m} = \frac{1}{2m}\sum_{i=1}^{n} p_i^2 \tag{7.2}$$

とポテンシャル関数 V との和である全エネルギー $H(\boldsymbol{r}, \boldsymbol{p}) = T + V$ の値は運動の初期条件 $(\boldsymbol{r}_0, \boldsymbol{p}_0)$ で決まる定数 $H(\boldsymbol{r}, \boldsymbol{p}) = E$ になります。実際、式 (7.1) を使って次のようにして確

かめることができ、力学エネルギーの保存則が成立します。

$$\begin{aligned}\frac{d}{dt}(T+V) &= \sum_{i=1}^{n}\left(\frac{\partial T}{\partial p_i}\frac{dp_i}{dt}+\frac{\partial V}{\partial x_i}\frac{dx_i}{dt}\right)\\ &= \sum_{i=1}^{n}\left(\frac{p_i}{m}\frac{dp_i}{dt}-f_i\frac{dx_i}{dt}\right)\\ &= 0\end{aligned}$$

関数 $H(\boldsymbol{r},\boldsymbol{p})$ はエネルギー関数、あるいは**ハミルトニアン** (Hamiltonian) と呼びます。

力学の一般的取り扱いは解析力学として深く研究されてきました [20][21]。位置 $\boldsymbol{r}$ を表すために直交座標系を用いる必要はなく、位置や運動量を一義的に表すような別の座標系（広義座標系といいます）を使っても構いません。以下、ハミルトニアン H は広義位置 $\boldsymbol{x}$ および広義運動量 $\boldsymbol{p}$ で与えられているとします。

ハミルトニアン $H(\{x_i\},\{p_i\})$ が与えられているとき、運動方程式は $2n$ 個の Hamilton の**正準方程式** (canonical equations)

$$\begin{cases}\dfrac{dx_i}{dt} &= \dfrac{\partial H}{\partial p_i}, \qquad i=1,\ldots,n\\ \dfrac{dp_i}{dt} &= -\dfrac{\partial H}{\partial x_i}\end{cases} \tag{7.3}$$

で与えられます（ハミルトンの運動方程式ということもあります)。逆に、時間を陽に含まないハミルトニアン $H(\boldsymbol{x},\boldsymbol{p})$ の値は正準方程式 (7.3) から一定であることがわかります。関数 H を**エネルギー積分**と呼ぶことがあります。

$$\begin{aligned}\frac{d}{dt}H(\{q_i\},\{p_i\}) &= \sum_{i=1}^{n}\left(\frac{\partial H}{\partial q_i}\frac{dq_i}{dt}+\frac{\partial H}{\partial p_i}\frac{dp_i}{dt}\right) = \sum_{i=1}^{n}\left(\frac{\partial H}{\partial q_i}\frac{\partial H}{\partial p_i}-\frac{\partial H}{\partial p_i}\frac{\partial H}{\partial q_i}\right)\\ &= 0\end{aligned}$$

たとえば、ハミルトニアンが

$$H=\frac{p^2}{2m}+\frac{\omega^2x^2}{2}$$

で与えられると、正準方程式から得られる式

$$\begin{aligned}\frac{dx}{dt} &= \frac{p}{m}\\ \frac{dp}{dt} &= -\omega^2 q\end{aligned}$$

は調和振動子の式 (5.3.2.1)($m=1$) となります。

さて、ハミルトンの正準方程式 (7.3) は広義座標 $(\{x_i\},\{p_i\})$ で表される $2n$-次元空間の**相空間** (phase space)M 上のベクトル場 $\boldsymbol{V}_H$ としてみてみると特別な形をしています。M の座標を $\boldsymbol{z}=(z_1,\ldots,z_n,z_{n+1},\ldots,z_{2n})$ と書いて

$$z_i=\begin{cases} q_i & (1\leqq i\leqq n)\\ p_{i-n} & (n<i\leqq 2n)\end{cases}$$

とすると、正準方程式は次のように表すことができます。

$$\frac{dz_i}{dt}=\sum_{j=1}^{2n}J_{ij}\frac{\partial H}{\partial z_i},\qquad i=1,\ldots,2n. \tag{7.4}$$

ここで、行列 J は n 次の**シンプレクティック行列** (symplectic matrix) で、零行列 $\mathbf{0}$ と単位行列 $\mathbf{1}$ を使って

$$J=\begin{bmatrix}\mathbf{0} & \mathbf{1}\\ -\mathbf{1} & \mathbf{0}\end{bmatrix} \tag{7.5}$$

と表されます。このように、ハミルトン力学系はシンプレクティック構造を持つベクトル場として定義されています。

ハミルトン力学系が有するシンプレクティック構造は正準方程式を与えるベクトル場 $\boldsymbol{V}_H$ の発散が

$$\begin{aligned}\mathrm{div}\boldsymbol{V}_H&=\sum_{k=1}^{n}\left(\frac{\partial^2 H}{\partial q_k\partial p_k}-\frac{\partial^2 H}{\partial p_k\partial q_k}\right)\\ &=0\end{aligned} \tag{7.6}$$

であることがわかります（ただし、ベクトル場の発散が 0 であっても Hamilton 系であるとは限りません)。Lorenz ベクトル場が負の一定値となる式 (6.12) となるとは対照的です。

ハミルトン力学系では相空間 M 内の閉曲面で囲まれた初期領域 $D(t_0)$ の体積 $\mathrm{Vol}(t_0)|$ は時間とともに領域の形を変えていきますがその体積 $\mathrm{Vol}(t)$ は変わりません。エノンの面積保存写像（節 3.2.3）で紹介した式 (3.2) を満たす面積保存写像に対応していることに注意してください。

このことは次のようにも説明することができます。M 上のハミルトンベクトル場 $\boldsymbol{V}_H(\{q_k\},\{p_k\})$ の流れを密度 $\rho(\{q_k\},\{p_k\})$ を持つ流体運動と見なしたとき、質量保存則から要請される連続の式

$$\frac{\partial\rho}{\partial t}+\mathrm{div}(\rho\boldsymbol{V}_H)=0$$

において、$\mathrm{div}\boldsymbol{V}_H = 0$ であることから

$$\frac{\partial \rho}{\partial t} + \boldsymbol{V}_H \cdot \mathrm{grad}\, \rho = \frac{d\rho}{dt} = 0$$

が導かれ、ハミルトン流は密度変化のない非圧縮性流体であると考えることができます。

7.2 Hénon-Heiles の Hamilton 系

Hénon-Heiles[34] は次の自由度 2 のハミルトニアンで定まる運動を数値計算によって詳しく調べました。

$$H(x, y, p_x, p_y) = \frac{1}{2}(p_x^2 + p_y^2) + \frac{1}{2}(x^2 + y^2) + x^2 y - \frac{1}{3}y^3 \tag{7.7}$$

この H で定まる正準方程式 (7.3) は次のようになります。

$$\begin{aligned} \frac{dx}{dt} &= p_x \\ \frac{dy}{dt} &= p_y \\ \frac{dp_x}{dt} &= -x - 2xy \\ \frac{dp_x}{dt} &= -y - x^2 + y^2. \end{aligned} \tag{7.8}$$

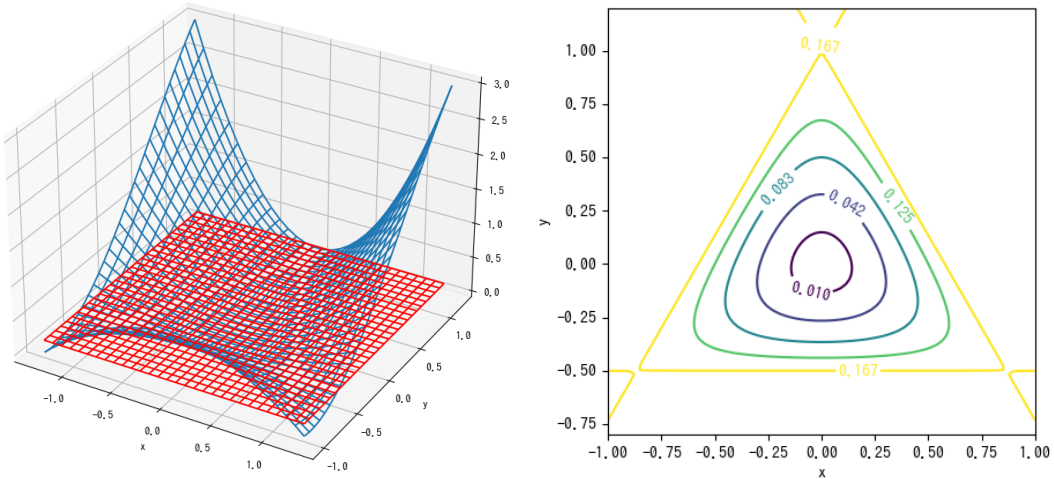


図 7.1　Hénon-Heiles 系 (7.7) のポテンシャル関数 $V(x,y) = \frac{1}{2}(x^2+y^2) + x^2y - \frac{1}{3}y^3$ の様子。(a) ポテンシャル曲面が高さ $E = 1/6$ の平面と交差する様子。(b) ポテンシャルの $E = 0.01, 0.0417, 1/12, 1/8, 1/6$ とした等高線（零速度曲線）プロット。$E \leqq 1/6$ で零速度曲線は有界な閉曲線、$E = 1/6$ で正三角形となり、$E > 1/6$ ではポテンシャルの井戸を超えて零速度曲線は無限遠に広がってしまう。

図 7.1(a) は、Hénon-Heiles 系のポテンシャル関数

$$V(x,y) = \frac{1}{2}(x^2+y^2) + x^2 y - \frac{1}{3}y^3 \tag{7.9}$$

の x,y-平面上で定めるポテンシャル曲面とエネルギー値 E で定まる曲面 $z = E$ と交差する様子を示しています。ポテンシャル曲面のワイヤフレーム `plot_wireframe` を使う図示についてはすでに節 1.4.3 で取り上げています。図 7.1(b) は、プログラム**??**やプログラム 6.6-1 と同様にして、ポテンシャルの等高線を全エネルギー値 $E = 0.01, 0.0417, 1/12, 1/8, 1/6$ について描いた零速度曲線です。

図 7.1 からわかるように、全エネルギー E が $E < 1/6$ では零速度曲線は x,y-平面内のおむすび状の閉曲線となり、ハミルトン軌道の位置 x,y はこの領域内に留まります ($E = 1/6$ のとき零速度曲線は正三角形となります)。$E > 1/6$ を超えるとポテンシャル井戸の谷間の高さを超え、零速度曲線は x,y-平面を無限遠に広がります。このことによって、臨界エネルギー値 $E = 1/6$ 以上ではハミルトン軌道はこの隙間から無限遠に遠ざかることが可能になります。

これらの観察から、Hénon-Heiles 系の運動を調べるためには臨界エネルギー値 $E = 1/6$ 以下の有限領域内に留まり続ける条件のもとで、さまざまなエネルギー値に応じた初期条件を設定してその長時間挙動を調べることになります。

図 7.2 は Hénon-Heiles の運動方程式 (7.8) をエネルギー値 $E = 1/8$ のとき、平面 $x = 0$ を $dx/dt > 0$ の方向で横切る軌道のポアンカレ切断面（y,p_y-平面）上の点列を初期条件 $x_0 = 0$ および $y_0 \in [-0.39, 0.64]$ についてプロットしたものです。ここでは、プログラム 7.2-1 のように、エネルギー曲面 $H(x,y,p_x,p_y) = E$ が $x = 0, p_x = 0$ を通ることで得られる曲線も等高線を描く `contour` を使って描いています。図 7.2 から、この $E = 1/8$ のエネルギー曲面には周期点（ポアンカレ切断面上の不動点）の周りを囲む島状の不変曲線上を準周期的に運動する領域とあたかもランダムに動き回る乱雑軌道の領域が混在している様子を観察できます。エネルギー値が低い（$E < 1/8$）場合は不変曲線が存在する領域は多くありますが、$E = 1/8$ を超えるとエネルギー曲面上の乱雑軌道の領域は急激に増加し臨界エネルギー値 $E = 1/6$ にごく近いときには乱雑軌道の領域はエネルギー曲面の殆どを占めることが知られています [34, Fig.7]。

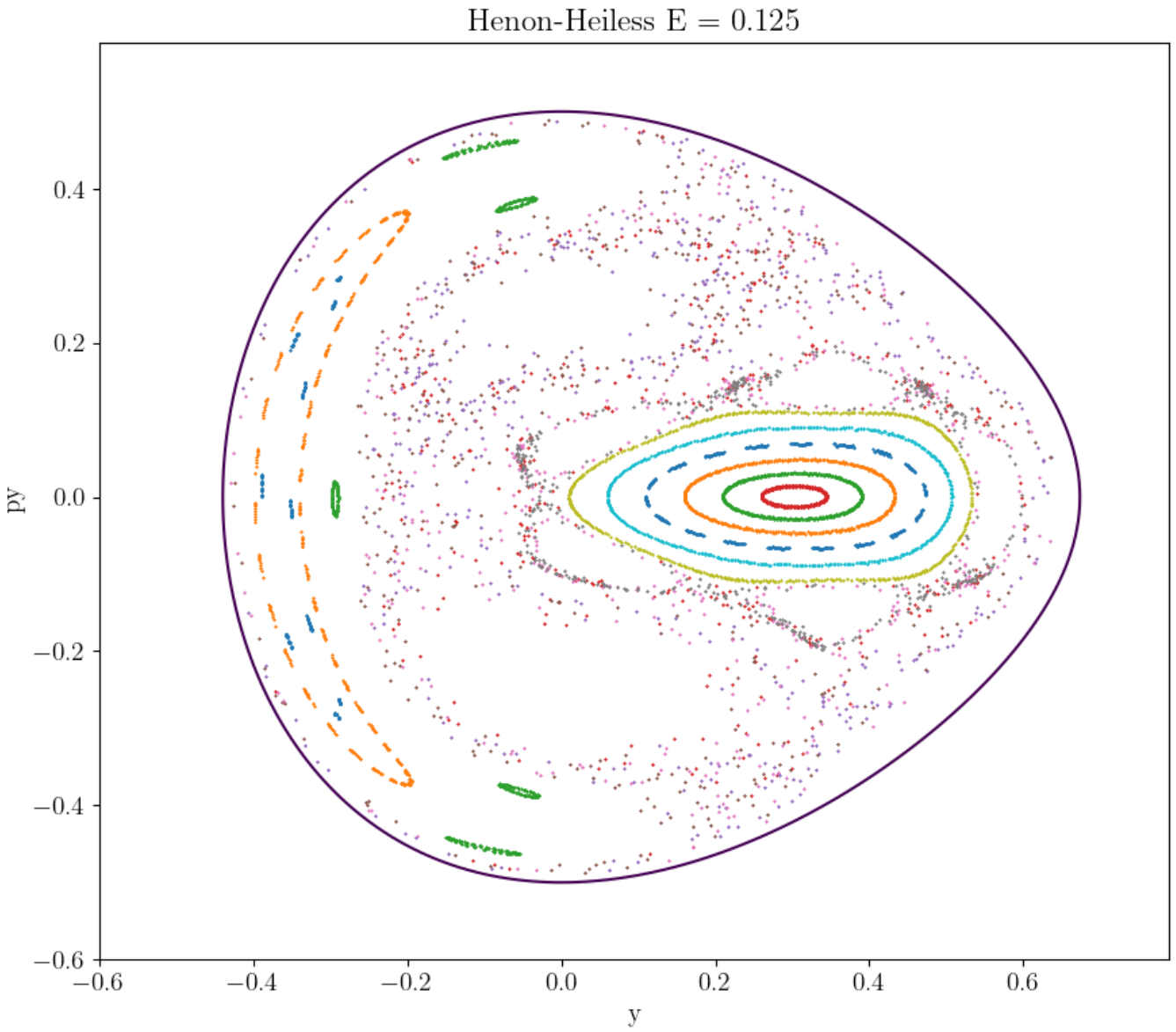


図 7.2　エネルギー $E = 1/8$ を持つ Hénon-Heiles 系 (7.7). 平面 $x = 0$ を $dx/dt > 0$ の方向で横切る軌道の y, p_y-Poincaré 切断面上の点列を様々な初期条件 $x_0 = 0$ および $y_0 \in [-0.39, 0.3]$ についてプロット。

コード 7.2-1　henon-heiles_section.py

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
from typing import Callable, List
```

```
def henon_heiless(x: np.ndarray, t: float) -> np.ndarray:
    dx = x[2]
    dy = x[3]
    dpx = -x[0] - 2.0 * x[0] * x[1]
    dpy = -x[1] - x[0] ** 2 + x[1] ** 2
    return(np.array([dx, dy, dpx, dpy]))

# Total energy
def energy(x, y, px, py):
    return((px ** 2 + py ** 2) / 2.0 + (x ** 2 + y ** 2) / 2.0 + (x ** 2) * y
        -  y ** 3 / 3.0)

# give a momentum px from given energy e, y, py and the condion x=0
def give_px_from_Energy(e, y, py: float) -> np.float_:
    return(np.sqrt(2.0 * e - py ** 2 - y ** 2 + 2.0/3.0 * y ** 3))

# surface of section 数列 seqがvalue値を正方向に横断するインデックス
def section_id(seq: np.ndarray, value: float) -> List[int]:
    crossed = [(seq[k] - value) < 0 and (seq[k+1] - value) > 0 for k in range(
        len(seq)-1)]
    crossed.append(False)# 末尾は常にFalse
    return(crossed)

# setting
t0 = 0#初期時間
T = 3000#最終時間
dt = 0.01#時間間隔
times = np.arange(t0, T, dt)

fig = plt.figure(figsize=(10.0, 10.0))
ax = plt.axes()
ax.set_aspect('equal')

E = 0.125# a given energy
for y0 in np.arange(-0.39, 0.3, 0.05):
    x0 = 0.0
    py0 = 0.0# initail momentum py0
```

