卒　業　論　文

二次元多重連結領域上の構造安定な非圧縮流の木表現の可視化手法

2016SE024　**亀谷 拓磨**
2016SE076　**田島 嘉人**
2016SE090　**渡辺 康平**

指導教員　**横山 哲郎**

2020年1月

南山大学 理工学部 ソフトウェア工学科

Visualization method of tree representation of structurally stable incompressible flow in two dimensional multiply-connected domain

2016SE024　KAMEGAI Takuma
2016SE076　TAJIMA Yoshihito
2016SE090　WATANABE Kohei

Supervisor　YOKOYAMA Tetsuo

January 2020

Department of Software Engineering
Faculty of Science and Engineering
Nanzan University

要約

流体力学は，気体や流体の運動について取り扱う力学の主要な研究分野である．流体力学の歴史は古く，離散解析や数値解析によって研究が行われてきた．離散解析では，流れをトポロジカルな観点に着目することで，流れの本質的な構造を抜き出すことができ，流れの解析を効率的に行うことができる．木表現を使った離散解析手法が存在する．木表現は流れの構造を代数的に扱い，流れの解析を行うことができる．しかしながら，初学者にとって木表現は直観的に流れの構造を把握しづらい．

本研究では，木表現を用いて表現された流れのトポロジーを図上へ描画する方法を提案し，この方法にしたがった可視化プログラムの設計とプログラミング言語である Python および Asymptote による実装を行った。さらに、可読性の評価指標を定めて、この可視化プログラムによって生成された図の可読性の評価を行うことで、本手法の有効性および定めた評価指標の妥当性を確かめた。

Abstract

Fluid dynamics is a major research field of dynamics that deals with the motion of gases and fluids. For a long time, it has been researched by discrete analysis and numerical analysis. In discrete analysis, by focusing on the topological point of view of the flow, it is possible to extract the essential structure of the flow, and to perform the analysis of the flow efficiently. Tree representation exists as a method of discrete analysis. It can analyze the flow by treating the structure of the flow algebraically. However, for non-experts, tree representation of the flow structure is difficult to intuitively understand. In this research, we proposed a method to draw the topology of the flow expressed using the tree representation on the diagram, and implemented it by Python and Asymptote, which is a programming language and the design of the visualization program according to this method. In addition, by establishing the evaluation index of readability, and evaluating the readability of the diagram generated by this visualization program, the validity of this method and the validity of the established index were confirmed.

目次

第 1 章　はじめに 1
- 1.1 トポロジーの背景 1
- 1.2 流体力学の背景 1
- 1.3 トポロジカルフローデータアナリシス 1
- 1.4 本研究の目的 1
- 1.5 アプローチ 2
- 1.6 役割分担 2

第 2 章　関連研究 3
- 2.1 構造安定な非圧縮流れの研究 3
- 2.2 語表現の研究 5
- 2.3 木表現の研究 6
- 2.4 その他の離散解析手法 6
- 2.5 流れの解析以外での離散解析の使われ方 6

第 3 章　自動可視化への準備 8
- 3.1 文法の定義 8
- 3.2 流れ構造の特徴 9
- 3.3 図が満たすべき条件 11
- 3.4 構文解析 12
- 3.5 デザインパターン 13
- 3.6 クラス図 14
- 3.7 Matplotlib 16
- 3.8 Asymptote 17

第 4 章　実装 18
- 4.1 Python による実装 18
- 4.2 Asymptote による実装 42

第 5 章　おわりに 49

参考文献 50

参考文献 50

付録 A　プログラムリスト 51

付録 B　実行例 70

第 1 章

はじめに

1.1 トポロジーの背景

流れをそのトポロジーに注目して解析する手法が知られている．トポロジカルには，連続的に変形できる図形は同じ形とみなされる．例えば，四角形と三角形は同じ形である．トポロジーに着目することは，詳細な情報は無視することになるが，データの本質的な特徴を見つけやすくなるなどのメリットがある．近年では，ビッグデータの解析などをトポロジーに着目して行うトポロジカルデータアナリシス（TDA）と呼ばれる解析手法が注目されている [8].

1.2 流体力学の背景

流体力学は，気体や液体の運動について取り扱う力学の主要な研究分野である．流体力学の歴史は古く，この学問が確立される以前から人々の生活に即したものとして発展してきた．近年では，コンピュータの発達により流体方程式から数値解を求め流れを精密に計算することが可能となった．この手法は数値解析と呼ばれ，事故や災害などの実験不可能な課題も計算できる利点を持った近年の主要な流体力学の解析手法である.

1.3 トポロジカルフローデータアナリシス

TDA は流体力学の分野においても有効な解析であることが知られている．トポロジカルフローデータアナリシスは流れをトポロジカルな観点で解析する．トポロジカルな観点に着目することは大域的な構造に着目するということを意味し，これにより流れの本質的な構造を抜き出すことが可能となる．この解析法の代表的なアプローチとして，流体をその構造安定性に着目して解析する方法がある．構造安定性は，力学系に小さな乱れが加わっても流れのトポロジーが変化しない性質のことである．構造安定性は一般の流体においては成立しないが，文献 [1] によれば，有限の精度や有限の時間における挙動に着目したり，本質的な構造に着目するなどの現実的な目的の範囲内であれば，現実の流体にもこの考え方が適用できる．構造安定性に着目すれば，構造安定性から流れの変化を考察することができるようになる．例えば，語表現の研究では流れのトポロジーに対応した文字列を定義することで，流体構造変化を文字列の変化として考察することが可能となった [2].

1.4 本研究の目的

本研究は，トポロジー表現の一つである木表現に対して，図上への可視化プログラムを作成する．ただし，本研究で扱う木表現は [3] によって与えられた手法である．また，以下で木表現はこの表現方法を指す．木表現は流線構造を代数的に扱い流れの解析を行うことができるが，その表現から直感的に二次元上の流れの形状を把握することは困難である．そのため，二次元の形状を得たい場合は解析者がその都度木表現を組み合わせて図示化することになるが，木表現が複雑になればなるほどその変換も煩雑になり，ともすれば途中で間違っ

た変換を行ってしまうこともあり得る．また，著作物などに木表現の図を挿入しようとする場合，描画ソフトを用いてそれを手作業で描く必要がある．本研究は木表現を図に自動的に変換する方法を与えることで先述したような手間を省き，効率的で確実な木文法による流れの解析を実現するものである．

1.5 アプローチ

本研究では，まず木表現描画プログラムが満たすべき条件を定め，また，クラス図を作成するなど，プログラムの要件定義を行った．そして，それをもとにプログラムを Python と Asymptote の 2 種類のプログラミング言語で実装した．Python と Asymptote では，クラス図などは同一であるがその実現方法は異なるため，それぞれに適した実装を行った．そして，作成したプログラムに対してテストを行い，それらの描画性能や描画限界について調査した．

1.6 役割分担

亀谷拓磨は主に Asymptote での描画プログラムの実装を担当し，2.4 節，2.5 節，3.2 節，3.8 節，4.2.1 節，4.2.2 節，4.2.3 節，4.2.4 節，4.2.5 節，4.2.6 節，5 章の Asymptote 部分，付録 A の Asymptote の説明文，付録 B の Asymptote の説明文を執筆した．田島嘉人は主に Python での描画アルゴリズムの作成と実装を担当し，第 1 章，および 2.1 節，2.2 節，2.3 節，3.3 節，3.7 節，4.1.1 節，4.1.2 節，4.1.3 節，4.1.4 節，4.1.7 節，付録 A の Python の説明文を執筆した．渡辺康平は主に構文解析とクラス図の作成を担当し，3.1 節，3.4 節，3.5 節，3.6 節，**??**，4.1.6 節，4.1.8 節，5 章の Python 部分，付録 B の説明文を執筆した．

第 2 章

関連研究

2.1　構造安定な非圧縮流れの研究

2.1.1　流線の構成要素

離散的な観点では，全ての静的な流れは流線と特異点の組み合わせによって構成される．特に，二次元多重連結領域上の構造安定な非圧縮流は，図 2.1 に示されている軌道 (a)，(b)，(e)，(f)，(i)，(g) と点 (c)，(d)，(e) の組み合わせのみによって表現できる．ここで，多重連結領域とは複数の障害物が含まれている領域のことである．図 2.1 中の S は，吸い込み湧き出し対を表している．吸い込み湧き出し対とは，流線が吸い込まれ，かつ，湧き出してくる場所のことである．また，グレーの部分は流線が発生しない場所を表している．流線が発生する場所と発生しない場所の境目を境界と呼ぶ．後述する流れの操作のために，図 2.1 中の軌道と点について，それぞれの名称を説明する．(a) は吸い込み湧き出し対から出て自分自身に戻る軌道であり，ss-orbit という．(b) は吸い込み湧き出し対から出て境界へとつながる軌道であり，ss-∂-saddle connection と呼ぶ．(c) は軌道と境界の交点であり，ss-∂-saddle と呼ぶ．(e) は境界上の点から出て同じ境界上の点につながる軌道であり，∂-saddle connection と呼ぶ．(d) は ∂-saddle と呼び，∂-saddle connection によって境界上とつながれている点である．(h) は境界上にない点であり，saddle point と呼ばれる．(f) は吸い込み湧き出し対から出て saddle point につながる軌道であり，ss-saddle connection と呼ばれる．(i) は saddle point から出て同じ saddle point に戻る軌道で homoclinic saddle connection と呼ばれる．(g) は境界や渦の周りを囲む閉じた軌道であり，closed orbit と呼ばれる．以後本論文では，流れの位相構造に影響しない ss-orbits と closed orbits は図に表示しない．