```
        px0 = give_px_from_Energy(E, y0, py0)
        x = [x0, y0, px0, py0]# initial point
        orbit = odeint(henon_heiless, x, times)
        sections = orbit[section_id(orbit[:,0], 0)] # x=0をdx/dt>0で横断
        ax.plot(sections[:,1], sections[:,3], "x", markersize=1)

yg, pyg = np.meshgrid(np.arange(-0.6, 0.8, 0.01),np.arange(-0.6, 0.6, 0.01))
# x = 0, px = 0 を通る energy曲線 with E
cont = ax.contour(yg, pyg, energy(0, yg, pyg, 0), levels=[E])
ax.set(xlabel = 'y', ylabel = 'py', title='Henon-Heiless  E = ' + str(E))
#fig.savefig('henon-heiless_section.png')
plt.show()
```

7.3 SymPy ハミルトン関数からベクトル場を算出して軌道計算する

Hénon-Heiles 系の運動方程式 7.8 はハミルトンの正準方程式 (7.3) から得られるため、SymPy を使ってハミルトニアン関数 H を偏微分することによって一般的にハミルトンベクトル場を導いて、軌道計算することを考えてみましょう。節 2.4.6 で紹介した SymPy 関数を NumPy 関数に変換する `sympy.lambdify` を使います。

プログラム 7.2-1: `henon-heiles_section_sympy` では、SymPy 変数 `sx, sy, spx, spy` を宣言してから自由度 2 の Hénon-Heiles 系のハミルトン関数を与えています。自由度に応じた変数の調整はユーザの責任としています。これより、直ちに `sympy.lambdify` を使って NumPy 関数としてエネルギー関数が定義できます。運動方程式については、SymPy ハミルトン関数と引数とする関数 `eq_motions` として、正準方程式 (7.3) の定義に習って偏微分を実行した SymPy 関数 `sdx,sdy,sdpx,sdpy` を返すように定義します。このとき一般的なハミルトニアンを想定して、これらの SymPy 関数は 4 つの引数を持つように定義しています。

最後に、scipy.odeint を使って微分方程式の軌道行列を計算するための関数 `hamilton_vectorfield` として、`eq_motions` で返される NumPy 関数に引数で与えた 1 次元 NumPy 配列 `x` を代入した値を返すように定義します。実行速度は遅くなりますが、万有引力の元での運動などでハミルトニアンから運動方程式を計算する手間がかかりません。

図 7.3 は、Hénon-Heiles のハミルトニアン (7.7) の場合のプログラム**??**を使って、臨界エネルギーエネルギー値 $E = 1/6$ にごく近い $E = 0.1665$ となる初期条件 $(0.75, -0.3, 0, 0)$ か

ら出発した経過時間 $T = 4000$ までの軌道を y, p_y-Poincaré 切断面上にプロットしたものです。軌道がエネルギー曲面上の大部分に渡って動き回る様子がわかります。

コード 7.3-1　henon-heiles_section_sympy.py

```
import numpy as np
import sympy as sym
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
from typing import List

# Henon-Heilesのハミルトニアン
sx,sy, spx, spy = sym.symbols('sx sy spx spy')
hamiltonian = (spx**2 + spy**2)/2 + (sx**2 + sy**2)/2 + sx**2 * sy -sy**3/3

# Total energy energy(x, y, px, py)
energy = sym.lambdify([sx,sy,spx,spy], hamiltonian, modules="numpy")

def eq_motions(hamiltonian): # 自由度2のハミルトンの運動方程式の右辺の関数
    sdx = sym.lambdify([sx,sy,spx,spy], hamiltonian.diff(spx), modules="numpy
        ")
    sdy = sym.lambdify([sx,sy,spx,spy], hamiltonian.diff(spy), modules="numpy
        ")
    sdpx = sym.lambdify([sx,sy,spx,spy], -hamiltonian.diff(sx), modules="numpy
        ")
    sdpy = sym.lambdify([sx,sy,spx,spy], -hamiltonian.diff(sy), modules="numpy
        ")
    return(sdx,sdy,sdpx,sdpy)

def hamilton_vectorfield(x: np.ndarray, t: float) -> np.ndarray: # odeint用の
    ベクトル場
    sdx, sdy, sdpx, sdpy = eq_motions(hamiltonian)
    dx = sdx(x[0],x[1],x[2],x[3])
    dy = sdy(x[0],x[1],x[2],x[3])
    dpx = sdpx(x[0],x[1],x[2],x[3])
    dpy = sdpy(x[0],x[1],x[2],x[3])
    return(np.array([dx, dy, dpx, dpy]))

def section_id(seq: np.ndarray, value: float) -> List[int]:
```

```
    crossed = [(seq[k] - value) < 0 and (seq[k+1] - value) > 0 for k in range(
        len(seq)-1)]
    crossed.append(False)# 末尾は常にFalse
    return(crossed)

t0 = 0#初期時間
T = 500#最終時間
dt = 0.01#時間間隔
times = np.arange(t0, T, dt)
x0, y0 = 0.75, -0.3
px0, py0 = 0.0, 0.0
x = [x0, y0, px0, py0] # E=0.1665
orbit = odeint(hamilton_vectorfield, x, times)
sections = orbit[section_id(orbit[:,2], 0)]

fig = plt.figure(figsize=(10.0, 10.0))
ax = plt.axes()
ax.set_aspect('equal')
ax.plot(sections[:,1], sections[:,0], "x", markersize=1) # x=0をdx/dt>0で横断
#ax.plot(orbit[:,1], orbit[:,3], "x", markersize=1)

yg, pyg = np.meshgrid(np.arange(-0.6, 1.0, 0.01),np.arange(-0.6, 0.6, 0.01))
E = energy(x0, y0, px0, py0)
# x = 0, px = 0 を通る energy曲線 with E
cont = ax.contour(yg, pyg, energy(0, yg, pyg, 0), levels=[E])
ax.set(xlabel = 'y', ylabel = 'py', title='Henon-Heiless  E = ' + str(E))
plt.show()
```

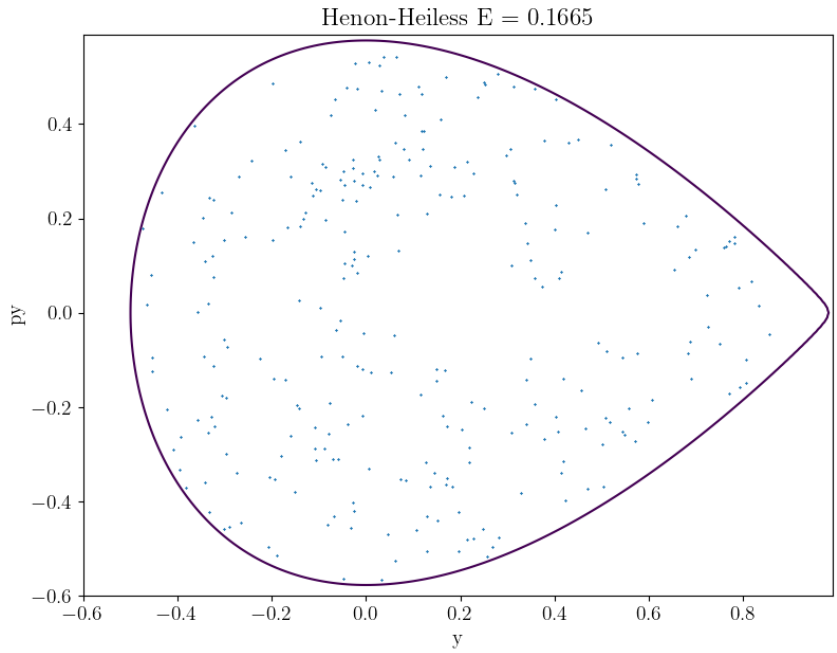


図 7.3　Hénon-Heiles 系 (7.7) で臨界エネルギーエネルギー値 $E = 1/6$ にごく近い $E = 0.1665$ となる初期条件 $(0.75, -0.3, 0, 0)$ から出発した軌道の y, p_y-Poincaré 切断面上の経過時間 $T = 4000$ までの点列。

7.4　万有引力

Newton が発見した万有引力の法則とは、位置 $\boldsymbol{x}_1 = (x_{1x}, x_{1y}, x_{1z})$, $\boldsymbol{x}_2 = (x_{2x}, x_{2y}, x_{2z})$ にある質量 m_1, m_2 の質点 1 と質点 2 の間に質量の積 $m_1 m_2$ に比例し、距離の 2 乗 $r_{12} = |\boldsymbol{x}_1 - \boldsymbol{x}_2|^2$ に反比例する大きさの引力が 2 点間を結ぶ方向に働く

$$\boldsymbol{f}_{12}(\boldsymbol{x}_1, \boldsymbol{x}_2) = -\boldsymbol{f}_{21}(\boldsymbol{x}_1, \boldsymbol{x}_2) = G\frac{m_1 m_2}{r_{12}}\frac{\boldsymbol{x}_2 - \boldsymbol{x}_1}{{r_{12}}^2} \tag{7.10}$$

とうものです。ここで、G は万有引力定数、r_{12} は 2 質点間の距離

$$r_{12} = \sqrt{(x_{1x} - x_{2x})^2 + (x_{1y} - x_{2y})^2 + (x_{1z} - x_{2z})^2}$$

を表します。式 (7.10) において、$(\boldsymbol{x}_2 - \boldsymbol{x}_1)/r_{12} = \boldsymbol{e}_{21}$ は質点 2 から質点 1 への単位ベクトルとなっており、2 質点間に作用する力の向きは互いに反対の向き（引力）となっています。万有引力のように距離だけに依存して働く力を**中心力**といいます。

7.4.1 ケプラー問題

2 質点が互いに万有引力に従って運動するという **Kepler 問題**は自由度 $3 \times 2 = 6$ の力学系で、式 (7.10) から直ちに次の 6 つの運動方程式が得られます。

$$\begin{cases} m_1 \dfrac{d^2\boldsymbol{x}_1}{dt^2} = G\dfrac{m_1 m_2}{r_{12}^2}\dfrac{\boldsymbol{x}_2 - \boldsymbol{x}_1}{r_{12}}, \\ m_2 \dfrac{d^2\boldsymbol{x}_2}{dt^2} = -G\dfrac{m_1 m_2}{r_{12}^2}\dfrac{\boldsymbol{x}_2 - \boldsymbol{x}_1}{r_{12}}. \end{cases} \tag{7.11}$$

運動方程式 (7.11) はハミルトニアン

$$H(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{p}_1, \boldsymbol{p}_2) = \frac{\boldsymbol{p}_1^2}{2m_1} + \frac{\boldsymbol{p}_2^2}{2m_2} - \frac{Gm_1 m_2}{r_{12}} \tag{7.12}$$

を使った正準方程式 (7.3) からも得られます。詳しくは力学のテキストに譲りますが、質点 1 と質点 2 の重心座標 $\boldsymbol{x}_G = (m1\boldsymbol{x}_1 + m2\boldsymbol{x}_2)/m_1 m_2$ を導入すると重心が等速直線運動をすること（運動量保存則）によって、自由度 3 の運動方程式

$$\frac{d^2\boldsymbol{x}}{dt^2} = \frac{K\boldsymbol{x}}{r^3}, \quad r = \sqrt{x^2 + y^2 + z^2}, K = Gm_1^3/(m_1 + m_2)^2$$

が得られます。中心力であることから、運動は 1 つの平面上で起こることもわかります。そこであらためて座標をその平面上に選ぶと、$z = 0$ と選んだ x, y に関する運動方程式（と対応するハミルトニアン $H(x, y, p_x, p_y)$）が得られ、x, y-平面上で楕円を描きます。さらに、中心力であるための角運動量が保存される結果、最終的には自由度 1 の力学系に帰着されるのでした。

図 7.4 は、自由度 6 の運動方程式 (7.11) に対応するハミルトンベクトル場をプログラム 7.4-1: `kepler.py` を使って（節 7.3 参照）、SymPy 関数としてハミルトニアン (7.12) から求めた上で NumPy 関数化し、scipy.odeint を使って軌道行列を計算し、2 質点の位置座標 $\boldsymbol{x}_1, \boldsymbol{x}_2$ をプロットしています。プログラムは複雑でなように見えまずが、プログラム 7.3-1 と本質的には同じです。自由度 6 であるためベクトル場を記述するための 12 個の変数が必要になり長大になってしまいました。

コード 7.4-1　kepler.py

```
import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
from scipy.integrate import odeint
```

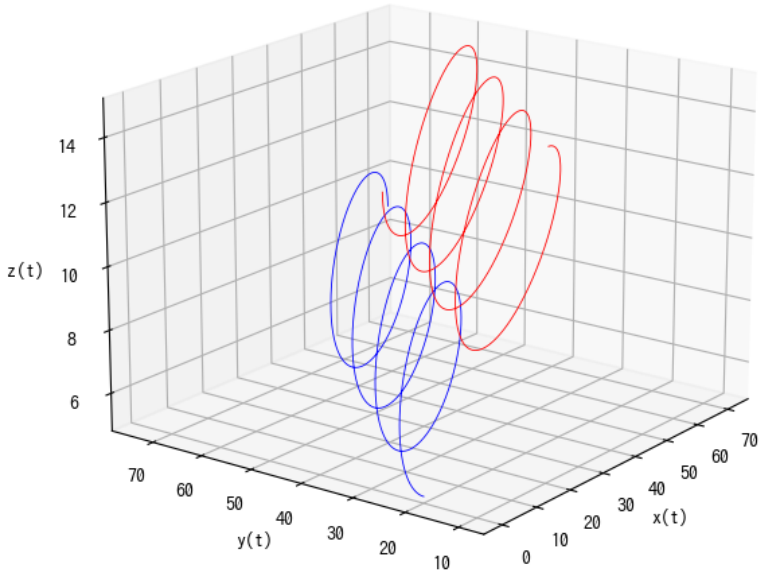


図 7.4　自由度 6 とした 2 質点が万有引力にしたがって 3 次元位置を移動する様子 ($G = 1, m_1 = 1, m_2 = 0.8$)。2 質点の重心は等速直線運動し、その周りを 2 質点が楕円軌道を描きながら移動していく。

```
G = 1  #引力定数
m1 = 1 # 質量
m2 = 0.8
sx1,sy1,sz1, spx1,spy1,spz1 = sym.symbols('sx1 sy1 sz1 spx1 spy1 spz1')
sx2,sy2,sz2, spx2,spy2,spz2 = sym.symbols('sx2 sy2 sz2 spx2 spy2 spz2')
hamiltonian = (spx1**2 + spy1**2 + spz1**2)/(2*m1) \
   + (spx2**2 + spy2**2 + spz2**2)/(2*m2) \
   - G*m1*m2/sym.sqrt((sx1-sx2)**2 + (sy1-sy2)**2 + (sz1-sz2)**2)

def eq_motions(hamiltonian): # 自由度6のハミルトンの運動方程式の右辺の関数
    sdx1 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,spz2
        ],\
        hamiltonian.diff(spx1), modules="numpy")
    sdy1 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,spz2
        ],\
        hamiltonian.diff(spy1), modules="numpy")
    sdz1 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,spz2
        ],\
        hamiltonian.diff(spz1), modules="numpy")
    sdx2 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,spz2
        ],\
```

```
        hamiltonian.diff(spx2), modules="numpy")
    sdy2 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,spz2
        ],\
        hamiltonian.diff(spy2), modules="numpy")
    sdz2 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,spz2
        ],\
         hamiltonian.diff(spz2), modules="numpy")
    sdpx1 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,
        spz2],\
        -hamiltonian.diff(sx1), modules="numpy")
    sdpy1 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,
        spz2],\
        -hamiltonian.diff(sy1), modules="numpy")
    sdpz1 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,
        spz2],\
        -hamiltonian.diff(sz1), modules="numpy")
    sdpx2 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,
        spz2],\
        -hamiltonian.diff(sx2), modules="numpy")
    sdpy2 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,
        spz2],\
        -hamiltonian.diff(sy2), modules="numpy")
    sdpz2 = sym.lambdify([sx1,sy1,sz1,sx2,sy2,sz2,spx1,spy1,spz1,spx2,spy2,
        spz2],\
        -hamiltonian.diff(sz2), modules="numpy")
    return(sdx1,sdy1,sdz1,sdx2,sdy2,sdz2,sdpx1,sdpy1,sdpz1,sdpx2,sdpy2,sdpz2)

def hamilton_vectorfield(x: np.ndarray, t: float, G, m1, m2: float) -> np.
    ndarray: # odeint用のベクトル場
    sdx1,sdy1,sdz1,sdx2,sdy2,sdz2, sdpx1,sdpy1,sdpz1,sdpx2,sdpy2,sdpz2 =
        eq_motions(hamiltonian)
    dx1 = sdx1(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x[11])
    dy1 = sdy1(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x[11])
    dz1 = sdz1(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x[11])
    dx2 = sdx2(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x[11])
    dy2 = sdy2(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x[11])
    dz2 = sdz2(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x[11])
    dpx1 = sdpx1(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x
```

```
        [11])
    dpy1 = sdpy1(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x
        [11])
    dpz1 = sdpz1(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x
        [11])
    dpx2 = sdpx2(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x
        [11])
    dpy2 = sdpy2(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x
        [11])
    dpz2 = sdpz2(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[10],x
        [11])
    return(np.array([dx1,dy1,dz1,dx2,dy2,dz2,dpx1,dpy1,dpz1,dpx2,dpy2,dpz2]))

# initial points
x0 = [0,20,5, 20,10,15, 0.0,0.1,0, 0.1,0.0,0.0]
# setting
t0 = 0#初期時間
T = 1000#最終時間
dt = 0.01#差分刻
times = np.arange(t0, T, dt)
args = (G, m1, m2)
orbit = odeint(hamilton_vectorfield, x0, times, args)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot(orbit[:, 0], orbit[:, 1], orbit[:, 2], '-b', linewidth=0.6)
ax.plot(orbit[:, 3], orbit[:, 4], orbit[:, 5], '-r', linewidth=0.6)
ax.set(xlabel='x(t)', ylabel='y(t)', zlabel='z(t)')
#fig.savefig('kepler1.png')
plt.show()
```

7.5 Fermi-Pasta-Ulam の格子振動

1950 年頃から Fermi はコンピュータによる数値計算によって非線形格子振動を調べていました。僅かな非線形性の存在が物質の熱拡散などの不可逆現象をもたらすだろうということは物理学者共通の信念がったからです。

固体比熱などは格子の調和振動モデルで良く説明されていますが、一方で熱拡散は物質

内部でのエネルギー交換・再配分の結果均質に広がっていくという理解は調和振動子では説明することができません。調和バネでつながった格子振動は基準座標で考えると基準モードは互いに独立でエネルギー交換が生じないからです。

直線状の格子点に並んだ質量 m を持つ $N+1$ 個の質点系において i 番目の質点の平衡位置からのの変位を x_i としたとき、$i+1$ 番目の質点の変位によって $x+1-x_i$ の復元力を受けるような運動を 1 次元調和格子振動といい、対応するハミルトニアンは

$$H = \sum_{k=1}^{N-1} \frac{1}{2m} {p_k}^2 + \frac{k}{2} \sum_{k=1}^{N-1} (x_{k+1} - x_k)^2, \quad x_0 = x_N = 0 \tag{7.13}$$

となります。ここでは格子の両端を固定 ($x_0 = x_N = 0$) して考えています。基準座標系 $\{P_j, Q_j\}$ を

$$x_k = \sqrt{\frac{2}{N}} \sum_{j=1}^{N-1} Q_j \sin \frac{jk\pi}{N}, \quad m\frac{dx}{dt} = P_j \tag{7.14}$$

によって導入すると、ハミルトニアンは

$$H = \sum_{j=1}^{N} \frac{1}{2m} ({P_j}^2 + m^2\omega_j^2 {Q_j}^2), \quad \omega_j = 2\sqrt{\frac{k}{m}} \sin \frac{i\pi}{N} \tag{7.15}$$

と書くことができます [25, §5.9]。ω_j を基準振動数、

$$H_k = \frac{1}{2m} ({P_k}^2 + m^2\omega_k^2 {Q_k}^2)$$

をモード k の基準エネルギーといいます。このように、基準座標系では調和振動子が互いに独立であってモード間で $\{H_k\}$ のエネルギー交換がないことがわかります。

そこで Fermi-Pasta-Ulam[31] は次のような 3 次のポテンシャルを含む非線形相互作用のある 1 次元格子振動を考えました。

$$H = \sum_{k=1}^{N-1} \frac{1}{2m} {p_k}^2 + \frac{k}{2} \sum_{k=1}^{N-1} (x_{k+1} - x_k)^2 + \frac{\alpha}{3} \sum_{k=1}^{N-1} (x_{k+1} - x_k)^3 \tag{7.16}$$

このとき、運動方程式は

$$m\frac{d{x_k}^2}{dt} = k(x_{k+1} - 2x_k + x_{k-1})(1 + \alpha(x_{k+1} - x_{k-1})) \tag{7.17}$$

と書くことができます。

図 7.5 はプログラム 7.5-1: `fpu_2.py` を使って、$\alpha = 1/4$ および初期条件として初速 0 で静止変位 $x_{j0} = \sin k\pi/N$ を与えたときの様子です。時間経過 $T = 4000$ までの様子を時間間隔 200 ごとに格子位置での変位を描きました。プログラム 7.5-1 ではハミルトンの正準方程式 (7.3) から得られるベクトル場を関数 `fpu_vector` で定義しています。格子上の変位は時間と共に不規則に変化したままなのでしょうか。

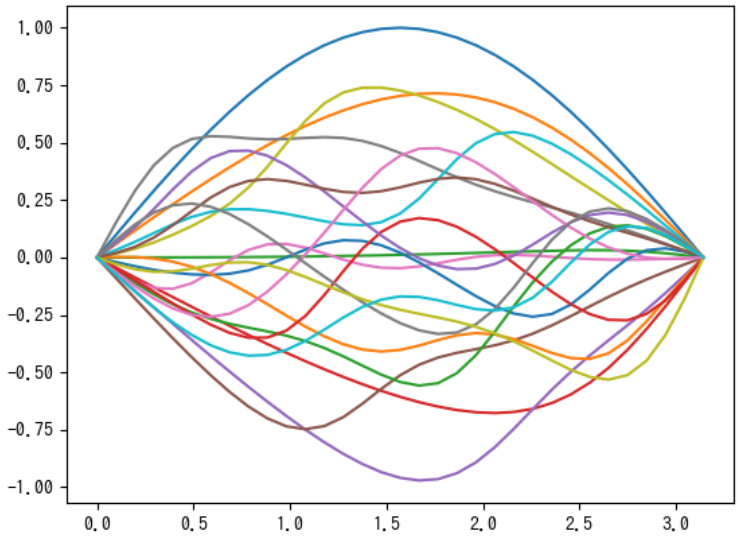


図 7.5　Fermi-Pastaz-Ulam のハミルトニアン 7.16（$\alpha = 1/4$）で定まる非線形格子振動の様子。初期条件を初速 0 で静止変位 $x_{j0} = \sin k\pi/N$ を与えて時間経過 $T = 4000$ まで時間間隔 200 ごとにプロット。

コード 7.5-1　fpu_2.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# N+1個の格子質点系のベクトル場
def fpu_vector(x: np.ndarray, t:float, alpha:float) -> np.ndarray:
        dx = np.empty(x.shape, dtype=float)
        dx[0] = 0                 # velocity of 1st point = 0
        for i in range(1, N): # velocity of 1st to (N-1)th = x[N+i+1]
                dx[i] = x[N+i+1]
        dx[N] = 0               # velocity Nth point = 0
        dx[N+1] = 0             # acceleration of 1st point = 0
        for i in range(1, N):# acceleration of 1st to N-1 =
                dx[N+i+1] = x[i+1] - 2 * x[i] + x[i-1] + alpha * ( (x[i+1] - x
                    [i])**2 - (x[i] - x[i-1])**2)
        dx[2*N + 1] = 0 # acceleration of N-th point = 0
        return(np.array(dx))

alpha = 0.25
N = 32 # 質点総数
```

```
# setting
t0 = 0#初期時間
T = 4000#最終時
dt = 0.01#時間間隔
times = np.arange(t0, T, dt)

# the initial conditions x0 = [position0, momentum0]
lattice = np.linspace(0, N, N+1) # [0,N]上の格子位置
position0 = np.sin(np.pi*lattice/N)
momentum0 = np.zeros(len(lattice)+1)
x0 = np.concatenate([position0, momentum0])
orbit = odeint(fpu_vector, x0, times, args=(alpha,))

fig, ax = plt.subplots()
step = 20000
for k in range(0,len(times), step):
##    if k in [0, 300, 500, 700]:
##        ax.plot(ticks, orbit[k, 0: N+1])
    ax.plot(lattice, orbit[k, 0: N+1])
##    ax.pause(0.01) # 一時停止
##    ax.cla() # クリア
ax.set_xticks(np.linspace(0, N, 5))
ax.set_xticklabels(['0','8','16','24','32'])
fig.show()
```

図 7.5 からはわかりませんが、Fermi たちは基準座標系のエネルギー値 $H_k = E_k$ に注目し、モードエネルギーは変化するもの初期値に何度も戻るという**再帰現象** (reccurence phenomena) を発見し、挙動はおおむね周期的となることを見出しました。つまり、非線形性を導入しただけではエネルギーはシステム全体に分配されて不可逆性を示すわけではないことがわかりました [24]。

ここで、Fermi-Pasta-Ulam の格子において 3 つの質点が周期境界で配置（$x_i = x_{i+3}$）されている場合を考えます。このとき、ハミルトニアン (7.16) は

$$H_3 = \frac{1}{2}({p_1}^2+{p_2}^2+{p_3}^2)+\frac{1}{2}((x_1-x_2)^2+(x_2-x_3)^2+(x_3-x_1)^2)+\frac{\alpha}{3}((x_1-x_2)^3+(x_2-x_3)^3+(x_3-x_1)^3) \tag{7.18}$$

となります。この 3 質点周期格子系の運動は、以下に見るように Hénon-Heiles の Hamilton

系（節 7.2）の運動に深く関係しています [26]。行列

$$A = \begin{bmatrix} 6^{-1/2} & 2^{-1/2} & 3^{-1/3} \\ -(2/3)^{1/2} & 0 & 3^{-1/2} \\ 6^{-1/2} & -2^{-1/2} & 3^{-1/2} \end{bmatrix}$$

で与えられる主軸変換

$$q_i = \sum_{j=1}^{3} A_{ij}\xi_j, \quad p_i = \sum_{j=1}^{3} A_{ij}\eta_j \tag{7.19}$$

によって変数 $\{\xi_i, \eta_i\}$ を導入します。このとき、ハミルトニアン (7.18) はこの座標系を使って

$$H_3 = \frac{1}{2}({\eta_1}^2 + {\eta_2}^2 + {\eta_3}^2 + 3{\xi_1}^2 + 3{\xi_2}^2) + \frac{3\alpha}{\sqrt{2}}\left(\xi_2{\xi_1}^2 - \frac{1}{3}{\xi_2}^3\right)$$

となります（この正準方程式から P_3 は定数になります)。さらに変換

$$\xi_1 = \frac{\sqrt{2}}{\alpha}x_1, \quad \xi_2 = \frac{\sqrt{2}}{\alpha}x_2, \quad t = \frac{1}{\sqrt{3}}\tau$$

をおこなうと、運動方程式は

$$\begin{aligned} \frac{dx_1}{d\tau} &= \frac{\alpha}{\sqrt{6}}\eta_1 \equiv p_1, & \frac{dp_1}{d\tau} &= -x_1 - 2x_1x_2, \\ \frac{dx_2}{d\tau} &= \frac{\alpha}{\sqrt{6}}\eta_2 \equiv p_2, & \frac{dp_2}{d\tau} &= -x_2 - {x_1}^2 + {x_2}^2 \end{aligned}$$

となりますが、これは Hénon-Heiles の Hamilton 系の運動方程式 (7.8) です。

Fermi たちが発見した再帰現象は数値計算上の誤差によって生じているわけではありません。その後も探求が続き、1960 年代にソリトン解の数値的発見 [48] に続いて戸田格子の発見 [26] があり積分可能な可解系へと研究へと発展していきます。

7.6 戸田格子

Fermi-Pasta-Ulam の格子振動（節 7.5）では、非線形性の導入は必ずしも挙動の乱雑性をもたらすわけでないことを見ました。格子質点間のポテンシャルが

$$\phi(r) = \frac{a}{b}\mathrm{e}^{-br} + ar \qquad ab > 0 \tag{7.20}$$

であるような格子を**戸田格子** (Toda lattice) といいます [26]。ここで、r は、格子間の変位の大きさを表します。$\phi(r)$ を展開すると

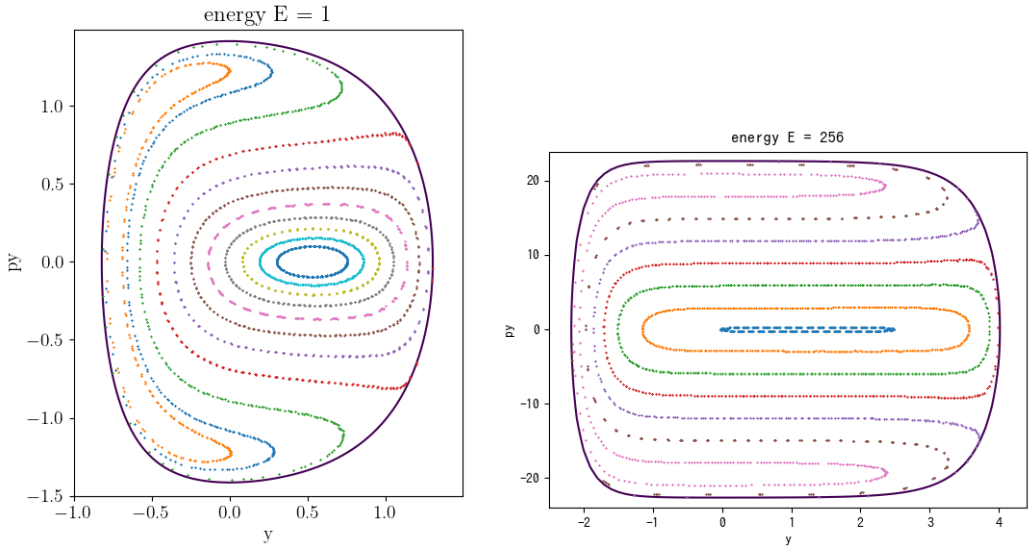
$$\phi(r) = \text{定数} + \frac{ab}{2}r^2 - \frac{ab^2}{6}r^3 + \dots$$

となり、r が小さい微小振動の場合にはバネ定数 $k = ab$ の調和格子になっています。

3 質点が周期配置された戸田格子振動のハミルトニアンは

$$H = \frac{1}{2}(p_1^2 + + p_2^2 + p_3^2) + \mathrm{e}^{-(q_2-q_1)} + \mathrm{e}^{-(q_3-q_2)} + \mathrm{e}^{-(q_1-q_3)} - 3 \tag{7.21}$$

となります。ここで、最終項の定数として運動しない（$q_i =_i= 0$）ときにエネルギーが 0 になるようにとりました。ポテンシャルを展開して 3 次までとると、Fermi-Pasta-Ulam の格子 (7.16)（非線形格子のパラメータ $\alpha = 1/2$）のハミルトニアン (7.18) になります。



(a) $E = 1$　　(b) $E = 256$

図 7.6　戸田格子の 3 粒子周期格子と同等なハミルトン力学系 (7.23) の複数軌道のポアンカレ断面。エネルギー値 $E = 1$ および $E = 256$ のエネルギー曲面上にある軌道の平面 $x = 0$ を $dx/dt > 0$ の方向で横切るポアンカレ断面上の点列はすべて不変閉曲線を成す線上にあり、積分可能性を示している [33]。

ハミルトニアン (7.21) の運動方程式を変換 (7.19) によって変数 $\{Q_i, P_i\}$ で書き換えると

$$\begin{aligned}
\frac{dQ_1}{d\tau} &= P_1, \quad \frac{dP_1}{d\tau} = \frac{1}{4\sqrt{3}}\left(-\mathrm{e}^{2Q_2+2\sqrt{3}Q_1} + \mathrm{e}^{2Q_2-2\sqrt{3}Q_1}\right) \\
\frac{dQ_2}{d\tau} &= P_2, \quad \frac{dP_2}{d\tau} = \frac{1}{6}\mathrm{e}^{-4Q_2} - \frac{1}{12}\left(\mathrm{e}^{2Q_2+2\sqrt{3}Q_1} + \mathrm{e}^{2Q_2}\right)
\end{aligned} \tag{7.22}$$

となります。この方程式に相当する式 (7.21) と同等な自由度 2 のハミルトニアンは

$$H = \frac{1}{2}({P_1}^2 + P_2 2) + \frac{1}{24}\left(\mathrm{e}^{2P_2+2\sqrt{3}P_1} + \mathrm{e}^{2P_2-2\sqrt{3}P_1} + \mathrm{e}^{-4P_2}\right) - \frac{1}{8} \tag{7.23}$$

となります。指数関数を 3 次の項までとると Hénon-Heiles のハミルトニアン (7.7) が得られることに注意してください。

図 7.6 は、プログラム 7.2-1 と同様にして、ハミルトニアン (7.23) が定める軌道の（平面 $x = 0$ を $dx/dt > 0$ の方向で横切る）ポアンカレ断面上の点列をプロットしたものです。エネルギー値 $E = 1$ および $E = 256$ を持つ複数の軌道から得られる点列はすべて不変閉曲線上にあることが観察できます。この Ford たち [33] の数値実験 (1973 年）後に Hénon は n の質点からなる戸田格子は積分可能であることを示しました [36]。

戸田 [26] は戸田格子の研究を推し進め、戸田格子が厳密な周期解およびソリトン解を持つことを発見し、非線形数理の世界の豊穣な世界への扉を開けました。

第 8 章　KdV 方程式を解く

KdV 方程式は偏微分方程式ですが、Fermi-Pasta-Ulam の実験で格子振動が示した再帰現象とも関わりがあり、可解系（積分可能系）という大きな数学の扉を開いた意義深い微分方程式です。非線形現象の解明において数多くの数学的な困難故にコンピュータによる数値実験が探求されてきましたが、多くの試みから理論的解明の方向性が示され新しい数理が誕生した良い例になっています。

8.1　KdV 方程式

Korteweg-de Vries(KdV) 方程式は時間と空間に関する発展方程式の 1 つで、$u(t,x)$ を時刻 t と 1 次元上の空間位置 x における波の高さ（波形）とした次の偏微分方程式で与えられます。

$$\frac{\partial u}{\partial t} + 6u\frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} = 0 \tag{8.1}$$

1965 年の Zabusky and Kruskal[48] は KdV 方程式がソリトン解を持つことを数値的に発見し、さらに 1967 年には Gardner, Greene, Kruskal, and Miura によってソリトン解を有する KdV 方程式の初期値問題が逆散乱法によって解けることが示されるなど、KdV 方程式は可積分系研究の新たな扉を開けることになった非線形偏微分方程式です [27]。

KdV 方程式の与え方には式 8.1 以外にもさまざまの流儀があり、変換 $u \to -u$ によって $u_t - 6uu_x + u_{xxx} = 0$、$u \to u/6$ によって $u_t + uu_x + u_{xxx} = 0$ としたり、また Zabusky and Kruskal が考えた式 (8.8) などとして考えることができます。KdV 方程式の一般型は

$$\frac{\partial u}{\partial t} + \alpha u\frac{\partial u}{\partial x} + \beta\frac{\partial^3 u}{\partial x^3} = 0 \tag{8.1$'$}$$

で表されます。

波 $u(x,t)$ がその形 f を変えないで伝搬するとき、定常波（stationary wave）または進行波といい、ある関数 f を使って

$$u(t,x) = f(x - ct)$$

と表されます。c は波の伝搬速度です。KdV 方程式がその特殊解として、進行波

$$\begin{aligned} u(t,x) &= \frac{c}{2}\mathrm{sech}^2\frac{\sqrt{c}}{2}(x-ct+\delta) \\ &= 2\kappa^2\mathrm{sech}^2\kappa(x-ct+\delta), \quad c=4\kappa^2 \end{aligned} \tag{8.2}$$

を持つことは、式 (8.1) に直接代入して式て確認することができます。ここで、sech は双曲線余弦関数 cosh の逆数

$$\mathrm{sech}\, x = \frac{1}{\cosh x} = \frac{2}{e^x + e^{-x}}$$

です。

特殊解 8.2 は 1 つの山を持ち、遠方で 0 になる波になっています。こうした形状を持つ波を**孤立波** (solitary wave) といいます。κ^{-1} は孤立波の幅の目安を与え、κ が大きいほど孤立波の幅が狭く波形は 2κ と高く、伝搬速度 c は大きくなります。

8.2 KdV 方程式の導出

KdV 方程式は、Fermi-Pasta-Ulam の格子振動 (7.17) に関係しています。格子粒子間の間隔に比べて長い波長を考えるた場合の**連続体近似** (Continuum approximation) から導出することができます。

連続体近似は格子間の自然長を h とすると k 番目の格子位置 $x = kh$ を連続変数とみなすことに基づきます。運動方程式 (7.17) の解 $u_n(t)$ において、位置と時間の関数 $u_n(t) = u(x,t)$ としてテーラー展開

$$u_{n\pm1} = u \pm h\frac{\partial u}{\partial x} + \frac{h^2}{2!}\frac{\partial^2 u}{\partial x^2} \pm \frac{h^3}{3!}\frac{\partial^3 u}{\partial x^3} + \frac{h^4}{4!}\frac{\partial^4 u}{\partial x^4} \pm \cdots \equiv \mathrm{e}^{\pm h\partial_x}u \tag{8.3}$$

が成立すると考えられます。ここで、演算子を $\mathrm{e}^{\pm h\partial_x} \equiv \sum_{k=0}^{\infty}\frac{1}{k!}\frac{\partial^k}{\partial x^k}$ としています。以下では

$$u_t = \frac{\partial u}{\partial t},\ u_x = \frac{\partial u}{\partial x},\ u_{xx} = \frac{\partial^2 u}{\partial x^2},\ u_{xxx} = \frac{\partial^3 u}{\partial x^3}$$

などと略記します。式 (8.3) を運動方程式 (7.17) に代入すると

$$mu_{tt} = kh^2\left(u_{xx} + \frac{h^2}{12}u_{xxxx} + \frac{h^4}{360}u_{xxxxxx} + \cdots + 2\alpha h u_x u_{xx} + \frac{h^3}{3}\alpha u_{xx}u_{xxx} + \frac{h^3}{3}\alpha u_x u_{xxxx} + \ldots\right) \tag{8.4}$$

となります。

非線形パラメータ $\alpha = 0$ のときには、波長が十分長い（h は十分小さい）と考えて $\mathcal{O}(h)$ 以上を無視した 0 次近似を取ると、波動方程式

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, \quad c = h\sqrt{k/m}$$

が得られます。

さて、変数 x, t を ξ, τ に取り替えるガードナー = モリカワ変換

$$\xi = \epsilon^p \frac{x - ct}{\ell}, \quad \tau = \epsilon^q \frac{ct}{\ell} \tag{8.5}$$