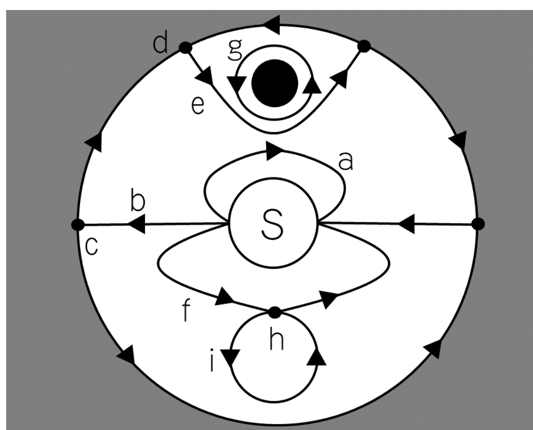

図 2.1　二次元多重連結領域内における構造安定な非圧縮流れの構成要素

2.1.2 構成要素を組み合わせる操作

全ての構造安定な流線は障害物と前節で述べた軌道と点を初期構造に帰納的に組み合わせることによって作図することができる．論文 [2] によって，これらの初期構造となる流線は 3 種類であり，以下に説明する 5 つの操作によってこれらの初期構造からすべての構造安定な流線を作図できることが証明されている．3 種類の初期構造を図 2.2 に示す．図中の (a) は吸い込み湧き出し対を持つ一様流である．(b) は，吸い込み湧き出し対を持ち，homoclinic saddle point と 2 つの ss-saddle connection を持つ．(c) は吸い込み湧き出し対を持たない構造安定な流れである．構造安定な流れに対して適用できる操作は以下の 5 つである．また，以下の操作による流線の変化を図 2.3 に示す．

- 1 つの ss-orbit を，homoclinic saddle connection と 2 つの ss-saddle connection に変換する操作．(A-I)
- 1 つの ss-orbit を，2 つの ss-∂-saddle connection と 2 つの ∂-saddles に変換する操作．(A-II)
- 1 つの closed orbit を，1 つの saddle point と 2 つの homoclinic 軌道に変換する操作．(B-I)
- 1 つの closed orbit を，2 つの ∂-saddles と 1 つの ∂-saddle connection 軌道に変換する操作．(B-II)
- すでに $2k(k > 0)$ の ∂-saddles を持つ境界に 2 つの ∂-saddles を付け加え，それらを 1 つの ∂-saddle connection でつなぐ操作．(C-I)

以下で説明する語表現と木表現は，これらの操作や初期構造に対しての表現方法であり，以上の研究の応用である．

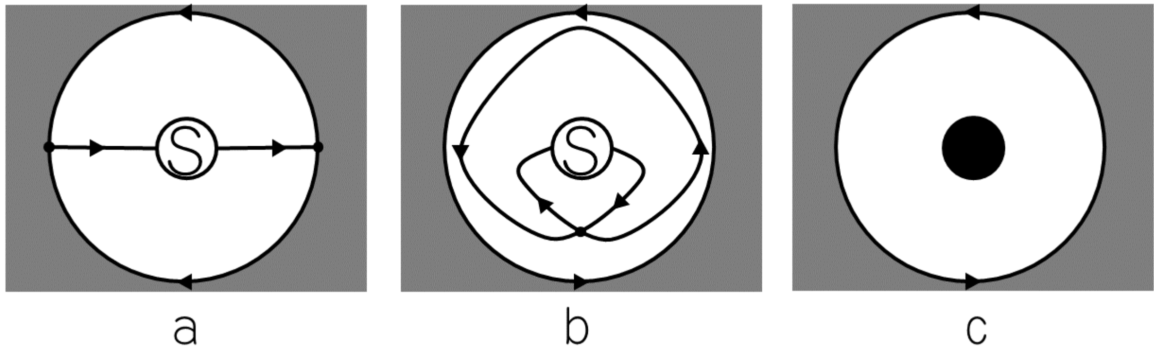

図 2.2　3 種類の初期構造

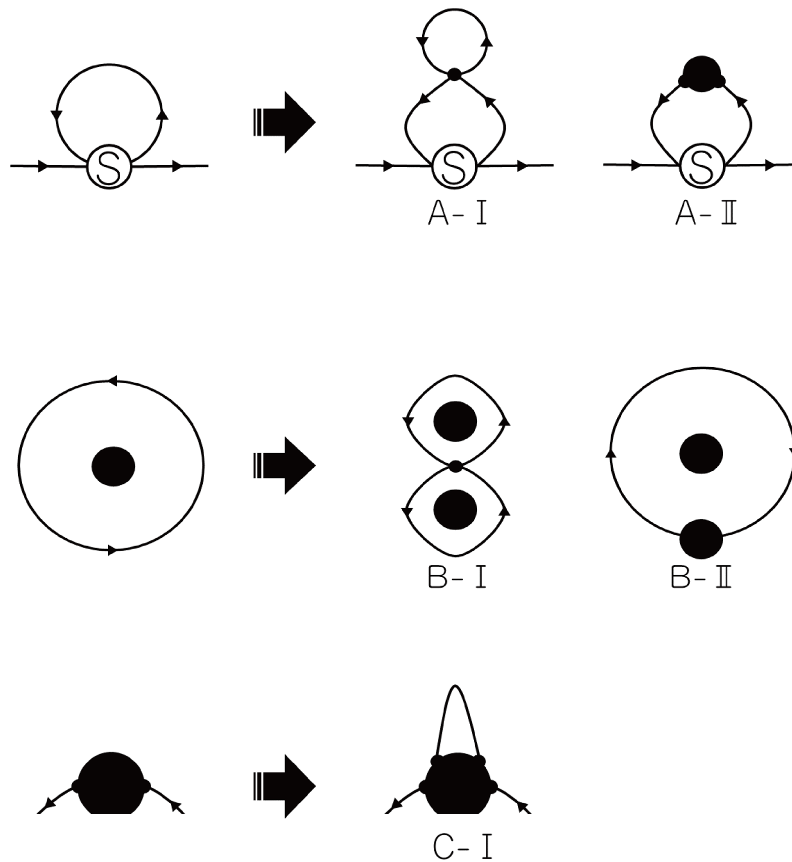

図 2.3　構造安定な流れに適用可能な 5 つの操作

2.2　語表現の研究

本研究で扱う木表現は，語表現の研究を発展させたものである．語表現の研究では，流線の位相的な構造を分類し，それぞれの流線構造を系統的に文字列で表現する方法を与えた [2]．語表現を用いた例として，翼の揚抗比の時間変化を語表現によって表した研究がある [4]．語表現は，有界な多重連結領域上で非圧縮かつ非粘性で構造安定な流れを代数的に表すことができる．非圧縮・非粘性という前提条件は現実の流体に対しての直接の適用に制限を与える．しかし，現実への応用に関しては文献 [4] でその適用方法が考えられている．流線構造を文字列で表現することで，流線構造を数学的に厳密に分類できるようになり流れの変動を特徴付けて捉えることができるようになる．また，代数的に扱うことができるため，流線の構造の特徴を説明するための共通言語として用いることもできる．文献 [2] によって与えられた語表現には同じ流線に複数の語表現を与えることができるという問題があった．そのため，文献 [5] によって自然な語表現を与えるアルゴリズムが与えられた．また，文献 [6] によって流れの向きを考慮した場合の語表現を与えるアルゴリズムが与えられた．

2.3　木表現の研究

木表現は木文法を用いて流線を表現する表現方法であり，その前提条件は語表現と共通である．木表現は先述した語表現の特徴に加えて，1 つの木表現に必ず 1 つの流れが対応するという特徴ある．このため，語表現より細かい流れの分類を可能とする．木表現は，流れの特徴を語表現より多く捉えることを期待されている．実際，円盤状の非圧縮流の反転の解析を木表現によって行った研究では，語表現と比べた木表現の表現力の高さが確かめられた [7]．木表現は表現力以外にも，形式言語理論の分野でよく使用される記法を用いているため，理解しやすいという特徴がある．

2.4　その他の離散解析手法

語表現および木表現と関連のある離散解析手法として，コンレイ・モース分解，グラフクラスタリングがある [1]．コンレイ・モース分解はモースが多様体上で解析している勾配流という関数の大きな値から小さな値へと流れるようなベクトル場と，コンレイの「任意の力学系は不変集合とそれらを結ぶ再帰性を持たない軌道に分解できる」という定理が組み合わさったものである．要するに勾配流を流れの停留点とそれを結ぶ軌道で分解するというものであるが，不変集合である停留点が無限個存在する場合は分解の計算が難儀になり，小さい誤差で構造が変化してしまう恐れがある．そこで解析に用いるグリッドを予め定め，グリッド以下の不変集合は無視することとしている．分解の手順として最初に力学の情報を有向グラフで表現する．図 2.4 の左図はグリッド内の不変集合の要素が他の要素への写像により表現されているが，グラフが大きいため構造情報が取り出しが難儀になっている．そこで，不変集合を一点にすることにより，グラフの構造を簡易化させて流れの構造の情報を取り出しやすくする．

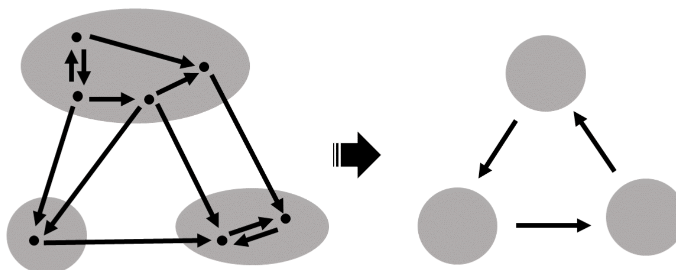

図 2.4　コンレイ・モース分解

グラフクラスタリングはグラフを部分グラフに分解しグラフの構造を分析するものである．近年ではデータマイニングなどで社会科学の発展も促している．離散的なこれらの手法は，流体に対してそれぞれ異なるアプローチを持つが，流体をその構造安定性に着目して解析するという点では一致している．

2.5　流れの解析以外での離散解析の使われ方

トポロジカルデータアナリシスという離散解析手法が存在する．これはデータをトポロジー的に形状を可視化し，データの特徴を捉えるというものである．現在，データは多次元なものが主流であり，従来の統計的な手法によって多次元データを低次元化する際に取り逃していた特徴も，この解析手法により特徴を捉えた可視

化が可能となる．また，多分野の発展に大きく関わるビックデータや IoT においても，この解析手法は扱われており，重要視されるデータ分析の精度の向上のために利用されている [8]．このように離散解析は流体力学以外にも多分野において扱われており，発展を支えていることがわかる．