を考えて，進行波解

$$u(x,t) = \epsilon^r u^{(1)}(\xi, \tau) \tag{8.6}$$

がみたすべきパラメータ ϵ の条件を考えてみます。式 8.5 から

$$\frac{\partial}{\partial t} = -\frac{\epsilon^p c}{\ell}\frac{\partial}{\partial \xi} + \frac{\epsilon^q c}{\ell}\frac{\partial}{\partial \tau}, \quad \frac{\partial}{\partial x} = \frac{\epsilon^p}{\ell}\frac{\partial}{\partial \xi}$$

を式 8.4 に代入します。ϵ に関して $p + q (q > p)$ より高い次数の項を無視すると 3 項が残り、これらが同じ次数であるとすると

$$p + q = 4p = 3p + r$$

が成立しなければならず，$r = \frac{1}{2}$ を選ぶと $p = \frac{1}{2}, q = \frac{3}{2}$ で

$$2u^{(1)}_{\xi\tau} + \frac{1}{12}u^{(1)}_{\xi\xi\xi\xi} + 2\alpha u^{(1)}_{\xi} u^{(1)}_{\xi\xi} = 0$$

となります。これより、$u = \alpha u^{(1)}_{\xi}$ とおくと

$$\frac{\partial u}{\partial \tau} + u\frac{\partial u}{\partial \xi} + \frac{1}{24}\frac{\partial^3 u}{\partial \xi^3} = 0 \tag{8.7}$$

が得られ [27][16]、KdV 方程式の一般型 (8.1′) になることがわかりました。

8.3 Zabusky-Kruskal のアルゴリズム

偏微分方程式において、初期時刻 $t = t_0$ での初期境界条件 $u(t_0, x)$ のもとで解 $u(t, x)$ を決定すること（そもそも解が存在するか）が初期値問題です。偏微分方程式の数値解法は、時間刻み幅 Δt と空間刻み幅 Δx を使って、$u(t, x)$ の離散時空間上の値 $\{u(j\Delta t, i\Delta x)\} = \{u^j_i\}$ を差分スキーム

$$u^{j+1}_i = \mathcal{F}\left[\left\{u^j_k\right\}_k, \left\{u^{j-1}_k\right\}_k, \left\{u^{j-2}_k\right\}_k, \ldots, \left\{u^{j-\ell}_k\right\}_k, \left\{u^0_k\right\}_k\right]$$

を使って、逐次的に求めます。$u(t,x)$ が 1 次元であっても、各離散時間 j の値として離散空間上のすべての値 $u_0^j, u_1^j, \ldots, u_{N-1}^j$ を計算する必要があります。

偏微分方程式の数値解法は、誤差評価や数値安定性において明らかに常微分方程式以上に格段に困難であり、時間だけでなく空間的離散的刻みが必要となるために計算量的にも大量の計算が必要となるため、今日における数値計算上の主要な研究テーマの 1 つとなっています。

Zabusky and Kruskal は KdV 方程式として

$$u_t + uu_x + \delta^2 u_{xxx} = 0, \qquad \delta = 0.022 \tag{8.8}$$

を、周期境界条件 $u(t,x) = u(t,x+L)$ のもとで（$L=2$)、時刻 $t=0$ での初期境界値

$$u(0,x) = \cos \pi x$$

の問題を数値的に解析し、粒子のように振る舞う孤立波の振る舞いとみなせる現象を発見しました [48]。波 $u(t,x)$ の初期値 $u(0,x)$ を $\cos pix$ で与えた波が時間経過と共に高さの異なる波列に分解して進行し、それらが互いに衝突し合った後に再び元の姿を回復して、あたかも衝突をすり抜けたかのように見えることが観察されたのです（図 8.2、[48, Fig.2]）。このため、そのような特性もつ波を粒子的であることを意味する–on をつけて**ソリトン**（soliton）と呼んでいます。今日では、KdV 方程式は N-soliton 解を持つことが分かっています。

Zabusky-Kruskal は、KdV 方程式 (8.8) を初期境界 $u(0,\cos \pi x)$ のもとで数値計算するために、空間区間 $[0,2]$ を周期境界条件 $u(t,x) = u(t,x+2)$ を仮定し時刻 j における 1 次元区間を N 個に等分割した離散点を $\{u_k^j\}_{k=0,\ldots,N-1}$ として、次の差分アルゴリズムを使いました [48, 論文末尾の注 (6)]。

$$\begin{aligned} u_i^{j+1} &= u_i^{j-1} + \mathcal{Z}\mathcal{K}(\left\{u_k^j\right\}_k) \\ &= u_i^{j-1} - \frac{\Delta t}{3\Delta x}\left(u_{i+1}^j + u_i^j + u_{i-1}^j\right)\left(u_{i+1}^j - u_{i-1}^j\right) - \delta^2 \frac{\Delta t}{\Delta x^3}\left(u_{i+2}^j - 2u_{i+1}^j + 2u_{i-1}^j - u_{i-2}^j\right) \end{aligned} \tag{8.9}$$

$$u_i^j = u_{i+N}^j \qquad \text{（周期境界条件）}$$

このアルゴリズムでは u_i^{j+1} を求めるために、$\{u_k^j\}$ だけでなく u_i^{j-1} の値も必要になります。このため、初期時間 $j=0$（$t=0\Delta t$） での初期境界条件 $\{u_i^0\}_{i=0,\ldots,N-1}$ から次の時間 $j=1$（$t=\Delta t$）での値 $\{u_i^1\}_{i=0,\ldots,N-1}$ を次のスキームを使って用意しておくことにします。

$$u_i^1 = u_i^0 - \frac{\Delta t}{6\Delta x}\left(u_{i+1}^0 + u_i^0 + u_{i-1}^0\right)\left(u_{i+1}^0 - u_{i-1}^0\right) - \frac{\Delta t \delta^3}{2\Delta x^3}\left(u_{i+2}^0 - 2u_{i+1}^0 + 2u_{i-1}^0 - u_{i-2}^0\right) \tag{8.9′}$$

差分アルゴリズム (8.9) と (8.9′) から、時刻 $j+1$ での 1 次元配列 u_i^{j+1} を計算するために、時刻 j での 1 次元配列 $\{u^j\}$ と時刻 $j-1$ での 1 次元配列 $\{u^{j-1}\}$ を使って次のように逐次的に求めていきます。

$$
\begin{aligned}
u_i^2 &= u_i^0 + \mathcal{ZK}(\left\{u_k^1\right\}_k) \\
u_i^3 &= u_i^1 + \mathcal{ZK}(\left\{u_k^2\right\}_k) \\
&\vdots \\
u^{j+1} &= u_i^{j-1} + \mathcal{ZK}(\left\{u_k^j\right\}_k)
\end{aligned}
$$

8.4 Zabusky-Kruskal の差分アルゴリズム計算

KdV 方程式を数値的に解く Zabusky-Kruskal の差分アルゴリズム (8.9) を効率的に計算する方法を考えましょう。アルゴリズム (8.9) では、時刻 $j+1$ の 1 次元配列 u^{j+1} の位置 i での成分 u_i^{j+1} を、時刻 $j-1$ と j での配列 u^{j-1} と u^j を使った $\left(u_{i+1}^j + u_i^j + u_{i-1}^j\right)\left(u_{i+1}^j - u_{i-1}^j\right)$ および $\left(u_{i+2}^j - 2u_{i+1}^j + 2u_{i-1}^j - u_{i-2}^j\right)$ の計算が必要です。初期時間 $j=0$ で与えた 1 次元配列 u^0 から次の時間 $j=1$ の配列 u^1 の計算 (8.9′) でも同様です。

これらの計算において、位置 i 成分に関して周期境界条件 $u_i^j = u_{i+N}^j$ を満たすように実行する必要があります。

8.4.1 パディング `numpy.pad`

周期境界条件から、配列要素 $\{u_i^j\}$ が位置 i について次のように円環的に配置

$$\ldots, u_{N-2}^j, u_{N-1}^j, u_0^j, u_1^j, u_2^j, \ldots, u_{N-2}^j, u_{N-1}^j, u_0^j, u_1^j, u_2^j, \ldots$$

されていると考え、配列 $\{u_i^j\}$ は位置成分について、

$$u_{-1}^j = u_{N-1}^j,\ u_{-2}^j = u_{N-2}^j \quad \text{および} \quad u_N^j = u_0^j,\ u_{N+1}^j = u_1^j$$

が成り立ちます。たとえば $\left(u_{i+2}^j - 2u_{i+1}^j + 2u_{i-1}^j - u_{i-2}^j\right)$ を計算するには、位置 $i=0$ のときには $\left(u_2^j - 2u_1^j + 2u_{-1}^j - u_{-2}^j\right)$ を、位置 $i=N-1$ のときには $\left(u_{N+1}^j - 2u_N^j + 2u_{N-2}^j - u_{N-3}^j\right)$ を計算する必要があります。

NumPy には、与えた配列 `u` の並びに前後に詰め物幅を指定することで配列を拡大（**パディング**）するための NumPy 関数 `numpy.pad` があります。目的とするパディングを達成されるようにパディングする際にモード指定が用意されていてされます。今の場合、一

次元配列を周期境界条件を満たすよう前後にパディングするためのモード'wrap' を使います。

次では、与えた 1 次元リスト a の前に 2 つ及び後ろに 3 つの要素をパディングするために詰め物幅 (2, 3) を指定して周期境界を満たすようにしています。

```
In [1]: import numpy as np

In [2]: a = [1, 2, 3, 4, 5]

In [3]: np.pad(a, (2, 3), 'wrap')
Out[3]:
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
```

8.4.2 たたみこみ numpy.convolve

1 次元配列 a と v が与えられているとき、

$$(a * v)[n] = \sum_{k=-\infty}^{\infty} a[k]v[n-k] \tag{8.10}$$

を**たたみ込み** (convolution) といいます。

NumPy 関数 numpy.convolve(a, v, mod) は a と v のたたみ込みを計算します。畳み込み計算での総和計算において負インデックスを持つ配列要素の取り扱いを調整する必要があります。このために、オプション mode として'full'（デフォルト）, 'same' および 'valid' の 3 つが用意されています。

配列 a の長さを $|a| = N$、配列 v の長さを $M = |v|$ とします。mode = 'full' は、v の有効インデックス全体で a の有効インデックスがある限りを使って総和し、長さは $N + M - 1$ の配列を返します。mode = 'same' は、v の有効インデックス全体で a の有効インデックスだけを使って総和し、長さは $\max(N, M)$ の配列を返します。mode = 'valid' は、a と v の有効インデックスだけを使いインデックス範囲が重ならない計算をせずに総和し、長さ $|a - v| + 1$ の配列を返します。図 8.1 にその差異を示しました。

たとえば次の例を観察してください。

```
In [4]: a = np.array([0, 1, 2, 3, 4, 5])

In [5]: v = np.array([0.2, 0.6])

In [6]: np.convolve(a, v, 'full')
Out[6]: array([0. , 0.2, 1. , 1.8, 2.6, 3.4, 3. ])
```

```
In [7]: np.convolve(a, v, 'same')
Out[7]: array([0. , 0.2, 1. , 1.8, 2.6, 3.4])

In [8]: np.convolve(a, v, 'full')
Out[8]: array([0. , 0.2, 1. , 1.8, 2.6, 3.4, 3. ])
```

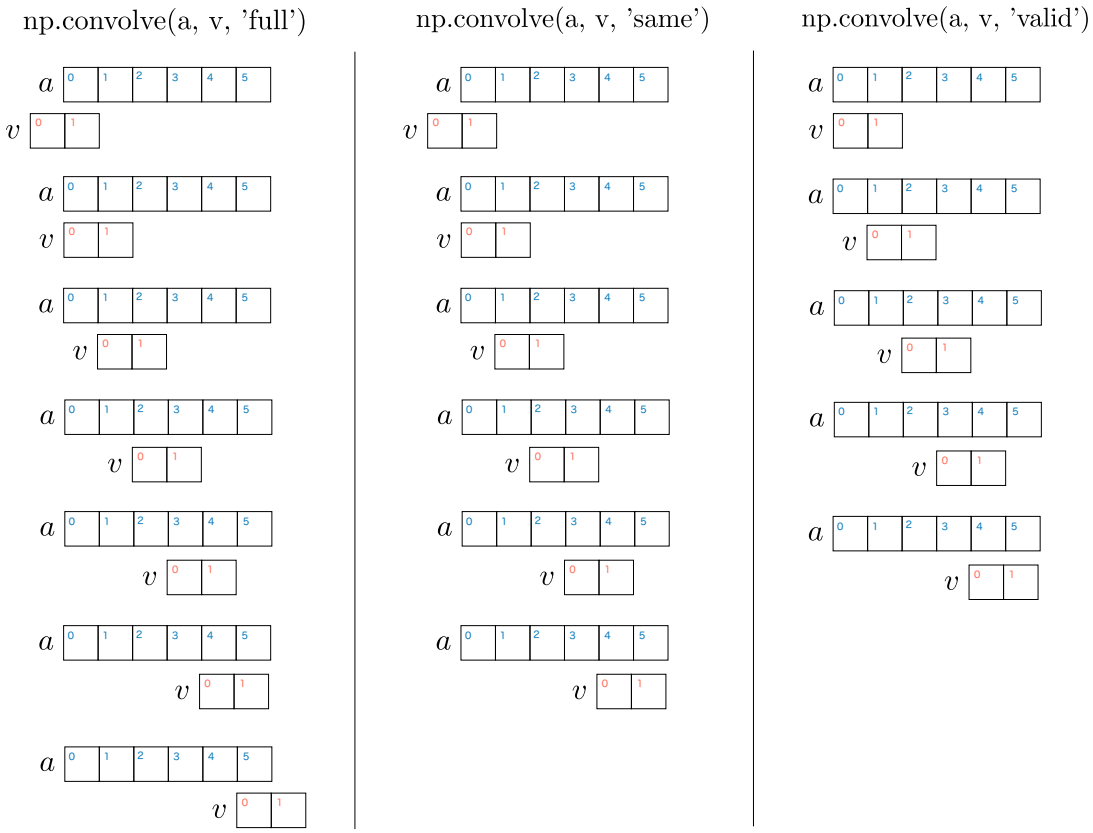


図 8.1　長さが 6 と 2 の 1 次元配列 a と v のたたみ込み `np.convolve(a, v, mode)` の様子。配列をずらしながら重なっている要素ごとを掛けて総和していく。`mode` の違い `'full'`, `'same'` および `'valid'` によって計算対象となる条件が異なる。

8.4.3　Zabusky-Kruskal アルゴリズムの計算

Zabusky-Kruskal アルゴリズムを NumPy の padding や convolve を使って効率化することができます。周期境界条件を満たす 1 次元配列 $\{u_i^j\}$ から、たとえば $\left(u_{i+2}^j - 2u_{i+1}^j + 2u_{i-1}^j - u_{i-2}^j\right)$ を計算するために、周期境界を満たすようにパディングによって配列を拡大したうえで、たたみ込みの計算を行います。

このとき必要になるのは、配列 $\{u_i^j\}$ の前後にパディングするために必要な詰め込み幅を決定してパディングして得られる配列 `extension` を、目的とする畳み込みに必要なパターン配列 `rangelist` を与えてたたみこみ `np.convolve(extension, rangelist, 'valid')` を計算することです。

次の関数 `moving_sum(u, pattern)` は、1 次元配列 `u` にたたみ込みパターンリスト `pattern` を周期境界条件を満たすように `u` を拡大して、指定した倍率で掛け算した連続和を計算します。ここでは、Zabusky-Kruskal アルゴリズム (8.9) において位置 i 成分に対して、$\left(u_{i+2}^j - 2u_{i+1}^j + 2u_{i-1}^j - u_{i-2}^j\right)$ ではパターンリスト $[1, -2, \check{0}, 2, 1]$、$\left(u_{i+1}^j - u_{i-1}^j\right)$ ではパターンリスト $[1, \check{0}, -1]$ のように位置 $\check{i}$ に関して対称にパターン指定することを念頭に置いています。

```
def moving_sum(u: np.ndarray, pattern: list) -> np.ndarray:
    rangelist = np.asarray(pattern)
    mid = rangelist.size // 2
    extension = np.pad(u, (rangelist.size-mid-1, mid), 'wrap')
    return(np.convolve(extension, rangelist, 'valid'))
```

引数で指定したパターンリスト `pattern` を `np.asarray(pattern)` で NumPy 配列化して `rangelist` とおいて、その長さを 2 で割った整数商 `mid` を計算します。配列 `u` をパディングする際に前後詰め幅を $(\text{rangelist.size} - \text{mid} - 1, \text{mid})$ と位置 i からの長さを使って決定し、モード指定'`wrap`' で周期境界条件を満たすように `u` を `extension` に拡張したうえで、パターン NumPy 配列 `rangelist` とでたたみ込みをします。

こうして関数 `moving_sum` を使い、初期時間での初期境界波形 `u0` とその単位時間後の波形 `u1` をスキーム (8.9′) で計算した後に、以降の波形を逐次的に計算するプログラムは次のようになります。

```
times = np.arange(dt, 2.6, dt)
dx = L / N
u0 = np.cos(np.pi * x)
u1 = u0 - dt/(6*dx) * moving_sum(u0, [1, 1, 1]) * moving_sum(u0, [1, 0, -1]) \
    - dt * (delta ** 2) / (2 * dx ** 3) * moving_sum(u0, [1, -2, 0, 2, -1])
...
for t in times:
    u0, u1 = u1, u0
    u1 -=  dt/(3*dx) * moving_sum(u0, [1, 1, 1]) * moving_sum(u0, [1, 0, -1]) \
          + dt * (delta ** 2) /(dx ** 3) *moving_sum(u0, [1, -2, 0, 2, -1])
```

8.5　KdV 方程式の数値解

いままでの議論をまとめたプログラム 8.5-1: zk_algorithm.py は、Zabusky-Kruskal アルゴリズム (8.9) を使って KdV 方程式 (8.8) を初期時刻の初期波形 $\cos \pi x$ を周期境界条件のもとで計算し、指定した時間（ここでは $t = 0, t = 1/\pi, t = 3.6/\pi$ および $t = 8.1/\pi$）における波形をプロットします（図 8.2 参照)。