第 3 章

自動可視化への準備

本研究では木表現を利用して可視化を行うため，文法を定義し，構文解析を行う．文法定義については [7] を参考にした．また，可読性の定義を行った．これは，可読性の評価指標を定めることで，提案した描画方法の有用性を評価するためである．本章では，可視化を行うために必要な準備について記述する．

3.1 文法の定義

BNF によって文法定義を行う．木文法は，$G = (S, N, F, R)$ によって定められる．S は開始記号，N は非終端記号の集合，F は終端記号の集合，R は生成規則である．ここで，$N = \{S, A_\emptyset, B_+, B_-, C_+, C_-, C_+^*, C_-^*\}$，$F = \{a_\emptyset, b_{\emptyset+}(,\{\}), b_{\emptyset-}(,\{\}), a_+(), a_-(), a_2(,), b_{++}\{,\}, b_{+-}\{,\}, b_{--}\{,\}, b_{-+}\{,\}, \beta_+\{\}, \beta_-\{\}, c_+(,), c_-(,), l, n, cons(,)\}$ とする．生成規則 R は以下のように表される．

$$
\begin{array}{lcl}
S & \rightarrow & a_\emptyset(A^*) \mid b_{\emptyset+}(B_+,\{C_-^*\}) \mid b_{\emptyset-}(B_-,\{C_+^*\}) \\
A & \rightarrow & a_+(B_+) \mid a_-(B_-) \mid a_2(C_+^*,C_-^*) \\
A^* & \rightarrow & n \mid cons(A,A^*) \\
B_+ & \rightarrow & l \mid b_{++}\{B_+,B_+\} \mid b_{+-}(B_+,B_-) \mid \beta_+\{C_+^*\} \\
B_- & \rightarrow & l \mid b_{--}\{B_-,B_-\} \mid b_{-+}(B_-,B_+) \mid \beta_-\{C_-^*\} \\
C_+ & \rightarrow & c_+(B_+,C_-^*) \\
C_- & \rightarrow & c_-(B_-,C_+^*) \\
C_+^* & \rightarrow & n \mid cons(C_+,C_+^*) \\
C_-^* & \rightarrow & n \mid cons(C_-,C_-^*)
\end{array}
$$

以上の木文法によって生成された木は，流線の形状を表す．開始記号 S は 3 つの基本パターン $a_\emptyset$，$b_{\emptyset+}$，$b_{\emptyset-}$ からなり，これは流れの初期構造に対応する．この基本パターンを図??に示す．$a_\emptyset$ は一様流を表し，$b_{\emptyset+}$，$b_{\emptyset-}$ はともに円盤状の流れで最外境界部をもつ流線を表す．また，木文法では $+$ が反時計回り，$-$ が時計回りの流れを表す．

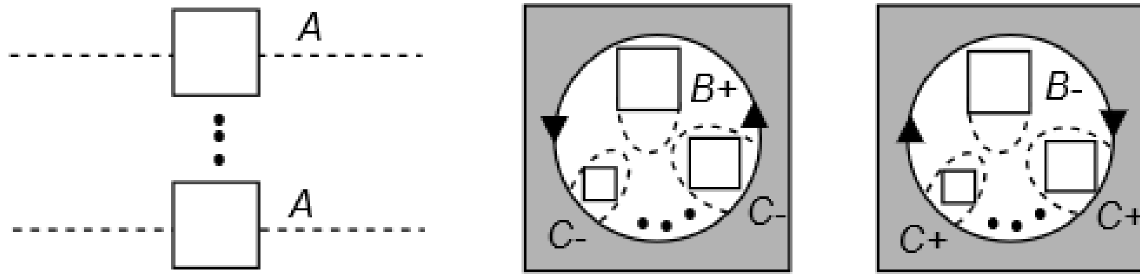

図 3.1　3 つの基本パターン　左から a_ϕ，$b_{\phi+}$，$b_{\phi-}$

そのほかにも，A 系の流れ，B 系の流れ，C 系の流れの構造が存在する．基本パターンに，これらの構造を生成規則に則り，組み合わせることで流線図を表現する．本研究では，木表現を確実に自動変換することが目的であるため [3] で考えられた生成規則の円順列 {} については考えないものとする．

3.2　流れ構造の特徴

3.2.1　A 系

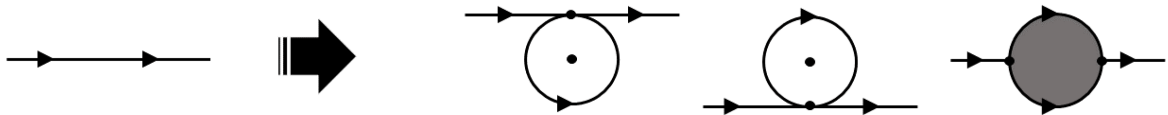

図 3.2　A 系の流れ

A 系は一つの無限軌道で流れる一様流を変換した構造の集合である．これを図 3.2 に示す．a_+ と a_- は一本軌道上に一つの saddle point と，そこから出て，同じ saddle point に戻る一つの homoclinic saddle connection を持った構造である．左からの流れる軌道の場合，a_+ と a_- はそれぞれの homoclinic saddle connection が反時計回りと時計回りで区別される．文法規則上では，homoclinic saddle connection の中や線上に B 系の変化が適用される場合がある．a_+ には b_{++}，b_{+-}，b_+ の a_- には b_{--}，b_{-+}，b_- が適用される可能性があるため，整合性を保つようにプログラムを作成する必要がある．a_2 は一様流上に物理境界が存在するような構造である．軌道は物理境界の周りをふた手に沿って進み，再び合流し一本の軌道に収束する．a_2 は文法規則上，一つ以上の C 系が物理境界の周りに形成される可能性がある．

3.2.2 B 系

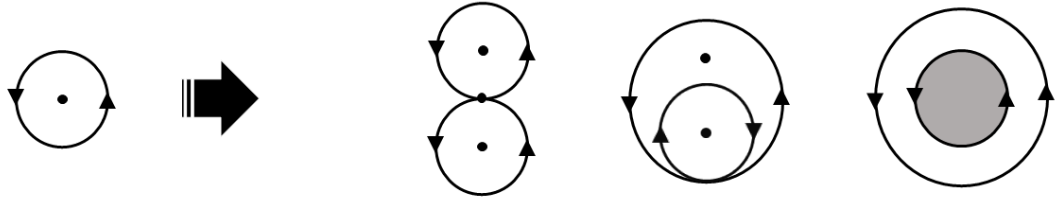

図 3.3　$b+$ 系の流れ

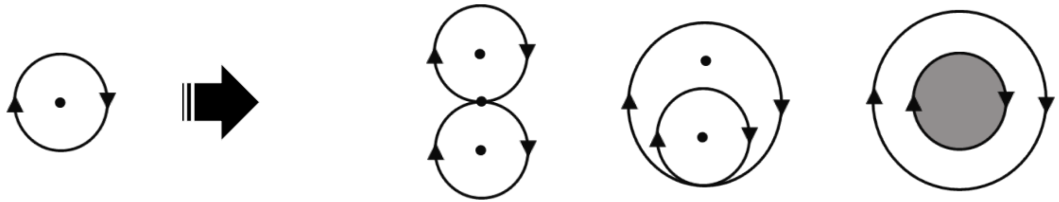

図 3.4　$b-$ 系の流れ

B 系は $b_{\phi+}$ と $b_{\phi-}$ の一部である反時計回りの閉軌道と時計回りの閉軌道から変換した構造の集合である．これを図 3.3，図 3.4 に示す．反時計回りの閉軌道の $b_{\phi+}$ は b_{++}，b_{+-}，b_{+} が，時計回りの閉軌道 $b_{\phi-}$ は b_{--}，b_{-+}，b_{-} に変換される可能性を持つ．b_{++} は 2 つの $b_{\phi+}$ が外接したような流線構造であり，対称的に b_{--} は 2 つの $b_{\phi-}$ が外接したような流線構造である．b_{+-} は $b_{\phi-}$ が $b_{\phi+}$ に内接しているような構造であり，対象的に b_{-+} は $b_{\phi+}$ が $b_{\phi-}$ に内接しているような構造である．b_{++} と b_{--} のそれぞれの外接点と，b_{+-} と b_{-+} のそれぞれの内接点は saddle point となり，4 つの流線の特徴として 1 つの saddle point に 2 つの homoclinic saddle connection が形成されていることが挙げられる．b_{+} と b_{-} は，それぞれ $b_{\phi+}$ と $b_{\phi-}$ の流線上に一つの物理境界が形成され，軌道はふた手に別れ物理境界の周りを沿って進み，再び一本の軌道に収束して永続的に周回する．その際，b_{+} と b_{-} には一本軌道が物理境界上にふた手に発散する場所と，再び収束する場所にはそれぞれ不動点が形成される．すなわち，その 2 つの点を弧を描いて結ぶような軌道で進む ∂-saddle connection が形成されると捉えることは自明である．前述で記述したように A 系の構造上に B 系が適用される可能性があるが，B 系の構造上にも適用される可能性がある．b_{++}，b_{+-}，b_{+} の変換を $B+$ とし，b_{--}，b_{-+}，b_{-} の変換を $B-$ とすると，b_{++} にはそれぞれの homoclinic saddle connection に $B+$ の変換が，そして対象的に b_{--} には $B-$ の変換が適用される可能性がある．b_{+-} と b_{-+} には，反時計周りの homoclinic saddle connection に $B+$ が，時計回りの homoclinic saddle connection に $B-$ の変換が適用される可能性がある．b_{+} の物理境界上に c_{+}，b_{-} の物理境界上に c_{-} が適用される可能性がある．