コード 8.5-1　zk_algorithm.py

```
import numpy as np
import matplotlib.pyplot as plt

def find_times(times, time_points):
#   times内の指定点列time_points=[1, 2, 3]の近接点列を抽出
    tms = np.asarray(times)
    idx = [np.abs(tms-t).argmin() for t in time_points]
    return(tms[idx])

def moving_sum(u: np.ndarray, pattern: list) -> np.ndarray:
#   点列u[i]におけるrelativelist = [1, 0, -1]に関する移動総和
    rangelist = np.asarray(pattern)
    mid = rangelist.size // 2
    extension = np.pad(u, (rangelist.size-mid-1, mid), 'wrap')
    return(np.convolve(extension, rangelist, 'valid'))

delta = 0.022 # KdVの係数
L = 2.0 # 周期境界区間の長さ
N = 256 # 空間格子の数
dx = L / N # 空間刻み幅
dt = 0.0002 # 1.0 / N 時間刻み幅
times = np.arange(dt, 2.6, dt) # 離散時間列

fig, ax = plt.subplots()
# initial condition u(x=i,t=0), i=0,...,N-1
x = np.linspace(0, 2, N, endpoint = False)
u0 = np.cos(np.pi * x) # 初期境界条件
ax.plot(x, u0) # 初期波形プロット
# Zabusky-Kruskalのアルゴリズム(0)
```

```
u1 = u0 - dt/(6*dx) * moving_sum(u0, [1, 1, 1]) * moving_sum(u0, [1, 0, -1]) \
    - dt * (delta ** 2) / (2 * dx ** 3) * moving_sum(u0, [1, -2, 0, 2, -1])

findedtimes = find_times(times, [1/np.pi, 3.6/np.pi, 8.1/np.pi])
# Zabusky-Kruskalのアルゴリズム(1)
for t in times:
    u0, u1 = u1, u0
    u1 -=  dt/(3*dx) * moving_sum(u0, [1, 1, 1]) * moving_sum(u0, [1, 0, -1])
        \
          + dt * (delta ** 2) /(dx ** 3) *moving_sum(u0, [1, -2, 0, 2, -1])
    if t in findedtimes: # 途中時間で波形プロット
        ax.plot(x, u1)

ax.set(xlim=[0, 2], ylim =[-1, 2.7],
       xlabel='Normalized Distance x', ylabel='Wave form u(x)')
ax.legend(["(A) $t=0$", "(B) $t=1/\pi$", "(C) $t=3.6/\pi$", "(D) $t=8.1/\pi$
    "])
#fig.savefig('zk_algorithm.png')
fig.show()
```

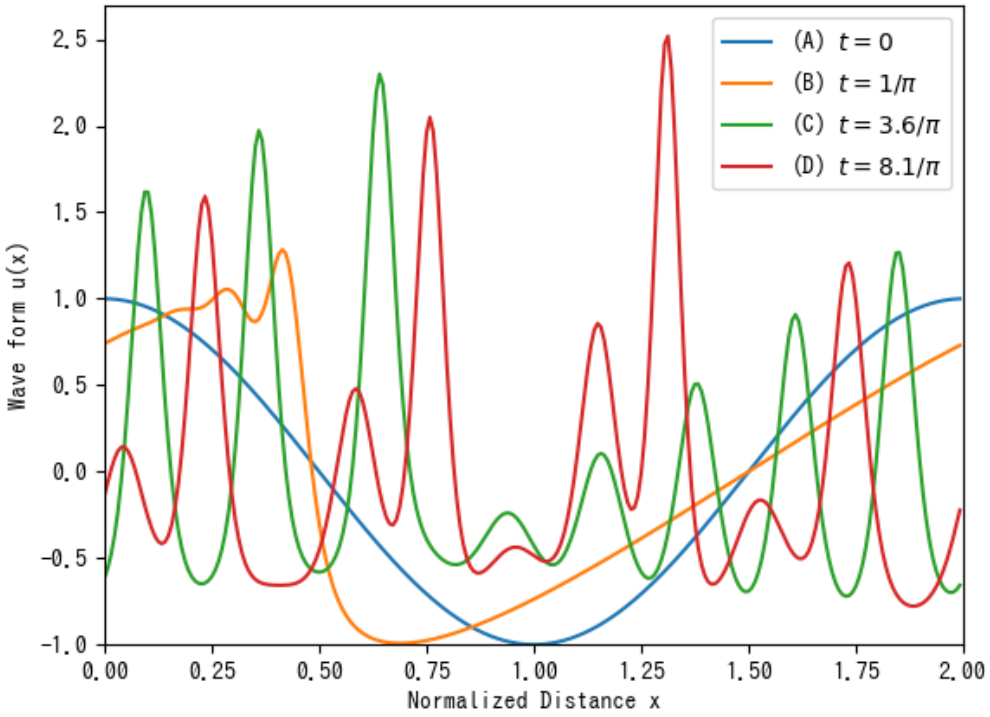


図 8.2　KdV 方程式 (8.8)($\delta = 0.022$) を周期境界条件 $u(t,x) = u(t,x+2)$ のもとで初期境界波形 $u(0,x) = \cos \pi x$ に対して Zabusky-Kruskal の差分アルゴリズム (8.9) に従って計算した波形変化。4 つの曲線は時間 $t = 0$, $t = 1/\pi$, $t = 3.6/\pi$ および $t = 8.1/\pi$ での波形 $u(t,x)$ を表している。時刻 $t = 0$, $t = 1/\pi$, $t = 3.6/\pi$ での波形プロットは論文 [48] の FIG.1 と同じである。

Zabusky-Kruskal[48] の報告にあるように、時刻 $t = 0$ で与えた初期波形 $u(0,x) = \cos \pi x$（図 8.2 曲線 (A)）は徐々に形を変えて時刻 $t = 1/\pi = T_B$ で位置 $x = 1/2$ において波の立ち上がり効果をもたらす非線形項 uu_x のために突っ立つようになります。$x < 1/2$ にあるわずかな振動は波の広がりの効果を表す分散項 u_{xxx} によるものです（図 8.2 曲線 (B)）。$\delta = 0.022$ としたとき、最初は $\max|\delta^2 u_{xxx}| / \max|uu_x| = 0.004$ 程度なので分散項を無視でき、波形 u の発展は Burgers 方程式 $u_t + uu_x = 0$ で近似できます。この解は再帰関係式 $u = \cos \pi(x - ut)$ を満たしますが、$x = 1/2$ および $t = 1/\pi \equiv T_B$ で不連続になってしまいます。

時間とともに振動波形が成長し、$t = 3.6T_B$ になると孤立波列が波高の高い方から降順に 8 個（$x = 1/6$ あたりから左側に 1,2,3 個、周期境界条件によって右端から 4,5,6,7,8）と並びます [48, Fig.1]（図 8.2 曲線 (C)）。これらの孤立波は波高の高さに比例した速度で右側に周期境界条件を満たすように移動するのですが、時刻 $t = 8.1T_B$ では各孤立は衝突後

にすり抜けるようにして元の波形を保っていることがわかります（図 8.2 曲線 (D)）。

Zabusky and Kruskal は詳細な数値計算によって、KdV 方程式の孤立波、相互の衝突によってもする抜けるようにしてその形を変えず「粒子」であるかのように振る舞うことを突き止め [48, FIG.2]、このような孤立波をソリトンと名付けたのです。

付録 A　Python の利用環境

近年、プログラミング環境は大きく改善され、しかも多くの良質なインターネット情報の提供されるようになって、誰でも無料で目的に応じたプログラム開発環境に触れることができます。

とりあえず必要とするプログラミング環境さえ整え使い慣れていくようになれば、ユーザにとって望ましいプログラム環境がどのようなものかを理解できるようになります。しかしながら、そもそも初心者にとって自分が必要とするプログラム環境をどう整えればよいのかが不明であり、このことが障壁となって次に進むことを難しくしてしまう場合が少なくありません[1)]。

本章はそのようなユーザのために Python のプログラム環境にはどのようなものがあり、どうしたら使えるようになるのかを紹介します。Python 環境のパソコンへのインストール法は改善され、簡単に Python 環境を整えることができるようになりました。また、クラウド利用の拡大と共にプログラム環境のクラウド化も進行し、節 A.4 で紹介する Google Colaboratory(Colab) のように、パソコンへのインストールを経ることなく Web ブラウザを経由して先端的な統合開発環境（クラウド IDE）を利用することもできます。

A.1　Python の利用環境

Python の利用環境には大別すると次のように自分のパソコン利用かクラウド利用かの 2 つがあります。

(1)手元のパソコンに Python 環境を整えてプログラムを実行する
- Python コマンドを使う
- IPython を使う
- Jupyter を使う

(2)クラウドサービスを利用してプログラムを（クラウド上で）実行する
- Google Colaboratry を使う

Python プログラミングでは Python コミュニティが積み上げてきたソフトウエア資産を活

1)　学校の教室に設置してあるパソコンであれば、管理者が前もってとりあえず必要とするプログラミング環境を整え、初心者は教えられたとおりに操作してプログラムを実行することができます

用することが重要なポイントです。Python に興味があるけれども、パソコン操作、特にコンソール（ターミナルウィンドウ[2)] にキー入力する操作）に詳しくないユーザや、どのようにパソコンでの利用環境を整えればよいか（実際、Python 環境構築のためにどんな方法がよいのか諸説あるようです）迷っているユーザは、まずは節 A.4 で紹介するクラウド経由でプログラミング環境提供するクラウド IDE として Google Colaboratory(Colab) に使ってみてはいかがでしょうか。手っ取り早く Python プログラムを書いて実行させることができます。ただし、どのアプローチを選んでも、どこにファイルを保存するかやファイルの読み込みなどファイル操作コマンドを含むファイル管理に関する多少のメタ知識は必要になります（経験を積むことで使い慣れていくはずです）。

パソコンにどの Python インストーラを利用して環境を整えるかを選択し、必要とする Python パッケージのインストールやバージョンアップには自分で対応しなければなりません（本書でが積極的に Python 拡張ライブラリを利用します）。その代わりに、好みのテキストエディタを使うなどして効率的な開発環境を追求することが可能になります。

一方、Google Colaboratory のようなクラウド環境を利用する場合、クラウドサービスを受けるためのアカウントを取得し（本書の範囲では無料利用で十分です）、クラウドドライブ上のファイルと手元のパソコンのファイルとの関係を正確に理解しておく必要があります。その代わりに、常に最新の環境のもとで複数の端末から同じようにプログラム開発ができます（パソコンだけでなくタブレットからでも可能です）。

A.2　パソコンに Python 環境をインストールする

Windows や macOS などのパソコンに Python 環境を整える方法を調べてみると、多数の選択肢があることがわかります。いずれも Python ソフトウエア資産が活用できるようにパソコンの Python 環境を整える必要があります。

A.2.1　インストール情報

Python イントールと関連ソフトウエア資産の取得には 2 種類があり、Python 公式版の配布元 Python Software Foundation（PSF: `https://www.python.org`）が運営する PyPI(Python Package Index: `https://pypi.org`）を利用する方法で、ソフトウェアパッケージ取得ツール `pip` を使います。もう一方は、科学技術計算のプラットホームサービスである Anaconda（`https://www.anaconda.com`）を利用する方法で、ソフトウェアパッケージ取得ツール `conda` を使います（Anaconda が非公式版 Python というわけではあ

2)　Windows ではターミナルウィンドウを「コマンドプロンプト」とか「パワーシェル」ということがあります。

りません)。Anaconda ではデータサイエンスで一般的に使われるツールやライブラリがインストール済みの状態で提供されるため、本書で紹介する範囲ではソフトウェアパッケージを追加する必要はありません。どちらも公開サービスですが、Python を使い込んでいくためにはソフトウェア取得ツール `pip` または `conda` の利用が大きなポイントになります。

ソフトウエア群の管理方法が違っているため、両者を同時にインストールして使う分けることは初心者ではトラブルの元になるため、どちらか一方を決めて Python 環境を整えることを勧めます。

A.2.2 インストールの実際

以下、簡単に公式 Python 環境の構築を紹介します。インターネットに多くの情報が寄せられていますから、インストールするまえに全体の様子を確認してから作業に取り組んでください。

A.2.2.1 Windows の場合

Windows で公式版 Python 環境を整えるためには、故郷 https://www.python.org からインストーラをダウンロードして Python3 をインストールします（Python2 系の開発は終了しています)。

A.2.2.2 macOS の場合

macOS にはデフォルトで Python がインストールされており、Big Sur(ver.11.2.1) では Python3 が/usr/bin/python3 にあります。macOS でも同様に公式版 Python を https://www.python.org からインストーラによってインストールできますが、あらかじめパッケージマネージャ Homebrew(https://brew.sh/index_ja) をインストールしておいた場合、公式 Python をインストールするには次のように version を指定してインストールします。

```
$ brew install python@3.9
```

A.2.3 Python 環境の確認

インストールが終了したら、ターミナルウィンドウを開いて次のように Python コマンド `python` のバージョンを確認してください。

```
$ python --version
```

公式版 Python をインストールしただけでは本書で必要になる Python 外部ライブラリパッケージが含まれていません。公式版 Python に必要な外部パッケージの導入には PyPI のパッケージ管理ツール `pip` を使います。まず、ターミナルで `pip` を使って現在の Python で利用できる外部パッケージの一覧をリストしてみましょう。パッケージの導入状況やそのバージョンを把握するために多用するコマンドです。

```
$ pip list
```

では、`pip` を使って外部ライブラリパッケージをインストールしてみましょう。本書での計算プログラムで欠かせない数値計算パッケージ numPy をインストールするには次のようにします。

```
$ pip install numpy
```

本書で頻繁に累葉するグラフィックスライブラリ matplotlib、科学計算ライブラリ scipy、記号計算ライブラリ sympy を一度にインストールするのは次のようにカンマ (`,`) で区切って並べます。

```
$ pip install matplotlib, scipy, sympy
```

ライブラリパッケージのインストールにおいて、依存関係にあるパッケージは自動的にインストールされます。インストールしたライブラリパッケージ群の整合性は次のように確認できます。

```
$ pip check
```

問題ない場合にはメッセージ"No broken requirements found."が表示されます。外部パッケージのインストールを繰り返したり、一括バージョンアップしたときには整合性をチェックしてください。依存関係に問題がある場合には、パッケージを改めてバージョン・ダウンしてインストールし直す必要があります。また、Python 自体のバージョンによって追加できない外部パッケージが存在します。

ターミナルを開いてコマンド入力によって作業を進めることは、プログラミングのように積極的にコンピュータ活用を目指す場合、いずれ避けて通ることはできません。経験を積んで徐々に慣れていきましょう。

A.2.4 Jupyter または JupyterLab のインストール

さらに、あとから活用するために IPython および Jupyter Notebook もインストールしておきましょう（節 A.3 参照)。`python` コマンドを実行して現れるプロンプト`>>>` ととも

に提供される対話的実行環境 PEPL（Read-Eval-Print-Loop）を強化した上でさまざまな支援機構を使えるようにした Ipython を次のようにインストールします。

```
$ pip install ipython
```

さらにこれらを Web ブラウザを使って Python プログラムと書式付きテキストをノートブックとして保存できるするものが Jupyter notebook で、次のようにインストールします。

```
$ pip install jupyter
```

Project Jupyter は Python だけでなく他のプログラム言語のプラットホームとにもなっています。現在、Jupyter notebook の後継プラットホームである JupyterLab に移行しており、次のようにとインストールします。

```
$ pip install jupyterlab
```

Jupyter をサポートしている環境たとえば、Julia（https://julialang.org）や SageMath（https://www.sagemath.org）などは JupyterLab を使うと表示されるアイコンを直接選択して利用することができます。

A.2.5 Python プログラム

コンピュータで実行される動作の連なりをプログラミング言語で記述したものをスクリプト（script）またはコード（code）あるいはプログラム（program）と呼ぶことにします。プログラムを構成する単位をステートメント（文: statement）といい、Python では Python インタプリタが先頭から順にステートメントを実行します。

目的とする処理のためにプログラムを構想し作業を進めることをプログラミング、Python インタプリタに渡して（ブロックの）先頭から順に途中で止めることなく（ブロックの）最後まで**一括実行**する目的で書かれたファイルを Python コードまたは Python スクリプトといいます。Python スクリプトはプレインなテキストファイルとして文字コード utf-8 で書くのが現在の標準です。たとえばスクリプト 1.2-1 `plot-sin.py` を一括実行するには、ターミナルで次のようにして Python コマンドにプログラムファイルを渡します。

```
$ python plot-sin.py
```

コンピュータのファイル管理システムについての知識がないと、ターミナル（コンソール）の利用、「指定したコマンドに目的ファイルを渡す」という部分で躓いてしまうかもしれません。身近でコンピュータ操作に詳しい人の助力を仰げないときには、節 A.4 で紹介する Google Colaboratory を利用する方が手っ取り早いかもしれません。それでも、ファ

イルをどこに保存しているか、どのようにして指定コマンドに目的ファイル内容を認識させるかについて常に意識を向けるようにしてください。

コード 1.2-1　plot-sin.py

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2 * np.pi, 100)
plt.plot(x, np.sin(x), 'r')
plt.show()
```

1.2-1 では、1 行目で数値計算ライブラリ Numpy を短縮形 np として利用できるように、2 行目でグラフィックス表示ライブラリ matplotlib.pyplot を plt として利用できるように読み混んでいます。4 行目では区間 $[0, 2\pi]$ を等間隔に 100 点とった値を 1 次元 NumPy 配列 x として代入し、5 行目は配列値を x-成分に各配列値の NumPy 関数 sin 値を y-成分とする点を赤色でつないでプロットする指示を与えて、6 行目で実際に描画させています。問題なければ描画されたウィンドウが開きます。

スクリプトの一括実行とは対照的に、スクリプトのステートメントを 1 つずつ実行する方法があります。これを REPL（Read-Eval-Print-Loop）または Python シェルと呼んでいます。REPL では Python インタープリタの世界に入って、ステートメントを 1 つずつ入力しその結果を確認しながら次のステートメントを入力するという具合に対話的に実行できます。

REPL のためにはターミナルで引数なしでコマンド `python` を実行し、現れるプロンプト»>にキーボードからステートメントを順次入力して（リターンまたはエンターで）実行していきます。REPL を終えるには `quit()` または `exit()` を入力します（括弧 `()` は必要です）。

```
$ python
Python 3.9.1 (default, Jan 26 2021, 01:30:54)
[Clang 11.0.0 (clang-1100.0.33.17)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 2 * np.pi, 100)
>>> plt.plot(x, np.sin(x), '-r')
[<matplotlib.lines.Line2D object at 0x11b1561d0>]
>>> plt.show()
```

REPL では過去の入力履歴を矢印キーで呼び出せますが、残念ながら REPL の内部から指

定したプログラムファイルを読み込んで一括実行することができません。REPL はちょっとしたステートメントの動作確認には便利で、本書でも説明のために利用することがありますがすが、本格的なプログラムの開発には向いていません。

A.3 IPython と Jupyter Notebook

IPython を使うにはターミナルから起動するか Jupyter ノートブックを使います。IPython は Python シェル (REPL) を高度に強化した対話型環境を提供します。IPython/Jupyter（あまたは JupyterLab）のインストールは節 A.2 に紹介しました。

IPython 内ではこれから説明するように、通常の Python ステートメントだでなく、プログラム開発支援のために各種のマジックコマンド、たとえば `%run` で目的のスクリプトファイルを一括実行したり、`%hist` や`%save` を使って入力履歴を表示・書き出したりすることができ、Python REPL よりもたいへん効率的なプログラム開発が可能になります。IPython での入出力はターミナルからだけでなく、Jupyter ノートブックとして Web ブラウザを仲介した場合のコードセルでも利用できます。

また、IPython の利用で重宝するのは記号 ! を先頭につけて OS のシャルコマンドが実行できることです。たとえば、Python 環境の外部パッケージのインストール状況を一覧するための PIP コマンド `pip list` を IPython から一旦出ることなく次のようにして IPython から知ることができます。

```
In [40]: !pip list
```

ただし、IPython や Jupyter ノートブックで実行可能なマジックコマンドは Python 言語の一部ではありません。マジックコマンドを含む IPython への入力をそのままファイルに書き出し、Python インタプリタに渡して一括実行を試みるとマジックコマンドの箇所で実行に失敗します。

マジックコマンドに続いて?をつけるとオプションの使い方など詳しい説明が表示されます。たとえば、IPython の入力履歴を保存する`%save` について知りたければ次のようにしていつでも確認できます。