3.2.3 C 系

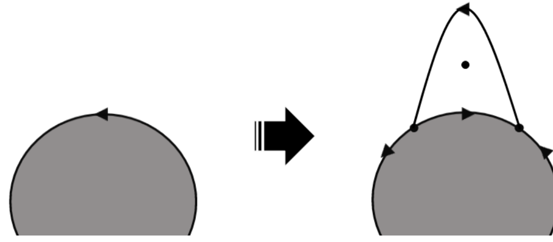

図 3.5 $c+$ の流れ

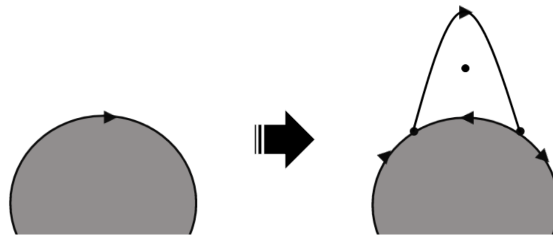

図 3.6 $c-$ の流れ

C 系は物理境界の周辺を流れる流線上に ∂-saddle connection が作られた構造の集合である．c_+ は反時計回り，c_- が時計回り ∂-saddle connection である．これを図 3.5，図 3.6 に示す．作られた閉軌道には他の流線構造が適用される可能性がある．c_+ には上記で示した $B+$ と c_-，c_- には $B-$ と c_+ が適用される．

3.3 図が満たすべき条件

本研究では，木表現から 2 次元上のトポロジーへの自動変換手法を考案し，実装する．この節では，実装したプログラムがどのような図を作図すれば本研究の目的を達成できたと言えるか定める．本研究では以下の 3 つの指標を満たすことができれば，プログラムの利用者によって見やすくトポロジーの検証に便利であり，本研究の目的を満たすことができたプログラムであるとする．

- 線が重ならず交差しない
- 線の間が適切な距離を保つ
- 線が滑らかである

実装したプログラムは少なくとも木表現を 2 次元上のトポロジーに変換した際に，その両方が確実に同一のものを再現している必要がある．例えば，木表現が $a_\emptyset(cons(a_+(b_{++}(l,l),l))),n))$ であれば，2 次元上のトポロジーも $a_\emptyset(cons(a_+(b_{++}(l,l),l))),n))$ を表していなければならず，間違っても $a_\emptyset(cons(a_-(b_{--}(l,l),l))),n))$ ではならない．線が重なったり交差してしまうと，それは期待するトポロジーとは異なるものを示す．

トポロジーを表す線同士の適切な距離は，主観的であるが，あまりに離れているように感じられると図の見栄えが悪くなる．逆に，近づきすぎていても同様である．

線の滑らかさも線同士の距離に同じく，角ばっていたりなどするとトポロジーの判定には影響が出ないが，見栄えが悪くなる．

3.4　構文解析

本研究では Python での実装を行うため，Python 専用の構文解析ライブラリ PLY を利用し構文解析を行った．PLY とは，Python Lex-Yacc の略であり，Python 専用の構文解析ライブラリであり，lex モジュールと yacc モジュールからなる．lex モジュールでは，字句を tokens で定義し，定義した字句に対応させたい正規表現を tokens に代入することで，字句解析を行っていく．

```
tokens=(
  ' 字句',
  …
)
t_字句=r' 正規表現'
```

本研究では，3.1 節において定義された終端記号を正規表現で定義する．字句に正規表現を代入することで字句解析を行う．また，改行と余白は無視するため，改行と余白は t_ignore に代入する．例としてプログラムの一部をプログラム 3.1 に示す．ここで代入される正規表現については実行例で説明する．

プログラム 3.1　lex.py

```
1  import ply.lex as lex
2  tokens=(
3    'NIL' ,
4    'A0' ,
5    'LPAREN' ,
6    'RPAREN' ,
7    'COMMA' ,
8    'A_PLUS' , …
9  )
10 t_ignore = '\t\n'
11 t_NIL = r'n'
12 t_A0 = r'a0'
13 t_LPAREN = r'\( '
14 t_RPAREN = r'\) '
15 t_COMMA =r','
16 t_A_PlUS = r'a\+' …
```

その他にも，t_error() でエラー処理を行い，lex.lex() でオブジェクトの生成を行う．

一方，yacc モジュールでは，lex モジュールによって定義された字句を読み込み，文脈自由文法によって定義された構文を評価して構文解析を行い抽象構文木を作成する．lex モジュールにより定義された字句を読み込み，読み込んだ字句からなる木構造を定義していく．

```
import ply.yacc as yacc
from lex モジュール import tokens
def p_関数名 (p) :
    ' 文法規則'
    p[ ]= …
```

それぞれの関数は一つの引数 p を受け取り，引数 p はルールと一致した字句の値を持っている．定義された引数は，連続した値になっており，左から順番に定義される．そして，文法規則に則って木構造を生成していく．本研究では，yacc モジュールが LALR 法を用いて構文解析を行っているため，字句ごとにクラスを作成した．例としてプログラムの一部をプログラム 3.2 に示す．

プログラム 3.2 yacc.py

```
1  import ply.yacc as yacc
2  from lex import tokens
3  import sys
4  import flow
5  def p_exper_a0(p):
6        ' s : A0 LPAREN as RPAREN'
7        p[0] = flow.A0(p[3])
8  def p_exper_b0_plus(p):
9        ' s : B0_PLUS LPAREN b_plus COMMA LPAREN cs_plus RPAREN RPAREN'
10       p[0] = flow.B0_plus(p[3],p[6])
11 def p_exper_b0_minus(p):
12       ' s : B0_MINUS LPAREN b_minus COMMA LPAREN cs_minus RPAREN RPAREN'
13      p[0] = flow.B0_minus(p[3],p[6]) …
```

lex.py によって定義されていないトークンから枝が派生され，それぞれが非終端記号を表している．文法ルールが一致した場合，flow.py に存在する各クラスを呼び出し抽象構文木の生成，図の大きさの定義を行う．上記で記述したプログラムは，3.1 節の開始記号 S から始まる文法定義を表しており，その他の非終端記号も同様にして記述していく．実際の流線の描画は flow.py の各クラスで描画していく．

3.5 デザインパターン

ソフトウェア設計者の経験から同じような問題は，典型的に同じパターンの解決策になることが発見され，それらのパターンをカタログ化したものをデザインパターンという．オブジェクト指向において，様々なプログラムで利用でき，目的に応じて 23 種の中から選択して扱うことができる．主にオブジェクト指向言語で再利用性が高いクラスやライブラリを作成する際に利用される．本研究では，デザインパターンの中のインタプリタパターンを取り扱う．インタプリタパターンについては，[9] を参考にした．一般的なインタープリタパターンを図 3.7 に示す．インタプリタパターンは，形式的に記述された記号列を解析した結果に則って処理したい場合に利用されるデザインパターンである．目的に応じた言語を作成することで，通訳の意味を果たすプログラムを用意することにより，素早く処理することが可能になる．再帰的な構造を持ち，全ての要素に共通な処理を持つ抽象クラスを定義することで構造の変更を容易にしたり，見た目をシンプルにすることが可能である．本研究では，木文法より定義された生成規則によって，字句解析と構文解析を行い，構文木を作成する．そのため，構文木に則った処理を実現するのに最適なインタプリタパターンを利用した．また，作成するクラスが多いため，オーバーライドを利用することで機能の追加が容易になり，修正を減らすことができる．

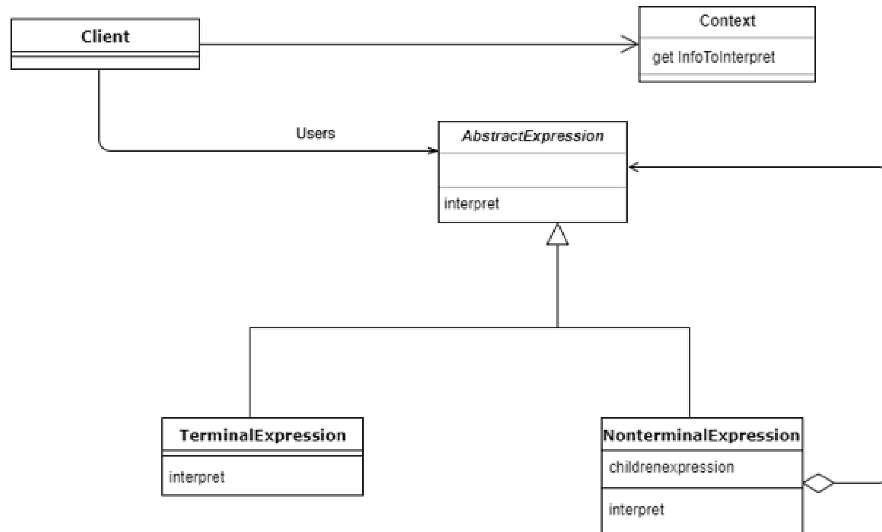

図 3.7　インタープリタパターン

3.6　クラス図

Python と Asymptote で実装するために，図 3.8 を作成した．クラス図は 18 のクラスからなり，抽象クラスとして，Node クラスを定義し, 非終端クラスと終端クラスに共通するインターフェースを定義する．残りのクラスは，主に描画する流線パターンに対応している．また, 流線パターン以外にも存在し,Cons クラスによりリストの連接を表す．また，非終端クラスの集合 {A0，B0_plus，B0_minus，A_plus，A_minus，A2，Cons，B_plus_plus，B_plus_minus，Beta_plus，B_minus_minus，B_minus_plus，Beta_minus，C_plus，C_minus}，終端クラスの集合 {Nil，Leaf} とする．

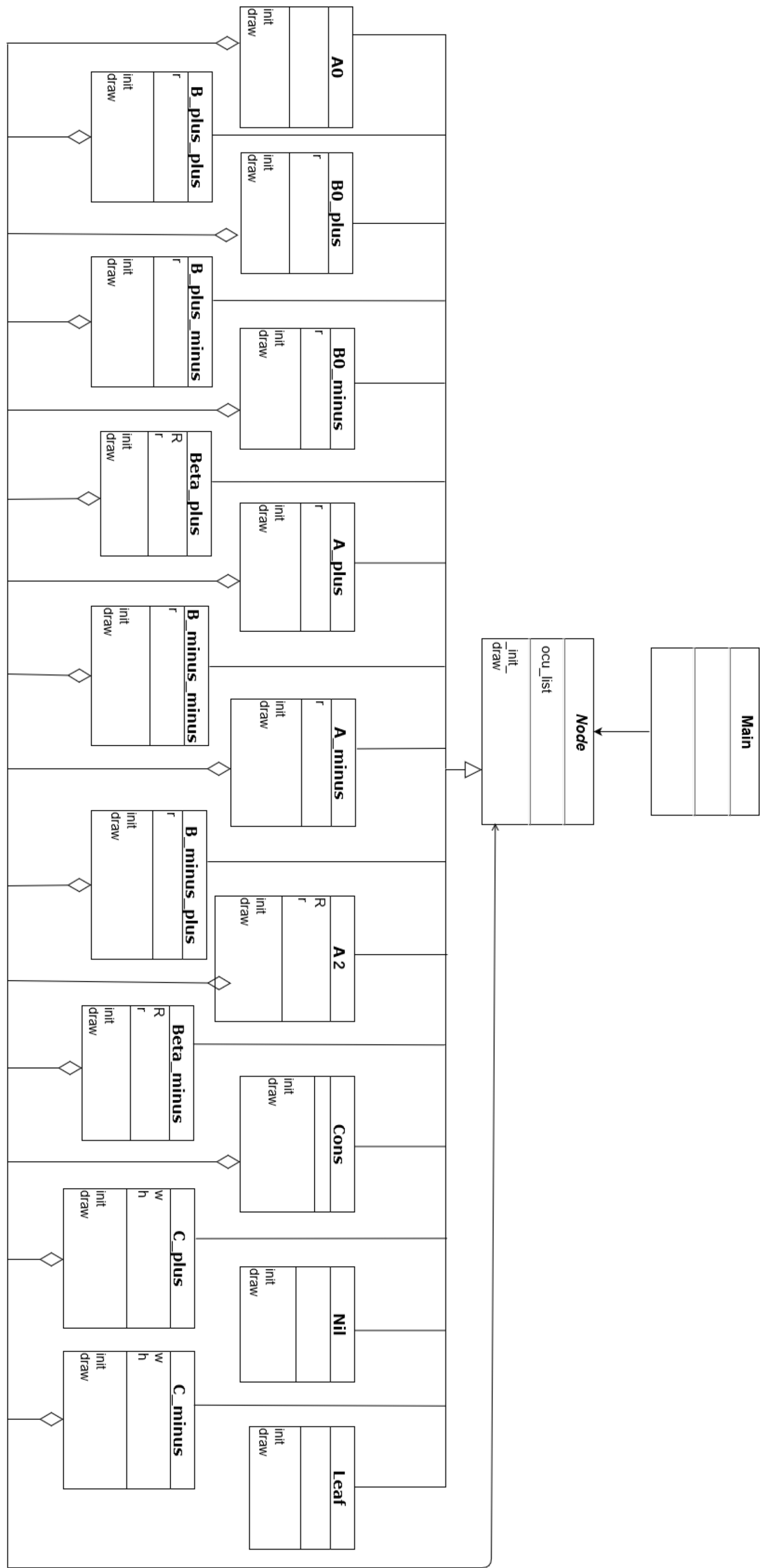

図 3.8　クラス図

- Node クラス
  図 3.7 の Abstractexpression の部分に相当し，各クラスが継承している．Node クラスを継承するクラスは操作に, 描画を行い，構文木を作成する関数を持っている．そのため，Node クラスに継承するクラスに，共通するメソッド名を定義することによってオーバーライドを行い，機能を上書きできるようにした.

- A0，B0_plus，B0_minus クラス
  これらは 3.1 節の非終端記号の S を表すクラスの集まりである.
- A_plus，A_minus，A2 クラス
  これらは 3.1 節の非終端記号の A を表すクラスの集まりである.
- Nil，Cons クラス
  これらは 3.1 節の非終端記号の A^*，C_+^*，C_-^* を表すクラスの集まりである.
- Leaf，B_plus_plus，B_plus_minus，Beta_plus クラス
  これらは 3.1 節の非終端記号の B_+ を表すクラスの集まりである.
- Leaf，B_minus_minus，B_minus_plus，Beta_minus クラス
  これらは 3.1 節の非終端記号の B_- を表すクラスの集まりである.
- Nil，C_plus
  これらは 3.1 節の非終端記号の C_+^* を表すクラスである.
- Nil，C_minus
  これらは 3.1 節の非終端記号の C_-^* を表すクラスである.

これらのクラスは図 3.7 の TerminalExpression，NonterminalExpression の部分に相当し，Node クラスと集約関係を持つ.

- Main クラス
  最初に呼ばれるクラスであり，文字列の入力，別のクラスのオブジェクトの作成，流線図の保存や表示を行う．利用者とプログラムをつなぐ役割を果たす.

3.7 Matplotlib

Python では流線図を描画するにあたり，Python 用のグラフ描画ライブラリである Matplotlib を使用する．Matplotlib はグラフ描画ライブラリであるが，棒グラフや折れ線グラフなどの一般的なグラフの描画だけでなく，平面上への関数・図形の描画や波のシミュレーションなど様々な描画を行うことができる．例えば，木表現 $b_{++}\{l,\ l\}$ を模した図を Matplotlib を用いて作図しようとする場合，プログラム 3.3 節のように記述することで図 3.9 が出力される (このプログラムでは，例を示すため明示的に円の中心と半径を示し描画を行った．また，プログラム簡略のため矢印と停留点を描画していない．)．このプログラムの 7 行目までは描画のための設定であり，描画領域として高さ・幅ともに 100 の正方形を用意している．また，draw_circ は円を描画する関数であり，描きたい円の中心の座標と半径を引数にとる．ここでは (x,y) 座標をそれぞれ $(50, 30)$，$(50, 70)$ とする 2 つの円を描画することで $b_{++}\{l,\ l\}$ を表現した．描画した画像は savefig で保存することができる．この例では，EPS 形式で保存した.

プログラム 3.3　$b_{++}\{l,\ l\}$

```
1 import matplotlib.pyplot as plt
2
```

```
3 fig = plt.figure()
4 ax = fig.add_subplot(111,aspect='equal')
5 W,H = 100,100
6 ax.set_xlim([0,W])
7 ax.set_ylim([0,H])
8
9 def draw_circ(center,r):
10     circ=plt.Circle(center,r,ec="black",fill=False)
11     ax.add_patch(circ)
12
13 draw_circ((50,30),20)
14 draw_circ((50,70),20)
15
16 fig.savefig("b++.eps")
```

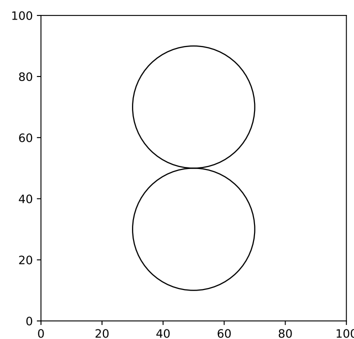

図 3.9　$b_{++}\{l,\ l\}$

3.8　Asymptote

Asymptote はベクタグラフィックのプログラミング言語である．ベクタグラフィックはビットマップ画像とは違い，線や輪郭など情報を解析幾何学的な図形として論理的に持たせているため，画像を拡大や縮小を行う際にも画質の品質は保たれる．よって，内部にループして入り続ける可能性もある流線構造であっても拡大すれば形が保たれるため，3.3 節の条件を満たすプログラミング言語として本研究で扱うこととした．ベクタグラフィックである PostScript による出力をデフォルトとしているが，ImageMagick が生成できる画像フォーマットであれば任意の出力を可能としている．また．Asymptote は CUI と GUI の機能も搭載しており，他のグラフィックプログラムよりも高水準なプログラミング言語とされている．TeX ディストリビューションの一つである TeXLive にはあらかじめ搭載されており，ラベルや数式を LaTeX に組版することも可能である．

第 4 章

実装

4.1　Python による実装

4.1.1　整合性のとれた図を作成するためのアルゴリズムの設計

自動描画で木表現のトポロジーを 2 次元上に再現するためには，それぞれのトポロジーの形を正確に描画することも求められるが，さらにすべてのトポロジーの位置関係の整合性を保つ必要がある．このような方法は直感的にはいくつか考えられる．例えば，木表現を根からたどりながら逐次的に描画する方法である．ただこの方法では図 4.1 のように，ある子ノードをたどっていくとその子ノードに多くの子孫ノードが繋がっており，出来上がった図が不自然に小さい図になってしまう可能性がある．逆に，葉からたどる方法も考えられる．しかし，この方法でも図 4.2 のように根までの大きさの整合性は保つことができるが，位置関係の整合性を保てない．そこで，私たちは葉から木表現をたどり大きさを決定し，描画は根から行う方法を考えた．この方法では，描画の位置を決めずにすべての部分木に対応する流線図の大きさのみを決定し，その大きさをもとに根から位置を決定し描画していくことで位置と大きさの整合性をとれた図を作図できる．例として，$a_{\phi}(cons(a_{+}(b_{++}(b_{++}(l,l),l))),n))$ という木表現を上げる．これを木表現で表した図が図 4.3 である．この木表現の大きさを決める順は図 4.4 のようになる．$leaf$ は大きさを持たないので b_{++} から，続いて a_{+}，a_{ϕ} の順で大きさが決まる．描画を行う際は，既にすべてのトポロジーの大きさには整合性があるので，根からそれぞれのトポロジーを図 4.5 のように親のトポロジーの枠内に収まるよう適した場所に描画すればよい．