```
In [50]: %save?
```

コマンド説明モードから抜けるには `q` を入力します。

さらに、マジックコマンドには記号 `%`を 2 つ続けて `%%` から始めると 1 行だけでなく複数行（セルと呼びます）に適用できるセルマジック (cell magic) もあります。たとえば、IPython 内で Python コードを複数行に渡って実行することができます。

コマンド	意　　味
%lsmagic	マジックコマンドの一覧
%ls	作業ディレクトリを一覧
%pwd	作業ディレクトリのファイルパス
%cd	作業ディレクトリの変更
%hist	入力履歴
%save	履歴を書き出す
%cpaste	既存プログラム片のペースト
%load	指定プログラムを読み込む
%matplotlib	Matplotlib を対話的に使う
%prun	コードプロファイルしてステートメントを実行
%run	指定プログラムの一括実行
%timeit	プログラム片の実行時間計測

表 A.1　IPython が提供するマジックコマンド（ごく一部）。マジックコマンドに?をつけて入力（たとえば、%save?）すると利用説明が表示される。

```
In [51]: %%python
  ...: print('Hello')
  ...: for i in range(5):
  ...:     print(i ** 3)
  ...:
Hello
0
1
8
27
64
```

ただし、%%python 以降に書いたコードは以前の変数内容を引き継がないため、次のような進行はエラーとなります。

```
In [52]: a = 3
In [53]: %%python
   ...: print(a)
   ...:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

A.3.1 IPython を使う

IPython をターミナルで起動した場合を紹介します。IPython 固有の使い方は Jupyter ノートブックのコードセルに対しても同様に可能です（Jupyter ノートブック複数行の入力が可能でたいへん便利です）。

ターミナルで IPython を次のようにして起動すると、スタートメッセージに続いて入力を促すプロンプト `In [1]:` が表示されます。

```
% ipython
Python 3.9.1 (default, Jan 26 2021, 01:30:54)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.20.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

IPython の終了には `quit` または `exit` を入力します（Python REPL で `quit()` や `exit()` のように括弧 `()` は不要）。

IPython シェルでは、入力（`In`）と出力（`Out`）とにプロンプトが区別され、`In [4]` や `Out[4]` のように連番が付きます。実際に起動してみてわかるように、出力をともなわない入力には `Out[]` プロンプトは出力されません。

A.3.1.1 matplotlib のインタラクティブサポート `%matplotlib`

以下のようにして、IPython で matplotlib をインポートしてグラフィック描画を行う場合、インポートに先立ってマジックコマンド `%matplotlib` （Jupyter ノートブックの場合には`%matplotlib inline`）を入力しておくと、対話的に描画を確認できます。こうしておけば、プロットコマンドが実行されるたびに、現在開いている描画ウィンドウ内でプロット内容が更新されるようになります。こうした場合、ターミナルで IPython をつかている場合には引き続くプロット描画の結果は現在の描画ウインドウに重ね描きされます（プロット結果が上書きされて不都合な場合には、一旦描画ウィンドウを閉じて再描画してください）。一方、Jupyter においては、引き続いてプロット描画をするたびに新たにインラインとしてウィンドウ内に追加されていきます。

```
$ ipython
Python 3.9.1 (default, Jan 26 2021, 01:30:54)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.20.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 3 + 5 * 6
```

```
Out[1]: 33

In [2]: %matplotlib
Using matplotlib backend: MacOSX

In [3]: import numpy as np

In [4]: import matplotlib.pyplot as plt

In [5]: x = np.linspace(0, 2 * np.pi, 100)

In [6]: plt.plot(x, np.sin(x), 'r')
Out[6]: [<matplotlib.lines.Line2D at 0x1236ae780>]

In [7]: plt.show()

In [8]: plt.plot(x, np.cos(x),'b', linestyle='dashed')
Out[8]: [<matplotlib.lines.Line2D at 0x1214df490>]
```

ここで示した IPython への逐次的入力において、`In[6]` の `plt.plot(x, np.sin(x), 'r')` を実行すると描画ウィンドウが開きプロット結果を得るので、`In[7]` の `plt.show()` が不要であるかのようにみえます。しかし、`plt.show()` の実行をスキップして、`In[7]` の入力として `In[8]` で与える描画コマンドを実行させようとしても、現在の描画ウィンドウを閉じない限りはプロット表示ができません（描画ウィンドウを閉じると新しいプロット結果が描画されます)。

IPython や Jupyter ノートブックを使う上でマジックコマンド `%matplotlib` が必ずしも必要なわけではありません。描画コマンドに続いて `plt.show()` でそのたびごとに描画出力させるようにしても構いません。本書で掲げる Matplotlib を使うスクリプトでは、通常の Python コマンドを使って一括実行してすべての描画結果が得られるように、IPython のマジックコマンド`%matplotlib` を省略し、描画コマンドのあとにその都度 `plt.show()` を記載するようにします。

A.3.1.2 IPython の履歴 `%hist` と入力履歴の保存 `%save`

この IPython 内でマジックコマンドを含む 2 行から 7 行までの入力履歴をファイル `iplot-sin.py` に保存するために、次のようにしてマジックコマンド`%save` を実行します。

```
In [9]: %save iplot-sin 2-7
The following commands were written to file ‘iplot-sin.py‘:
```

```
get_ipython().run_line_magic('matplotlib', '')
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2 * np.pi, 100)
plt.plot(x, np.sin(x), 'r')
plt.show()
```

IPython を起動したフォルダ（ディレクトリ）に、拡張子.py が付いた次の内容のテキストファイル iplot-sin.py があることを確かめてください。ファイル iplot-sin.py 内容は次のようになっているはずです。先頭行はコメントで（記号 # から行末まで Python に認識されません）で、文字コード utf-8 で記述されていることを示しています。

```
# coding: utf-8
get_ipython().run_line_magic('matplotlib', '')
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 2 * np.pi, 100)
plt.plot(x, np.sin(x), 'r')
plt.show()
```

IPython の履歴はマジックコマンド %hist で確認できま。オプションなしの %hist はすべての入力履歴を番号無しで、オプションをつけて %hist -n で番号付きの入力履歴、%hist -o とすると各入力の番号つき履歴が出力と一緒に表示されます．オプションは組み合わせて使うことができ、

```
In [75]: %hist -on 50-60
```

は重力の 50 番から 60 番まで番号付きで出力も合わせ表示します。

A.3.1.3 IPython からの一括実行 %run

現在の描画ウィンドウを閉じた上で、保存されている iplot-sin.py を IPython から読み込んで一括実行するマジックコマンド %run を次のように試してみましょう（読み込むべき正確なファイル名がわからなくなった場合にはマジックコマンド %ls を入力してファイル・フォルダのリストを一覧するのもよいでしょう)。Python スクリプトを読み込んで一括実行するにはマジックコマンド%run に目的のファイルを渡します。

```
In [10]: %run iplot-sin.py
Using matplotlib backend:  MacOSX
```

Ipython 内では問題なく描画されたはずです。しかしながら、このファイルを iplot-sin.py を Python コマンドに渡して

```
$ python iplot-sin.py
```

で一括実行しようとしようと試みると、NameErrorとして'get_ipython' is not definedというメッセージを出力して失敗します。また、IPython（Jupyter）内であっても%runで正しく一括実行できるためには、このファイル内にあるget_ipython().run_line_magic('matplotlib', '')のようにマジックコマンドの正式な書式に従っている必要があります。読み込むファイル内の1行に%lsと記載して%runで一括実行を試みても失敗します(get_ipython().run_line_magic('ls', '')と正式な書式で記載する必要があります)。

A.3.1.4 IPysonへの一括ペースト %cpaste

ターミナルIPythonを使っている場合、開発途中のプログラム片の検証、たとえば関数定義など他の箇所から独立したステートメントブロックの動作確認のためにIPythonへの入力を各行ずつ逐一（コピーペストするとしても）入力するのは非効率です。そうした目的のためにペースト実行するマジックコマンド%cpasteが用意されています(Jupyterでは何ら問題なくコードセルに複数のプログラム行を貼り付けることができるので%cpasteは不要です)。%cpasteは次のように使います。IPythonに張り込みたいプログラム片をコピーしておいた上で、%cpasteを入力してから現れるプロンプト：を確認してからペースト（Windowsでは Ctrl+V、Macでは Command+V）でペーストし、表示される::::::でリターン（エンター）してから--を入力して終了します。

```
In [11]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:def fibo(n):
    if n == 0:
        return(1)
    elif n == 1:
        return(1)
    else:# n > 1
        return(fibo(n - 1) + fibo(n - 2))::::::
:--

In [12]: fibo(4)
Out[12]: 5
```

ここでは、検証したい関数定義 fibo(n) が動作するかを IPython にペースト入力し、In[12] に fibo(4) を入力して関数値を確認しています。

A.3.1.5 IPython を使って Python スクリプトを書く

今まで説明したターミナル IPython の対話性と入力履歴を保存できる機能を使って、手探りで試行錯誤を繰り返しながら目的とする Python スクリプトファイルを完成するプログラミンを行うことができます。

IPython の入力履歴を`%save` を使ってファイル `myplot.py` として書き出して（拡張子 `.py` は自動的に付与される）目的とする Python スクリプトの素案に使えばよいのです。別途テキストエディタでこのファイルを開き、エラー入力や IPython 内で使ったコマンドなど Python スクリプトとして無用な箇所を削除して保存し、これを改めて IPython から`%run` で読み込んで一括実行を試みデバックしながら、開いているファイルのコードを追加指定していくやり方です。

しかしながら、節 A.3.2 で紹介する Jupyter を利用する場合と違ってターミナルからの IPython の利用ではスクリプト全体を見渡す機能がないために、最初からテキストエディタで目的とする Python スクリプトファイルを書き、これを IPython 内から`%run` 一括実行してデバク修正を編集中のスクリプトファイルの反映する作業を繰り返す方が手っ取り早いことになります。

本書の範囲では格別複雑なプログラム開発を紹介するわではありませんから、テキストエディタで Python スクリプトを書き、これを Python コマンドに渡して実行したりまたは IPython 内で `%run` 一括実行しながらエラーをデバクしながら目的の Python スクリプトを完成する方法で十分です。

複雑で巨大なプログラミング開発の手法はソフトウエア工学として長年研究・提案されてきました。いきなり最終目標を目指すのでなく、問題解決のためのプログラム対象を部分問題に分割しそれらの小問題の解決に還元する分割統治法の採用やオブジェクト指向の方法を取り入れてコードロジックを再検討することが必要になるでしょう（Python でオブジェクト指向に従ったスクリプトを書くことが可能ですが、本書で紹介するコードは計算手順を示すために手続き的に書いておりクラス定義を含みません）。

本書では各部分問題の達成において、達成目標を手前に置きながらまず正しく稼働するスクリプトを書いてから、徐々に本来の目標に向けて拡張するという方針を取りました。

A.3.2 Jupyter を使う

目的とする計算を実行するためのスクリプトの完成は昔も今もプログラミングにおける主目的であることはかわりません。プログラム言語処理系から見えなくする特別な記号を導入して、Python では記号 # から行末までが無視されることになり、コメントを加えてスクリプトの可読性の向上を助けてきました。

しかしながら計算結果を出力するだけのスクリプト（と僅かなコメント）だけでは問題

としている対象理解のためには必ずしも十分とは言えません。対象としている計算モデルやそれを実行して得られるデータを多面的に検討する、たとえば処理方法を変更してみたりあれこれグラフィックス表示した結果を観察しながらインタラクティブに理解を深めることが、深い理解や発見をもたすことが少なくありません。

これから紹介する Jupyter プラットホームの利用は計算資源を多く必要としますが、研究対象を 1 つのストーリーとして語りながらプログラム実行結果とそのドキュメントを一体として取り扱うことを可能にします[3)]。Jupyter ノートブックは Python だけでなく Julia や SageMath などを使って説得力の高いプログラミングを含む報告作成に大いに貢献します。

図 A.1　Jupyter Notebook で起動して、[新規] から Python3 を選択して Python ノートブックを開く。

A.3.2.1　Jupyter notebook または JupyterLab の起動

Jupyter で Python ノートブックを利用するには Jupyter notebook またはその後継版である JupyterLab を起動します。JupyterLab では利用インターフェースがより便利に工夫されていますが、ノートブックとしての利用は双方で同じです。どちらも Jupyter サーバーの起動と処理にあわせてターミナルに多数のメッセージが表示されます。

Jupyter notebook を起動するには次のようにターミナルに入力します（図 A.1 参照）。

---

3)　報告を発表した著者だけでなく読者もまたその状況や結果を再現・検証に参加できることが大切だという認識が科学の発展を支えてきました。データ解析を含む報告書でも同様に、計算結果だけでなく、データ自身およびデータをどのように取り扱って結果を得たのかというプログラムの双方が提供されるべきだという考えが定着しつつあります。

とりわけ現在のように取り扱う対象が複雑でデータもますます膨大となってくると、結果報告だけではそこで取り扱われているデータ自体や分析方法の正当性を検証することが困難になってきています。これは「研究における再現可能性の問題」として 21 世紀の科学の課題として考えられています。

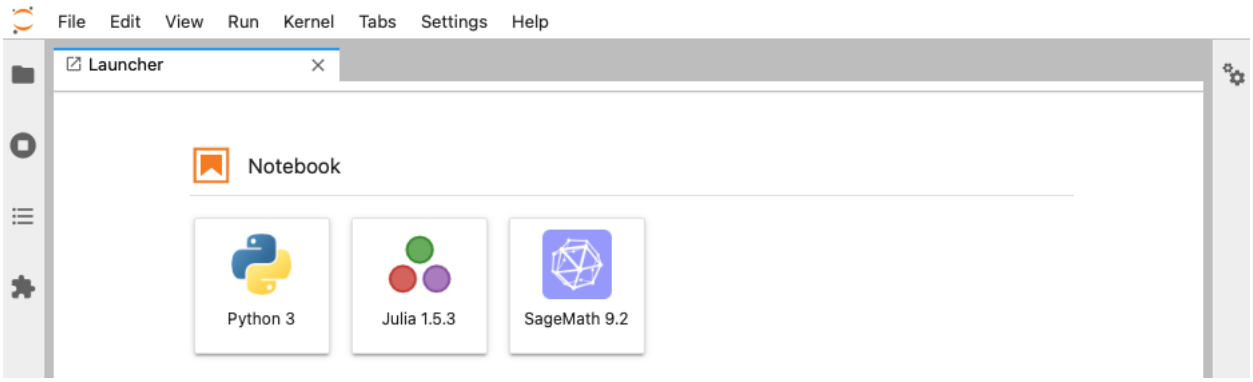


図 A.2　JupyterLab を起動するとノートブックランチャーが現れる。Python3 を選択して Python ノートブックを開く。

```
$ jupyter notebook
```

Jupyter notebook を起動すると、Web ブラウザには Jupyter Notebook の初期ディレクトリの様子が一覧されます（ノートブックダッシュボード）。Python ノートブックとして作業するために図 A.1 のように [New] から Python3 を選ぶと、タイトル未定 (Untitled) の新規ノートブックが開かれ空のコードセルが表示されます。Jupyter の終了には、保存すべきノートブック内容とファイル名を確認してから起動ターミナルで `Ctrl+C` を押して Jupyter サーバーを停止します。

JupyterLab を起動するには次のようにターミナルに入力します（図 A.2 参照）。