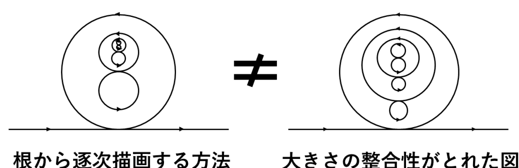

図 4.1　根から逐次描画するデメリット

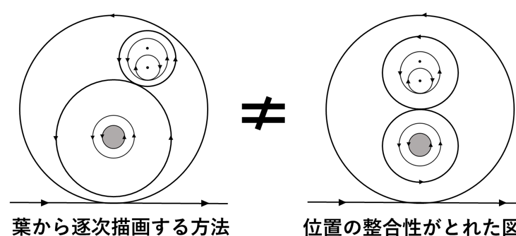

図 4.2　葉から逐次描画するデメリット

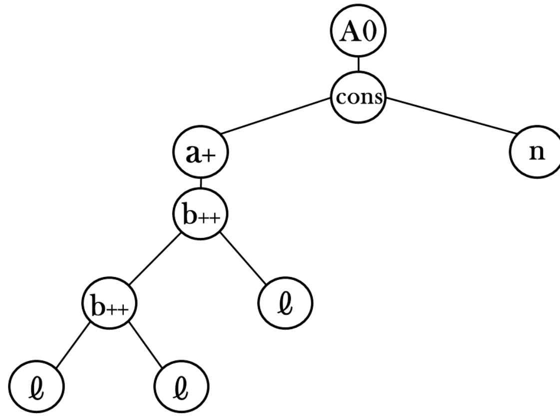

図 4.3　木表現 $a_\phi(cons(a_+(b_{++}(b_{++}(l,l),l))),n))$

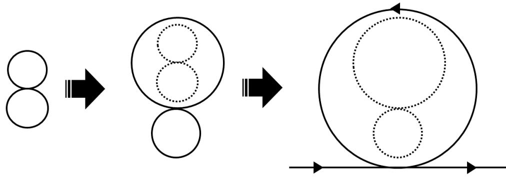

図 4.4　大きさを決定する手順

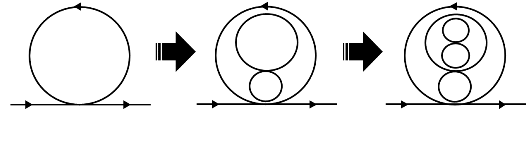

図 4.5　描画する手順

ここで，このアルゴリズムを実際にプログラムにするにあたって子ノードから親ノードに正確に大きさを伝達する方法を考える必要がある．例えば，図 4.4 では，a_+ の大きさを決めるためには b_{++} の大きさを a_+ に伝達する必要がある．そこで，伝達する方法としてすべてのトポロジーに画一的な指標を定義する．これを占有領域と呼ぶ．この占有領域を子ノードから親ノードに伝達し親ノードの大きさを決定する．占有領域は円で表す．これは，ほとんどのトポロジーが homoclinic saddle connection を持っているため円状で表すことが容易だからである．図 4.6 にこれらの占有領域の概要を示す．ここで，図中の a_+，a_- の一様流は占有領域に含まれない．ただ，例外として C 系のみは円状で表すことが難しい．そこで C 系の占有領域は長方形とする．C 系の占有領域の概要を図 4.7 に示す．この占有領域を用いた先ほどの $a_\phi(cons(a_+(b_{++}(b_{++}(l,l),l))),n))$ の大きさの決定手順は図 4.8 のようになる．

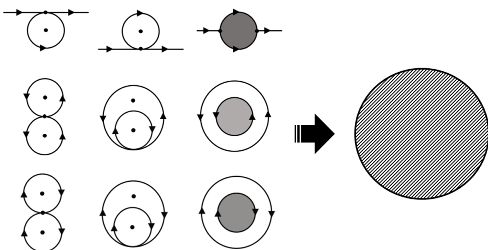

図 4.6　円で表すことができる占有領域

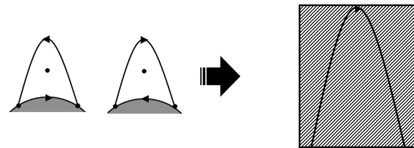

図 4.7　C 系を表す占有領域

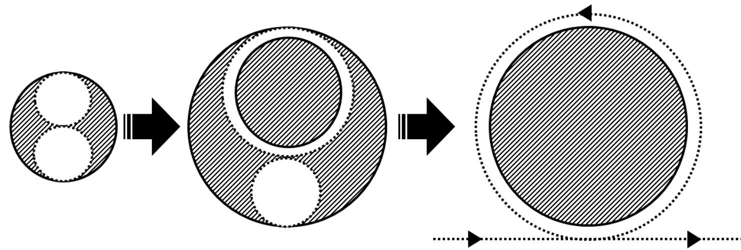

図 4.8　占有領域による大きさの決定

4.1.2　流線のトポロジーの表現方法

各トポロジーを 2 次元上の図として再現するためには，各トポロジーを 2 次元上に表現可能な図形でモデル化する必要がある．homoclinic saddle connection や物理境界は円形であるため，円でモデル化できる．したがって，これらで構成されているすべての A 系と B 系のトポロジーは円を組み合わせることで表現できる．しかし，C 系のトポロジーは基本的な形の組み合わせで表現することは難しい．そこで，C 系はそのトポロジーを図 4.9 のように点でモデル化し，この点を結ぶような滑らかな線を描画することで表現する．図中では右下の点，左下の点，最も上にある点の 3 点を補完している．また，本研究ではこの線の描画に，一般的な補間方法であるスプライン補間を用いた．

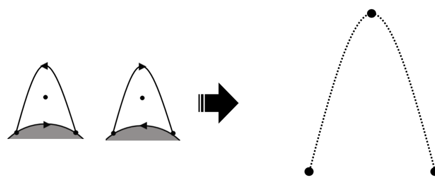

図 4.9　C 系のモデル化

4.1.3　個々の大きさ決定アルゴリズム

それぞれのトポロジーの大きさはその子の占有領域の大きさに合わせて決められるが，その大きさの決定方法は各トポロジーの種類によって異なる．この節では，これを実装する言語はオブジェクト指向言語であることを前提として，大きさ決定のアルゴリズムについて言語によらない一般的なアルゴリズムを説明する．本研究では以下のアルゴリズムの説明のために，木表現においてあるノードにおける左下の子ノードを「$head$」，右下の子ノードを「$tail$」と呼ぶ (子ノードが 1 つの場合は，1 つめの子ノードを「$head$」と呼ぶ．).

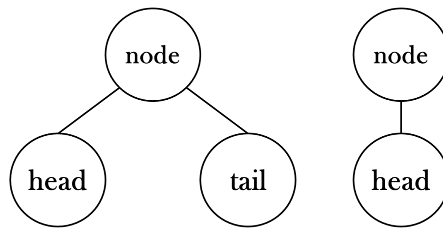

図 4.10　あるノードに対する子ノードの呼び方

このアルゴリズムは，3.6 節のクラス図に対応する．各トポロジーはそれに対応するクラスのインスタンスとして作成され，作成 (__init__メソッド) 時にインスタンス変数としてその大きさと占有領域の情報を作成し，また，持つ．これらの情報を持つためには，__init__メソッドに引数として子ノードの占有領域の情報が渡される必要がある．占有領域は円で表せられるので，多くのクラスでは引数として占有領域の半径が渡されるが，A0 クラスなどの一部のクラスではそのトポロジーの特性上それだけでは大きさを決められないため，独自のデータ構造が用いられる．また，大きさも占有領域も持たないクラスに Cons クラスがある．Cons クラスは *cons* を扱うクラスであり，*cons* は子ノードをまとめるためのみに用いられる．以下のアルゴリズムでは，a_ϕ や b_ϕ，C 系などがその大きさを決める際，子ノードを除く子孫ノードの情報を得ようとすると処理が煩雑になってしまうため，Cons クラスに *cons* に繋がる子ノードの情報をインスタンス変数として保持することにする．言い換えると，A や C 系がその占有領域の情報を親ノードに伝えたいときは Cons クラスを経由して伝えることになる．よって，Cons クラスの__init__メソッドはアルゴリズム 1 になる．また，このアルゴリズムの変数の流れを図式化したものが図 4.11 である．*cons* の *head* は必ず A 系か C 系である．また，*tail* は *cons* であり，この *cons* にも *tail* 以下で繋がれた同一の高さに描画すべき A 系か C 系が子ノードとして繋がれている．Cons クラスの処理は，親ノードで同一の高さに描画すべき子孫ノードの占有領域の情報をまとめ，自身のインスタンス変数として保持することである．

アルゴリズム 1 Cons クラスの__init__メソッド

条件:

$children1$：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数)
$children2$：$tail$ の占有領域の情報 ($tail$ クラスのインスタンス変数・リスト型)
$data$：head と tail の情報をリスト型として保持するインスタンス変数

1: **procedure** __INIT__(children1,children2)
2: 　　$data \leftarrow [children1, children2]$

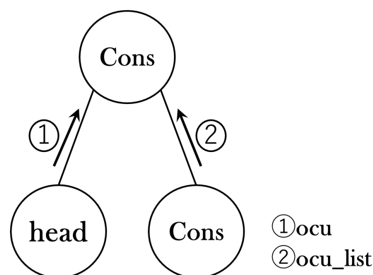

図 4.11　Cons の__init__処理に用いる情報

a_ϕ の大きさ決定

a_ϕ のトポロジーは，A0 クラスによって扱われる．a_ϕ は開始記号であり，大きさを決める際は一番最後にその大きさが決まる．しかし，a_ϕ は $cons$ で繋がれた A 系の数に合わせて一様流を描く特性上，自身の大きさを決める必要がなく，子の情報はその描画時にのみ使用する．また，親ノードを持たないため自身の占有領域を親ノードに渡す必要がなく，その占有領域を持たない．そのためこのクラスの__init__メソッドの処理は $cons$ の占有領域の情報をインスタンス変数に保持することに留める．$A0$ に必要な占有領域の情報は，占有領域の半径と A 系の種類である．A 系の種類を必要とするのは，その種類によって一様流の描画を変えるためである．例えば，a_2 を描画するためには，a_2 の占有領域の部分のみ一様流を描画しない．よって，A0 クラスの__init__メソッドはアルゴリズム 2 である．また，このアルゴリズムの変数の流れを図式化したものが図 4.12 である．

アルゴリズム 2 A0 クラスの__init__メソッド

条件:

$children$：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数・リスト型)

$data$：$head$ の占有領域の情報を保持するためのインスタンス変数

```
1: procedure __INIT__(children)
2:     data ← children
```