```
$ jupyter lab
```

JupyterLab の終了は、保存すべきノートブック内容とファイル名を確認してからノートブックを閉じて起動ターミナルで `Ctrl+C` を押して JupyterLab サーバーを停止します。

JupyterLab では、図 A.3 のように、タブでテキストファイルを含む複数のファイル内容を同時に開いて作業することできます。Python ノートブックのコードセルと Python スクリプトとの間のコピー・ペーストやテキストセルへの説明文の入力などを効率的に進めることができます。また、JupyterLab ウィンドウ左端のフォルダアイコンで作業フォルダ・ファイルを一覧させることができます。

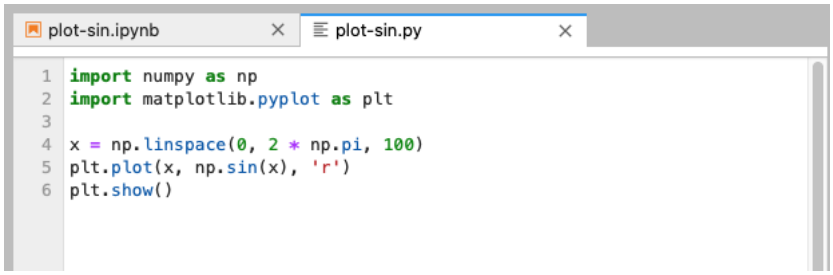


図 A.3　JupyterLab は複数のファイルを開いてタブで切り替えながら作業できる。

A.3.2.2　ノートブックの使い方

ノートブックの入力単位をセルといい、実行する内容を記載する**コードセル**と説明文を記載する**テキストセル**（Markwon）の 2 種類があります。Jupyter を使う恩恵の一つは、テキストセルにプレインテキストだけでなく **Markdown** 書式が使えることです（図 A.5 参照）。

コードセル内への入力は 1 行以上の Python ステートメントをまとめて記載できるだけでなく、IPython で実行可能な各種コマンドも入力できます。ノートブックを使ってプログラム開発する場合には、デバッグし易いように複数の Python ステートメントを適宜まとめてコードセルに入力し、マジックコマンドなど IPython 固有のコマンとを分けて入力するようにします。こうしておくと、コードのデバク作業が単純化され、各コードセル内容をコピーしてコードブロックとして他のスクリプト流用したり一括実行可能な Python スクリプトを書き出し易くなります（ノートブックファイルは Python ステートメント以外の情報を含んでいます）。

なお、Python ノートブックから Python スクリプトへの一括変換は常に可能です。節 A.3.2.6 で紹介するように、Jupyter の [ファイル] メニューから [書き出し]（Export Notebook As）から Executable Script を選んで書き出すと入力番号やテキストセルなどがコメントアウトされて拡張子 .py がついたファイルとして保存されます。ただし、ノートブック内で使ったマジックコマンドは IPython/ノートブック内で`%run%`内で一括実行できるように、正式なIPython コマンドに変換されて残されます（IPython の入力履歴の保存`%save` と同じです）。Python ステートメントだけを使ったスクリプトにしたい場合にはテキストエディタでそれらを使ったさらに#でコメントアウトしてください。

ノートブックにセルを追加するには [挿入] メニューまたは [＋] を押します。現在の選択しているセルをテキストセルまたはコードセルに変更することも可能です。。また、選択したセルを [↑] や [↓] でノートブック内で移動させることもできます。

選択したコードセルを実行するには [実行] メニューから選択または `Shift+Return` を

押します。実行された順にコードセルに連番で入力番号が付きます（出力があるときは相応する出力番号も付きます）。

開発しているノートブックが全体として正しく動作するかを検証したり、既に保存してあるノートブックを読み込んで動作を確認したいときには、[セル] メニューから「すべてを実行」を選ぶと入力してあるコードセルを一括実行します。このときコードセルは先頭から “順にすべて実行” されます。

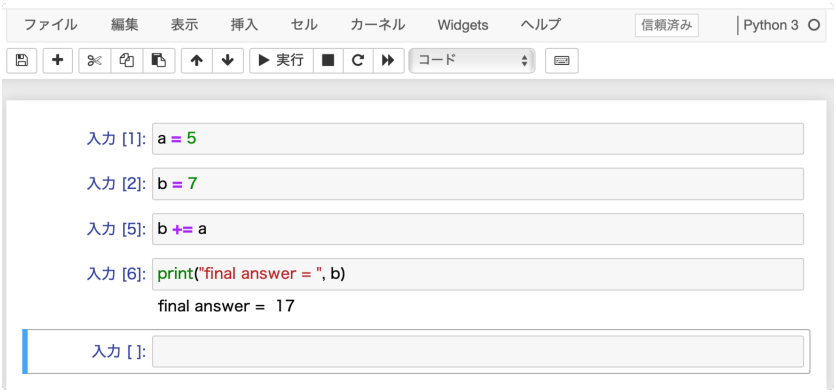


図 A.4　意図しない結果をもたらす不用意なセル実行

A.3.2.3　ノートブックにおけるコードセル実行時の注意

既に入力してあるコードセルを選択して再実行したりセル内容を変更して実行するとそのコードセルには新しい番号が付与されます。このようにノートブックでは入力コードを一望することができる上に、入力したコードセルを指定して実行し直すことで（それまでの入力経歴に依存するとはいえ）コードセル内のエラーをデバグすることができるというノートブックプログラミングならではの利点があります。

しかしながら、ノートブックではコードセルの実行結果が入力履歴に依存するが故に注意すべきこともあります。図 A.4 のように、コードセルの実行結果（入力 [6] の結果）がノートブックのコードセルを先頭から実行した結果（12）と一致しません。過去のコードセル遡って実行した場合にはコードセルの並びと対応した逐次実行結果との対応を保つことができず（このままノートブック内容を保存した場合には）意図しない結果を残すことになります（その発見はコードエラーとは異なり大変困難です）。

ノートブック利用ではコードセルの並びと実行結果との整合性管理はユーザの責任です。ノートブックは固定された場所にある入力セルを後から何回でも実行することができるために、思わぬ副作用をもたらし誤った印象を与えてしまうことがないように計算の流れには十分な注意を払ってください。あわせて節 A.3.2.5 で紹介するように、テキストセ

ル（Markdown）を積極的に活用してプログラム処理の流れに解説を加えてプログラムの意図を明確しておくとよいでしょう。

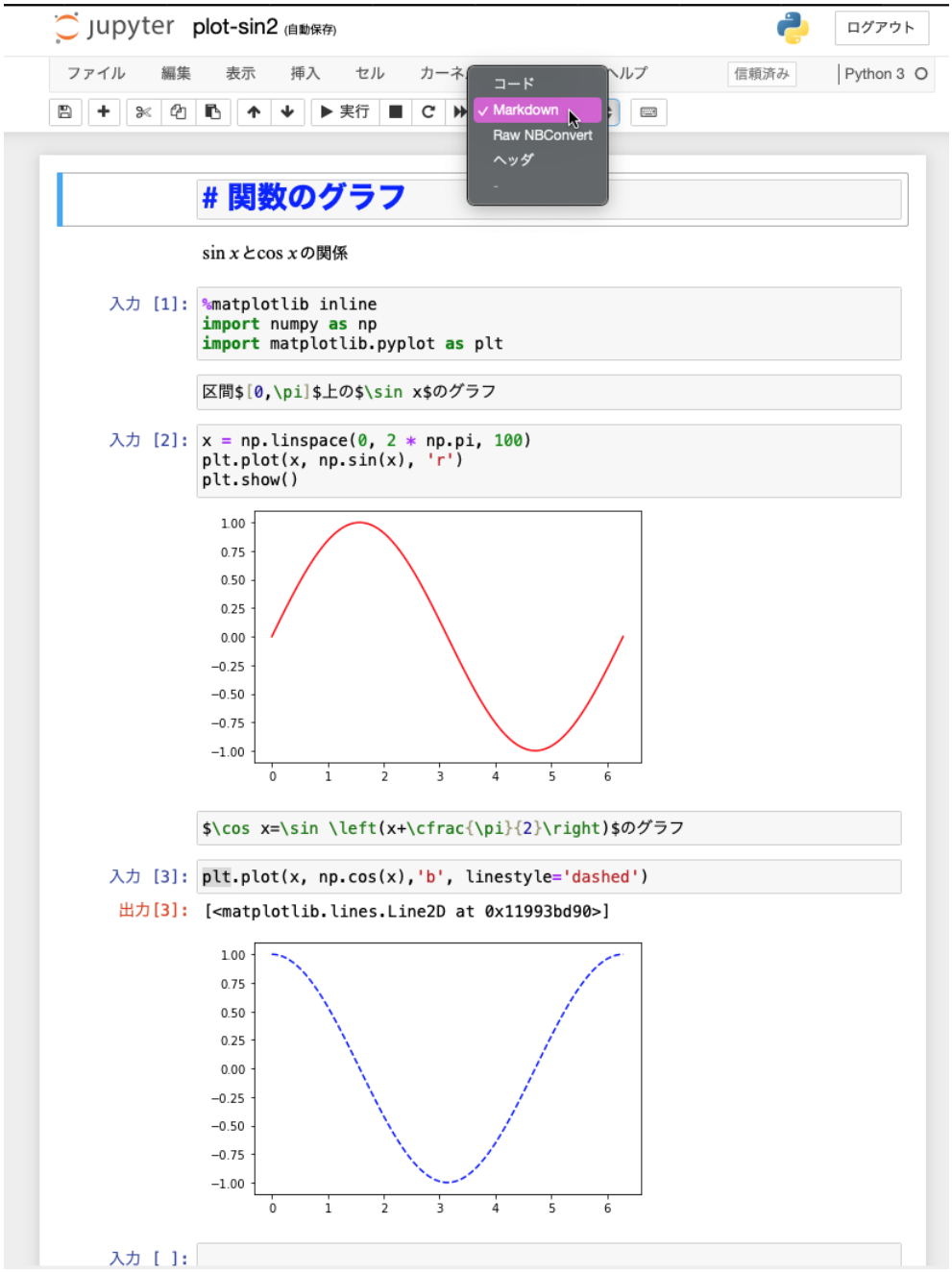


図 A.5　Jupyter Python ノートブックにはテキスト（Markdown）セルが挿入でき、何行に渡って記述できる。プレインテキスト以外に Markdown 書式で記載でき、さらに LaTeX 記法に従った数式を$...$内に書いて行内モードで、$$...$$内に書いて行モードで表示できる（整形結果は `Shift+Enter` で確認）。

A.3.2.4　ノートブックの保存

Python ノートブックではグラフィックス出力があればそれらを含めて拡張子.ipynbの付いた Jupyter Notebook 形式で保存できます。新規にノートブックを開くと、左上のJupyter アイコンの右隣に Untitled（あるいはその連番）のタイトルが付き、ノートブック内に変更があるとその都度自動保存されます。現在のタイトルをクリックしてふさわしいタイトルに変更しておきましょう。

現在のノートブックファイルの内容を改めて指定した場所に保存するには、[ファイル]メニューから [名前をつけて保存]（Save As）をクリックします（ファイルパスの区切り記号はスラッシュ/を使います)。一方、既に保存してあるノートブックファイルを開いて作業する場合には [ファイル] の [開く] から目的のノートブックファイルを選択します。

A.3.2.5　テキストセルの利用

Jupyter ノートブックでは、追加または選択したセルを [セル] メニューから「コード」または「テキスト」（Markdown）に変更することができます。テキストセルはコードセルの実行には影響を与えず、何行にもわって入力できるため、通常のスクリプト内のコメント以上の役割を持ちえます。

特筆すべきことは入力テキストを Markdown 書式で記載することができるという点です。さらに LaTeX 記法に従った数式を$...$内に書いて行内数式モードで、$$...$$内に書いて行数式モードで表示できるため、わかり易いノートブックドキュメントの作成が可能になっています。その整形結果は Shift+Enter で確認できます（図 A.5 参照)。

テキストセルをうまく利用したノートブックは、スクリプトへのコメントという位置づけから離れてプログラムコードとその実行結果が添えられた説明ノートとして活用する道が開かれます。

A.3.2.6　Jupyter ノートブックのファイル形式

Jupyter ノートブックで取り扱うファイル形式を確認しておきましょう。Jupyter ノートブックの保存時のデフォルト書式は拡張子.ipynb が付いた Jupyter Notebook 形式で、データ交換書式 JSON で保存されます。

さらに、Jupyter の [ファイル] メニューから書き出される（Export Notebook AS）書式には、Asciidoc、HTML,LaTeX、Markdown、PDF、ReStructured Text、Executable Script, Reveal.js Slides があり、様々なドキュメント作成にたいへん重宝します（作成した Jupyter ノートブック自体は、節 A.4 で紹介するクラウドサービス Google Coraboratory の利用を前提とすれば、Google ドライブを経由してファイル共有することができます)。

なお、書き出される Executable Script とは、節 A.3.2.2 のノートブックの使い方で触れた

ように、Python ノートブックからテキストセルや入力番号をコメントアウトした Python スクリプトです（ただし、ノートブック内で IPython コマンドを使った場合には、正しく展開された IPython コマンドを含みます）。

繰り返しになりますが、ノートブックファイル形式は実行可能な単純なスクリプトとは違い、異なるプログラミング言語間でのデータのやりとりを可能にするデータ記述言語 JSON 書式に準拠して記述されています。

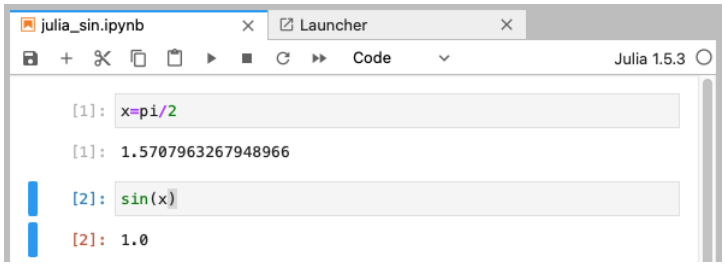


図 A.6　Jupyter Lab で Juli を使って、$\sin(\pi/2)$ の計算させる。

ためしに、Jupyter Lab でプログラム言語 Julia をつかって図 A.6 のように $\sin(\pi/2)$ の値を計算させたノートブックファイル（拡張子 .ipynb が付きます）を保存したときのファイル内容は次のようになります。Jupyter Notebook で取り扱うことができるように、入力コード以外の多くの情報を含んでいること、また、metadata セクションにカーネル種別や言語が明記されており、対応している環境ではファイルを開いて計算を続けることができるようになっています。

```
{
    "cells": [
      {
        "cell_type": "code",
        "execution_count": 1,
        "id": "polar-posting",
        "metadata": {},
        "outputs": [
          {
            "data": {
              "text/plain": [
                "1.5707963267948966"
              ]
            },
            "execution_count": 1,
            "metadata": {},
            "output_type": "execute_result"
          }
```

```
      ],
      "source": [
        "x=pi/2"
      ]
    },
    {
      "cell_type": "code",
      "execution_count": 2,
      "id": "helpful-index",
      "metadata": {},
      "outputs": [
        {
          "data": {
            "text/plain": [
              "1.0"
            ]
          },
          "execution_count": 2,
          "metadata": {},
          "output_type": "execute_result"
        }
      ],
      "source": [
        "sin(x)"
      ]
    }
  ],
  "metadata": {
    "kernelspec": {
      "display_name": "Julia 1.5.3",
      "language": "julia",
      "name": "julia-1.5"
    },
    "language_info": {
      "file_extension": ".jl",
      "mimetype": "application/julia",
      "name": "julia",
      "version": "1.5.3"
    }
  },
  "nbformat": 4,
  "nbformat_minor": 5
}
```

A.4 Google Colaboratory を使う

現在、様々なプログラム言語に対応した多くのクラウド統合開発環境（クラウド IDE）が利用できるようになってきました。クラウド IDE を使うとインターネット接続している端末のブラウザだけでスクリプトコードの記述・実行・デバッグ作業が可能になり、パソコンにプログラミング環境をインストールすることなく、タブレットやスマートフォンなどの端末でも同じ環境でプログラミングができるようになります。

クラウド IDE には潜在的に大きな需要があります。プログラミング初心者（とくに直接にアドバイスを受けられない自学者・独習者）にとっての最大の壁はするれば実行できその結果がどうなるかを得ることであり、その利用環境を整えること自体が困難な場合が多いのではないでしょうか。また、グループ学習や共同研究を成功させるためには、パソコンの OS や Python 環境の差異（バージョンの違いを含む）に起因するトラブルを回避し同一条件のもとで、さらにファイル共有しながら作業できることが望ましいです。

ここでは Python が使えるクラウド IDE として Google Colaboratory http://colab.research.google.com を紹介しします (通称 Colab と呼んでいます)。Google Colaboratory は Google クラウドプラットホーム上の仮想マシンが提供する原則無料で利用できる Python 実行環境で、Jupyter ノートブックの形で利用します。Google Colaboratory が提供する仮想マシンには科学計算や機械学習などの大抵の計算に必要な外部パッケージが既に用意されてれています。

Google Colaboratory にインストールされているパッケージとバージョンを確認するにはコードセルで次を実行します。

```
!pip list
```

パッケージをインストールするには

```
!pip install パッケージ名
```

で行うことができますが、仮想機械の Python 環境にインストールされるため、仮想機械の接続がきれたとき（節 A.4.1 で紹介する Google Colaboratory の利用条件）や新しく仮想機械を利用する場合にはこうしてインストールした外部パッケージを使うためにはサイドインストールし直す必要があります。こうした煩雑さを避けるために、節 A.4.3 で紹介しますが Google ドライブをマウントした後に、外部パッケージの保存先をユーザの Google ドライブにインストールしておき、Python にライブラリの検察パスを通知する方法を採用するとよいでしょう。

本書で取り上げるプログラムではプログラムの可読性を向上し、運用効率を高めるまでに型ヒントを積極的に利用します。そのために、書いたスクリプトコードを前もって型ヒントを静的チェックするツール `mypy` を使うのが便利ですが、現在 Google Colaboratory にはデフォルトで用意されていないため、インストールして利用します（節 A.4.3.4 参照）。

A.4.1 Google Colaboratory の利用条件

この仮想マシンの無料利用を実現するために Google Colaboratory では利用できる計算リソースを状況によって動的に変動させており、以下のような使用制限を設けています。この制限を超えると割り当てられた仮想マシン接続が削除され、再接続が必要になります。

- 12 時間ルール。ノートブックは最長 12 時間仮想マシンで実行されますが、12 時間経つと切断される。計算続行のためには必要な途中データを Google Drive に退避させるなどの工夫をする。
- （通称）90 分ルール。ノートブックを長時間（90 分程度）放置してアイドル状態にすると仮想マシンから切断される。ノートブックの画面右上の [再接続] ボタンから再接続してアイドル状態を解除するか、定期的に（60 分程度ごとに）ノートブックにリロードする。

本書の範囲では数十分以上かかるような巨大な計算量を必要としないため、これらの制限は実際には問題とはなりません。Google Colaboratory ノートブックに適切な名前をつけて Google ドライブに保存されたことを確認し、必要としないタブを閉じるように心がけるだけで十分です。

A.4.2 Google Colaboratory を使う

Google Colaboratory を利用するためには Google アカウントが必要です。Google サービスから誰でも無料で取得することができます。Colaboratory ノートブックの使い方は原則 Jupyte ノートブックと同じです。ノートブックから別ファイルにアクセスしたり書き出したりすることもできますが準備が必要です。

Google Colaboratory の利用開始は簡単です。Google サービスから Google ドライブを選び、マイドライブのウィンドウ左隅にある「新規＋」ボタンから [アプリを追加] から Google Colaboratory を選びます。Google Colaboratory を起動すると、仮想機械で実行された Jupyter とおなじような新規ノートブック（Untitled0.ipynb）が開きます。作成したノートブックファイルは Google ドライブに自動保存され、ファイル共有も可能になります。

図 A.7 はその様子を示しています。タイトルは目的に合わせて変更しましょう（Colaboratory ではノートブック拡張子も表示されています）。先頭のコードセルには、現

在の作業ディレクトリ（フォルダ）を取得するマジックコマンド `%pwd` を実行しています。ウィンドウ左端にあるフォルダアイコンから仮想機械のディレクトリツリーを表示させると `%pwd` で表示されたディレクトリ `/content` があることがわかります。

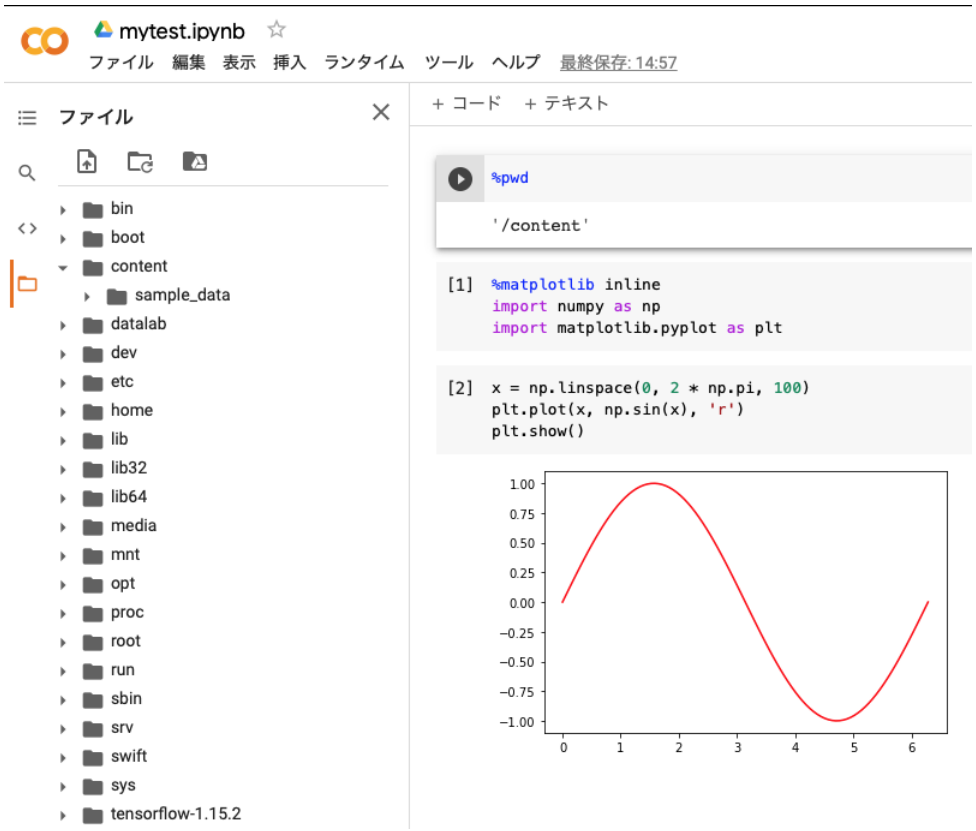


図 A.7　Google Colaboratory で Jupyter ノートブックと同じように使って区間 $[0, 2\pi]$ 上の正弦波を描いた様子。ウィンドウ左側のフォルダアイコンから仮想マシンのディレクトリツリーが表示できる。入力 [1] は現在の作業フォルダを確認するマジックコマンド `%pwd` で `/content` であることを示している。

作成したノートブックを Google ドライブに保存するには [ファイル] メニューの [保存] を選びます。Google ドライブではマイドライブ内に自由にフォルダを作成した上で、自由にファイルを移動させることができます（ファイル共有していた場合には注意が必要で、ファイル移動によって共有リンクが変わります）。

A.4.3　Google ドライブをマウントしてファイルを読み込む

Google ドライブにあるノートブック（拡張子 .ipynb）をクリックすると、Colaboratory アプリケーションが起動し、別のシステム内に稼働している**仮想機械** (virtual machine) 上にある Python 環境との橋渡しをしてノートブックの実行を支援します。

Google ドライブをサービスする仕組みと仮想機械とは区別されているために、ノートブックの実行に必要なデータ（ファイル）が Google ドライブにあったとしても、そのままではノートブックから Google ドライブのファイルにアクセスすることができません。図 A.7 で一部見たように、仮想機械のファイルシステムからは Google ドライブは見えていませんでした。

A.4.3.1 ファイルシステムのマウント

Python 環境を整えている仮想機械から Google ドライブにアクセスするためには、仮想機械でのマウント作業が必要です。使っているパソコンに USB メモリ（ストレージ）を挿入すると外部装置の追加を検出し、そのストレージ内部にアクセスできるようになります。このようなことが可能であるのはパソコンが自動的にマウント処理を行っているからです。

ファイルシステムの**マウント** (mount) とは、目的とするファイルシステム（ストレージ）を使っているコンピュータ内のディレクトリ (**マウントポイント**) に接続し、目的のファイルシステムを現在使っているコンピュータで使用可能にすることです。

今の場合、Google Colaboratory が橋渡しをしている仮想機械が現在のコンピュータに、Google ドライブが目的とするファイルシステムに相当します。

A.4.3.2 仮想機械から Google ドライブをマウントする

Google Colaboratory ノートブックから Google ドライブに置いたファイルの読み込みや書き出しが必要になる場合には、ノートブック内で次の手順を踏みます。

(1)必要とするファイルを Google ドライブに置く（アップロードする）。
(2)Colab ノートブック内で Google ドライブをマウントする。
(3)Colab ノートブック内に目的のファイルを読み込む。

仮想マシンのファイルシステムにあるディレクトリ `/content` 内にマウントポイント（ディレクトリ）`drive` を作成して Google ドライブ（マイドライブ）をマウントするために、コードセル内に次のように書いて実行します。このマウント作業は、ノートブック単位で必要です。