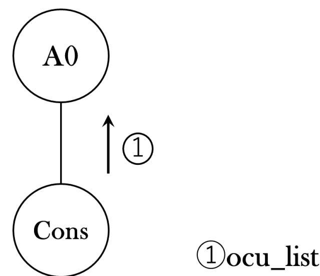

図 4.12　A0 の__init__処理に用いる情報

$b_{\phi+}$ 及び $b_{\phi-}$ の大きさ決定

$b_{\phi+}(b_{\phi-})$ のトポロジーは，B0_plus(B0_minus) クラスによって扱われる．$b_{\phi+}$ は開始記号であり，大きさを決める際は一番最後にその大きさが決まる．大きさの情報はインスタンス変数として保持され，描画の際に用いられる．また，a_ϕ と同様に親ノードを持たないため，自身の占有領域を決める必要がない．$b_{\phi+}$ の大きさの決定に必要な占有領域の情報は $head$ の占有領域の情報 (B 系の占有領域の半径) と $tail$ の占有領域の情報 ($cons$ で繋がれた C 系の情報) である．C 系の占有領域は長方形で表されるため，占有領域の情報は底辺長と高さの情報をまとめたタプル型 (占有領域の高さ, 占有領域の底辺長) である．このタプル型をまとめたリストが $tail$ の $cons$ にインスタンス変数として保持されている．B0_plus の__init__メソッドはアルゴリズム 3 である．このアルゴリズムの変数の流れを図式化したものが図 4.13 である．また，このアルゴリズムによる大きさを図式化したものが図 4.14 である．

アルゴリズム 3 B0_plus(B0_minus) クラスの__init__メソッド

条件:

R：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数・実数型)

$children$：$tail$ の占有領域の情報 ($tail$ クラスのインスタンス変数・リスト型)

r：自身の大きさを表す円の半径を保持するためのインスタンス変数

関数 HIGH_C は C 系の情報をまとめたリストから最も高い C 系の占有領域の高さを返す関数

関数 LENGTH_C は C 系の情報をまとめたリストから C 系が接する物理境界の円周を求める関数

```
1: procedure __INIT__(R, children)
2:     h ← HIGH_C(children)
3:     circumference ← LENGTH_C(children)
4:     if circumference ÷ (2 × π) > R + h + 1 then
5:         r ← circumference ÷ (2 × π)
6:     else
7:         r ← R + circumference + 1
```

アルゴリズム 4 関数 HIGH_C

条件:

$children$：C 系の情報をまとめたリスト ([(占有領域の高さ, 占有領域の底辺長), ...])

保証:

返り値 h は最も高い占有領域の高さを表す

```
1: function HIGH_C(children)
2:     h ← 0
3:     for i = 0 . . . children.length do          ▷ children リストの要素数だけ繰り返す
4:         child ← children[i]                       ▷ children の i 番目のタプルを取り出す
5:         if h < child[0] then                      ▷ child[0] が占有領域の高さに対応する
6:             h ← child[0]
7:     return h
```

アルゴリズム 5 関数 LENGTH_C

条件:

$children$：C 系の情報をまとめたリスト ([(占有領域の高さ, 占有領域の底辺長), ...])

保証:

返り値 $circumference$ は C 系が接する物理境界の円周を表す

```
1: function LENGTH_C(children)
2:     circumference ← 0
3:     l ← 0
4:     for i = 0 ... children.length do
5:         child ← children[i]
6:         circumference ← circumference + child[1] + 1    ▷ child[1] が占有領域の底辺長に対応する
7:         if l < childr[1] then
8:             l ← child[l]
9:     if circumference ÷ 2 <= l then
10:        circumference ← l × 2
11:    return circumference
```

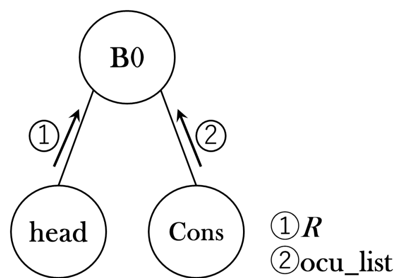

図 4.13　B0_plus の__init__処理に用いる情報

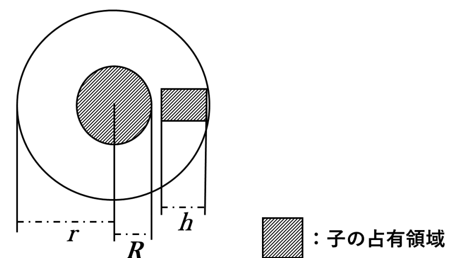

図 4.14　B0_plus の大きさのイメージ

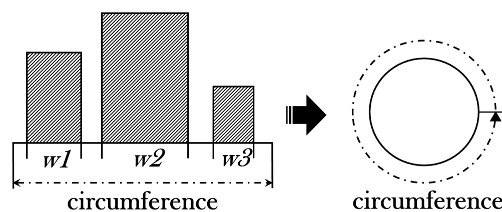

図 4.15　関数 LENGTH_C が C 系の情報をまとめたリストから物理境界の円周を求めるイメージ

a_+ 及び a_- の大きさ決定

$a_+(a_-)$ のトポロジーは，A_plus(A_minus) クラスによって扱われる．a_+ は非終端記号であるので，このトポロジーは親ノードと子ノードの両方を持つ．そのため，大きさを決める際は子ノードの占有領域の大きさをもとにその大きさを決め，その大きさを占有領域として親ノードが利用する．a_+ の大きさの決定に必要な占有領域の情報は $head$ の占有領域の情報 (B 系の占有領域の半径) である．また，自身の占有領域の情報は，(

占有領域の半径, トポロジーの種類) のタプル型としてインスタンス変数に保持され，a_ϕ ではこれが大きさの決定に用いられる．a_+ の占有領域の情報がタプル型であるのは，a_ϕ を描画する際にトポロジーの種類によってその処理を変えるためである．以上より A_plus の__init__メソッドはアルゴリズム 6 となる．このアルゴリズムの変数の流れを図式化したものが図 4.16 であり，このアルゴリズムによる占有領域を図式化したものが図 4.17 である．

アルゴリズム 6 A_plus(A_minus) クラスの__init__メソッド

条件:

R：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数・実数型)

r：自身の大きさかつ占有領域を表す円の半径を保持するためのインスタンス変数

$data$：自身の占有領域の情報を保持するためのインスタンス変数

A_plus：トポロジーのタイプを親クラスで a_+ と判別するための値 (本研究では string 型)

1: **procedure** __INIT__(R)

2: $r \leftarrow R + 1$

3: $data \leftarrow [R, A_plus]$

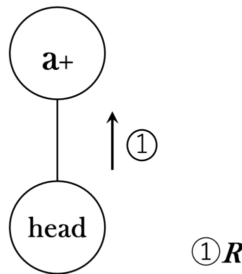

図 4.16　A_plus の__init__処理に用いる情報

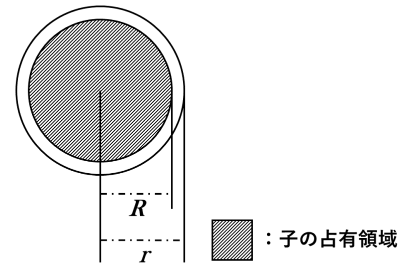

図 4.17　A_plus の占有領域のイメージ

a_2 の大きさ決定

a_2 のトポロジーは，A2 クラスによって扱われる．a_2 は非終端記号であるので，このトポロジーは親ノードと子ノードの両方を持つ．そのため，大きさを決める際は子ノードの占有領域の大きさをもとにその大きさを決め，その大きさを占有領域として親ノードが利用する．a_2 の大きさの決定に必要な占有領域の情報は $head$ の占有領域の情報 ($cons$ で繋がれた c_+ の情報) と $tail$ の占有領域の情報 ($cons$ で繋がれた c_- の情報) である．また，自身の占有領域の情報は，a_+ と同様に (占有領域の半径，トポロジーの種類) のタプル型としてインスタンス変数に保持される．以上より A2 の__init__メソッドはアルゴリズム 7 となる．このアルゴリズムの変数の流れを図式化したものが図 4.18 であり，このアルゴリズムによる占有領域を図式化したものが図 4.19 である．

アルゴリズム 7 A2 クラスの__init__メソッド

条件:

$children1$：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数・リスト型)

$children2$：$tail$ の占有領域の情報 ($tail$ クラスのインスタンス変数・リスト型)

R：自身の物理境界を表す円の半径を保持するためのインスタンス変数

r：自身の大きさかつ占有領域を表す円の半径を保持するためのインスタンス変数

$data$：自身の占有領域の情報を保持するためのインスタンス変数

$A2$：トポロジーのタイプを親クラスで a_2 と判別するための値 (本研究では string 型)

関数 HIGH_C は C 系の情報をまとめたリストから最も高い C 系の占有領域の高さを返す関数

関数 LENGTH_C は C 系の情報をまとめたリストから C 系が接する物理境界の円周を求める関数

1: **procedure** __INIT__($R, children$)
2: **if** HIGH_C($children1$) $>$ HIGH_C($children2$) **then**
3: $h \leftarrow$ HIGH_C($children1$)
4: **else**
5: $h \leftarrow$ HIGH_C($children2$)
6: $circumference1 \leftarrow$ LENGTH_C($children1$) $+ 1$
7: $circumference2 \leftarrow$ LENGTH_C($children2$) $+ 1$
8: **if** $circumference1 >= circumference2$ **then**
9: $circumference \leftarrow circumference1 \times 2$
10: **else**
11: $circumference \leftarrow circumference2 \times 2$
12: $R \leftarrow circumference \div (2 \times \pi)$
13: $r \leftarrow R + h$
14: $data \leftarrow [r, A2]$

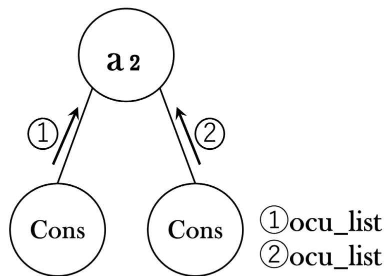

図 4.18　A2 の__init__処理に用いる情報

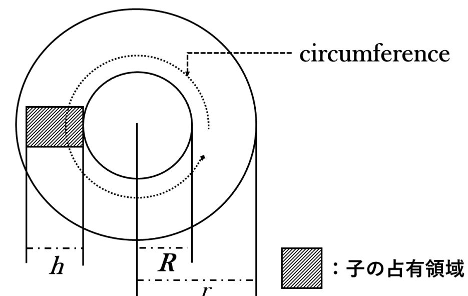

図 4.19　A2 の占有領域のイメージ

b_{++} 及び b_{--} の大きさ決定

$b_{++}(b_{--})$ のトポロジーは，B_plus_plus(B_minus_minus) クラスによって扱われる．b_{++} は非終端記号であるので，このトポロジーは親ノードと子ノードの両方を持つ．そのため，大きさを決める際は子ノードの占有領域の大きさをもとにその大きさを決め，その大きさを占有領域として親ノードが利用する．b_{++} の大きさの決定に必要な占有領域の情報は $head$ の占有領域の情報 (B 系の占有領域の半径) と，$tail$ の占有領域の情報 (B 系の占有領域の半径) である．ここで，本来 b_{++} のトポロジーには向きの区別がないが，描画の都合上本研究では $head$ を上側の homoclinic saddle connection の子ノードとし，$tail$ を下側の homoclinic saddle

connection の子ノードとする．また，自身の占有領域の情報は占有領域を表す円の半径として実数型でインスタンス変数に保持される．以上より B_plus_plus の__init__メソッドはアルゴリズム 8 となる．このアルゴリズムの変数の流れを図式化したものが図 4.20 であり，このアルゴリズムによる占有領域を図式化したものが図 4.21 である．