```
from google.colab import drive
drive.mount('/content/drive')
```

すると、図 A.8 のように URL が表示されるのでクリックして Google アカウントの認証を受けます（利用しているアカウントを確かめて「確認」を押します）。認証コードが発行

されるので、コピーして先程の Enter your authorization code の欄にペースト入力してください。

図 A.8　Colaboratory ノートブックに仮想マシン内にマウントポイント `/content/drive` を作成して Google ドライブをマウントするコードを実行するには Google 発行の認証コードの入力が必要。

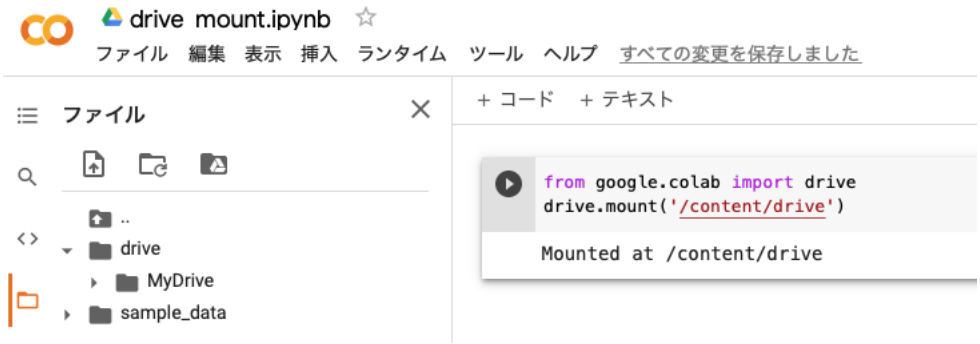


図 A.9　ノートブックウィンドウ左端のフォルダアイコンを開いて、仮想マシンの `/content` に Google ドライブ `drive/MyDrive` がマウントされたことが確認できる。

マイドライブがマウントされると図 A.9 のように、メッセージ `Mounted at /content/drive` が返ります。これによって仮想マシン上で実行されるノートブックからは、ユーザの Google ドライブにあるファイルへのアクセスは `drive/MyDrive` で始まるパスを使って指定できるようになります（節 A.4.2 で確認したように、ノートブックの作業ディレクトリはディレクトリ `/content` です）。

A.4.3.3 Google ドライブにあるスクリプトファイルの一括実行

Google ドライブにあるファイルにアクセスしてみましょう。たとえば、スクリプト 1.2-1 `plot-sin.py` を Google ドライブ（マイドライブ）内のフォルダ `MyScripts` に置いた場合、仮想マシンからの完全なファイルパスは

```
/content/drive/MyDrive/Myscripts/plot-sin.py
```

となります。Google ドライブをマウントした上で、このスクリプトを読み込んでマジックコマンド`%run` で一括実行するにはコードセル内に次のように入力します（ノートブックの作業ディレクトリが`/content` なのでそこからのファイルパスを指定しています。

```
%run drive/MyDrive/MyScripts/plot-sin.py
または
%cd drive/MyDrive/Myscripts
%run plot-sin.py
```

後者のように、マジックコマンド `%cd` で作業ディレクトリを目的ファイルを含みディレクトリに移動してから一括実行してもかいまいません。

ノートブックからユーザの Google ドライブ内の目的ファイルへのファイルパス内にディレクトリまたはファイル名に空白 ␣ が入っている場合にはディレクトリまたはファイル名を引用記号 " または ′ で囲む必要があります（何も対策しないと読み込みに失敗します)[4)]。

A.4.3.4 Colaboratory では外部パッケージを Google ドライブに保存する

Google Colaboratory では仮想機械にインストールした外部パッケージは、仮想機械への再接続や新規利用の際にはその都度再インストールする必要があします。この状況を回避するために、Google ドライブ（マイドライブ）をマウントした上で外部パッケージをマイドライブを指定してインストールしておき、仮想機械で実行する Python ノートブックで目的のパッケージが見つかるようにサーチパスに追加する方法を紹介します。

量子回路シミュレータパッケージ Qulacs は現在の Google Colaboratory ではデフォルトで用意されていないため、pip でインストールが必要です。そのために、Google ドライブ内に外部パッケージをインストールしておく場所を用意しておきます。たとえば、ノートブックを保存するフォルダ `notebooks` 内にパッケージインストール用のフォルダ `module` を準備しておきます。

まず、節 A.4.3.2 に従って、Google ドライブを仮想機械の `/content/` 以下に次のよう

4) ファイル名には空白文字を含まないように心がけましょう。空白の代わりにアンダースコア (_) を使うとよいでしょう。

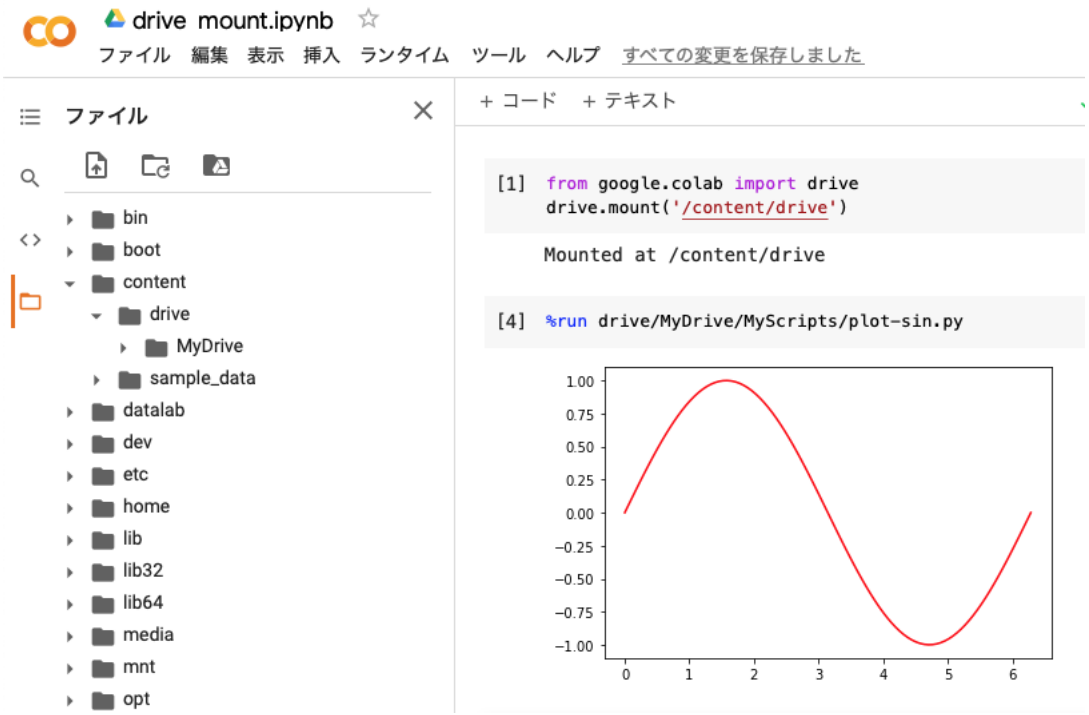


図 A.10　スクリプトファイル 1.2-1 を **Google** ドライブ（マイドライブ）内のフォルダ **`MyScripts`** に置いて、ノートブックからマジックコマンド **`%run`** で一括実行する。**Google** ドライブをマウントしていることが前提。

にして /content/drive/My Drive としてマウントしておきます。

```
from google.colab import drive
drive.mount('/content/drive')
Mounted at /content/drive
```

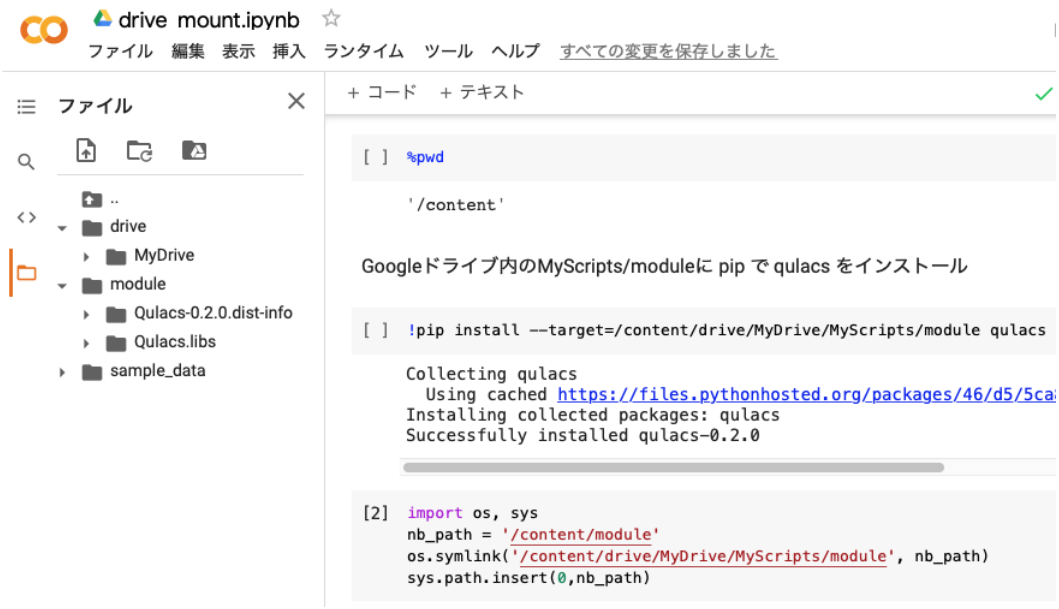


図 A.11　仮想機械の/content/module/に Google ドライブ内に用意しておいたフォルダに外部パッケージをインストールしておくと、ノートブック内で仮想機械の/content/module/にシンボリックリンクを張った上で、Pyhon モジュールをインポートする際のサーチパスとして加えるようにすると Google Colaboratory の利用上の制約に縛られることなくいつでも外部モジュールを使うことができる。

準備した Google ドライブ内の外部パッケージを格納しておくフォルダ MyScripts/module をターゲットディレクトリとして、!pip（記号!が pip の前についています）で目的の外部パッケージ qulacs をしてインストールします（ここではターゲットディレクトリを仮想機械での完全パスで指定しました）。

```
!pip install --target=/content/drive/MyDrive/MyScripts/module qulacs
```

こうしておくと、ノートブック内で Google ドライブにインストールしてある外部パッケージを収納しているディレクトリを os.symlink によって仮想機械内のディレクトリ /content/module へシンボリックリンクし、sys.path.insert によって/content/module をモジュールのサーチパスに加える以下のコードセルを実行した後で、qulacs 内のモジュールなど Google ドライブに保存してあるパッケージがいつでも

インポートできるようになります（図 A.11 参照）。もちろん、Google ドライブのマウントは必要です。

```
import os, sys
module_path = '/content/module'
os.symlink('/content/drive/MyDrive/MyScripts/module', module_path)
sys.path.insert(0,nb_path)
```

参考文献

■ Python 拡張モジュール

[1] Matplotlib User's Guide: https://matplotlib.org/users/

[2] NumPy Reference: https://docs.scipy.org/doc/numpy/reference/

[3] SciPy: https://docs.scipy.org/doc/scipy/reference/

[4] SymPy: https://docs.sympy.org/latest/tutorial/

[5] Comparison with SQL: https://pandas.pydata.org/pandas-docs/stable/getting_started/comparison/comparison_with_sql.html

■線形代数

[6] 長岡亮介,『線型代数入門講義』, 東京図書 (2010).

[7] 齋藤正彦,『線型代数入門』, 東京大学出版会 (1966).

■解析学

[8] 杉浦光夫、『解析入門 I, II』, 東京大学出版会 (1980).

■数値計算

[9] 伊理正夫/藤野和建,『数値計算の常識, 共立出版 (1985).

[10] 三井斌友/小藤俊幸/斉藤善弘『微分方程式による計算科学入門』, 共立出版 (2004)

[11] 桂田祐史,『常微分方程式の初期値問題の数値解法入門』, http://nalab.mind.meiji.ac.jp/~mk/labo/text/num-ode.pdf.

■微分方程式と力学系

[12] V.I. アーノルド,『常微分方程式』, 現代数学社 (1981).

[13] Brin,M. and Stuck, G., *Introduction to Dynamical Systems*, Cambridge University Press (2002).

[14] Hirsch, Smale and Devaney,『力学系入門（第 2 版）』, 共立出版 (2007)

[15] E.A. Jackson,『非線形力学の展望 I, II』, 共立出版 (1994)

[16] 森真/水谷正大,『入門力学系』, 東京図書 (2009).

■カオス、フラクタル

[17] アリグッド/サウアー/ヨーク,『カオス (1)』, 丸善出版 (2012).

[18] M. Barnsley, *Fractals Everwhere*, Dover Publications(2012).

[19] パイトゲン/リヒター、『フラクタルの美』、シュプリンガー・フェアラーク東京 (1988).

■古典力学

[20] 大貫義郎/吉田春夫, 『力学』, 岩波書店 (1994).

[21] 山本義隆/中村 孔一, 『解析力学 1/2』, 朝倉書店 (1998).

[22] E.T. Whittaker, 『解析力学 上/下』, 講談社 (1977).

[23] 浅田秀樹、『三体問題』, 講談社ブルーバックス B2167(2021).

■格子振動、エルゴード

[24] 斉藤信彦, 『岩波講座 統計物理学 第 10 章エルゴードの問題』, 岩波書店 (1978).

[25] 戸田盛和, 『振動論』, 培風館 (1968).

[26] 戸田盛和, 『非線形格子力学 (増補版)』, 岩波書店 (1987).

[27] 戸田盛和, 『非線形波動とソリトン』, 日本評論社 (2000).

[28] Weissert, T.P., *The Genesis of Simulation in Dynamics, Pursuing the Fermi-Pasta-Ulam Problem*, Springer-Verlag(1997).

■研究論文

[29] Berry, M.V., *Regular and irregular motion*, AIP Conference Proceedings 46, 16 (1978).

[30] Branchard, P., *Complex Analytic Dynamics on the Riemann Sphere*,Bulltim A.M.S 11(1984), 85-141.

[31] Fermi, E., Pasta, J. and Ulam, S., *Studies of nonlinear problems*, Los Alamos Report LA-1940(1955).

[32] Flaschka, H., *The Toda lattice. II. Existence of integrals*, Phys. Rev. B9(1972), 1924−1925.

[33] Ford, J., Stoddard,S.D and Turner,J.,*On the Integrability of the Toda Lattice*, Progress of Theoretical Physics 50(1973), 1547−1560.

[34] Hénon, M and Heiles, C, *The applicability of the third integral of motion: Some numerical experiments*, Astronom.J 69:1(1964), 73-79.

[35] Hénon, M, *Numerical study of quadratic area-preserving mappings*, Quart. Appl. Math. 27 (1969), 291-312.

[36] Hénon, M., *Integrals of the Toda lattice*, Phys. Rev. B9(1972), 1921−1923.

[37] Hénon, M, *A two-dimensional mapping with a strange attractor*, Comm. Math. Phys 50:1(1976), 69-77.

[38] Hénon, M, *On the numerical computation of Poincare maps*, Physica D 5 (1982) 412−414.

[39] Lorenz, E, *Deterministic nonperiodic flow*,J. of Atmos. Sciences 20(1963), 130—141.

[40] Mandelbrot, B., textitFractal Aspects of the iteration of $z \to \Lambda z(1-z)$ for complex Λ and z, Annals of the New York Academy of Sciences. 357(1980), 249−259.

[41] May, R. M., *Simple mathematical models with very complicated dynamics*, Nature 261:

5560(1976), 459–467.

[42] R. Show, Strange Attractors, Chaotic Behavior, and Information Flow, Z. Naturforsch.**36a**(1981) pp.80–112.

[43] Smale,S., *Differentiable dynamical systems*, Bull. Amer. Math. Soc. 73 (1967), 747-817 .

[44] Smale,S., *Mathematical problems for the next century*, The Mathematical Intelligencer **20**(1998) pp.7–15.

[45] Tucker, W., *The Lorenz attracter exist*, C.R.Acad.Sci.Paris Sér.I Math.328(1999): 1197.

[46] Tucker, W.,*Computing accurate Poincare maps*, Physica D 171(2002), 127–137.

[47] 上田睨亮,『非線形性に基づく確率統計現象–Duffing 方程式で表わされる系の場合』, 電気学会論文誌 A(1978), 167-173.

[48] Zabusky, N.J. and Kruskal, M.D., *Interaction of "solitons" in a collisionless plasma and the recurrence of initial states*, Phys. Rev. Lett. 15:6(1965), 240–243.

``

索引