アルゴリズム 8 B_plus_plus クラスの__init__メソッド

条件:

$R1$：*head* の占有領域の情報 (*head* クラスのインスタンス変数・実数型)

$R2$：*tail* の占有領域の情報 (*tail* クラスのインスタンス変数・実数型)

r：自身の大きさかつ占有領域の情報を保持するためのインスタンス変数

1: **procedure** __INIT__($R1, R2$)

2: $r \leftarrow (2 \times R1 + 2 \times R2 + 4) \div 2$

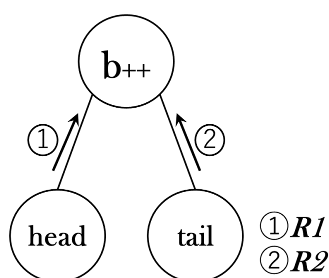

図 4.20　B_plus_plus の大きさ決定に用いる情報

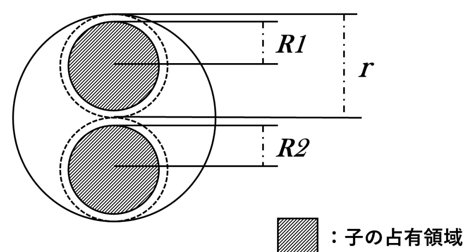

図 4.21　B_plus_plus の占有領域のイメージ

b_{+-} 及び b_{-+} の大きさ決定

$b_{+-}(b_{-+})$ のトポロジーは，B_plus_minus(B_minus_plus) クラスによって扱われる．b_{+-} は非終端記号であるので，このトポロジーは親ノードと子ノードの両方を持つ．そのため，大きさを決める際は子ノードの占有領域の大きさをもとにその大きさを決め，その大きさを占有領域として親ノードが利用する．b_{+-} の大きさ決定に必要な占有領域の情報は *head* の占有領域の情報 (B 系の占有領域の半径) と，*tail* の占有領域の情報 (B 系の占有領域の半径) である．ここで，*head* は内側の homoclinic saddle connection の子ノードであり，*tail* は外側の homoclinic saddle connection の子ノードである．また，自身の占有領域の情報は占有領域を表す円の半径として実数型でインスタンス変数に保持される．以上より B_plus_minus の__init__メソッドはアルゴリズム 9 となる．このアルゴリズムの変数の流れを図式化したものが図 4.22 であり，このアルゴリズムによる占有領域を図式化したものが図 4.23 である．

アルゴリズム 9 B_plus_minus(B_minus_plus) クラスの__init__メソッド

条件:

$R1$：*head* の占有領域の情報 (*head* クラスのインスタンス変数・実数型)

$R2$：*tail* の占有領域の情報 (*tail* クラスのインスタンス変数・実数型)

r：自身の大きさかつ占有領域の情報を保持するためのインスタンス変数

1: **procedure** __INIT__($R1, R2$)

2: $r \leftarrow (2 \times R1 + 2 \times R2 + 4) \div 2$

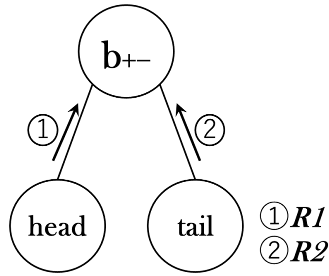

図 4.22　B_plus_minus の大きさ決定に用いる情報

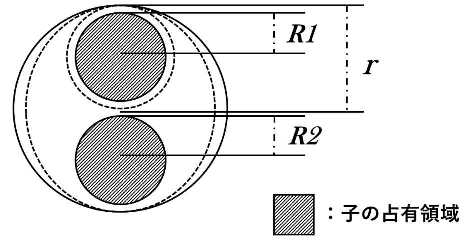

図 4.23　B_plus_minus の占有領域のイメージ

β_+ 及び β_- の大きさ決定

$\beta_+(\beta_-)$ のトポロジーは，Beta_plus(Beta_minus) クラスによって扱われる．β_+ は非終端記号であるので，このトポロジーは親ノードと子ノードの両方を持つ．そのため，大きさを決める際は子ノードの占有領域の大きさをもとにその大きさを決め，その大きさを占有領域として親ノードが利用する．β_+ の大きさ決定に必要な占有領域の情報は *head* の占有領域の情報 (*cons* で繋がれた C 系の情報) である．また，自身の占有領域の情報は占有領域を表す円の半径として実数型でインスタンス変数に保持される．以上より Beta_plus の__init__メソッドはアルゴリズム 10 となる．このアルゴリズムの変数の流れを図式化したものが図 4.24 であり，このアルゴリズムによる占有領域を図式化したものが図 4.25 である．

アルゴリズム 10 Beta_plus(Beta_minus) クラスの__init__メソッド

条件:

children：*head* の占有領域の情報 (*head* クラスのインスタンス変数・リスト型)
R：自身の物理境界を表す円の半径を保持するためのインスタンス変数
r：自身の大きさかつ占有領域を表す円の半径を保持するためのインスタンス変数
関数 HIGH_C は C 系の情報をまとめたリストから最も高い C 系の占有領域の高さを返す関数
関数 LENGTH_C は C 系の情報をまとめたリストから C 系が接する物理境界の円周を求める関数

1: **procedure** __INIT__(*children*)
2: 　$h \leftarrow$ HIGH_C(*children*)
3: 　$circumference \leftarrow$ LENGTH_C(*children*)
4: 　$R \leftarrow circumference \div (2 \times \pi)$
5: 　$r \leftarrow R + h$

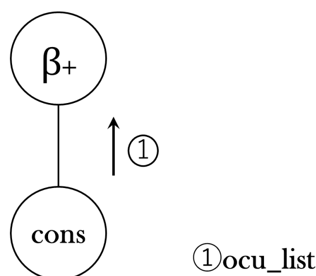

図 4.24　Beta_plus の大きさ決定に用いる情報

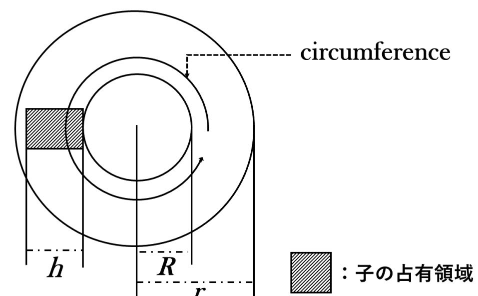

図 4.25　Beta_plus の占有領域のイメージ

c_+ 及び c_- の大きさ決定

$c_+(c_-)$ のトポロジーは，C_plus(C_minus) クラスによって扱われる．c_+ は非終端記号であるので，このトポロジーは親ノードと子ノードの両方を持つ．そのため，大きさを決める際は子ノードの占有領域の大きさをもとにその大きさを決め，その大きさを占有領域として親ノードが利用する．c_+ の大きさの決定に必要な占有領域の情報は *head* の占有領域の情報 (B 系の占有領域の半径) と，*tail* の占有領域の情報 (*cons* で繋がれた C 系の情報) である．また，自身の占有領域の情報は底辺長と高さの情報をまとめた (占有領域の高さ, 占有領域の底辺長) となるタプル型であり，これをインスタンス変数に保持する．以上より C_plus の__init__メソッドはアルゴリズム 11 となる．このアルゴリズムの変数の流れを図式化したものが図 4.26 であり，このアルゴリズムによる占有領域を図式化したものが図 4.27 である．

アルゴリズム 11 C_plus(C_minus) クラスの__init__メソッド

条件:

R：*head* の占有領域の情報 (*head* クラスのインスタンス変数・実数型)
children：*tail* の占有領域の情報 (*tail* クラスのインスタンス変数・リスト型)
w：自身の占有領域の底辺長を表す値を保持するためのインスタンス変数
h：自身の占有領域の高さを表す値を保持するためのインスタンス変数
data：自身の占有領域の情報を保持するためのインスタンス変数
関数 HIGH_C は C 系の情報をまとめたリストから最も高い C 系の占有領域の高さを返す関数
関数 LENGTH_C は C 系の情報をまとめたリストから C 系が接する物理境界の円周を求める関数

1: **procedure** __INIT__$(R, children)$
2: 　**if** $(2 \times R) >$ LENGTH_C$(children)$ **then**
3: 　　$w \leftarrow R$
4: 　**else**
5: 　　$w \leftarrow$ LENGTH_C$(children)$
6: 　$h \leftarrow 2 \times R +$ HIGH_C$(children) + 1$
7: 　$data \leftarrow [h, w]$

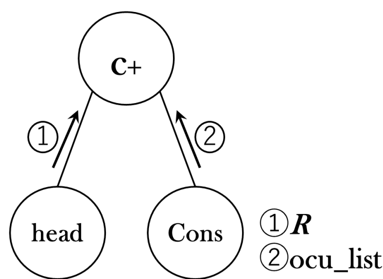

図 4.26　c_plus の大きさ決定に用いる情報

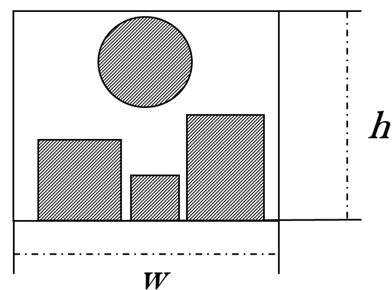

図 4.27　c_plus の占有領域のイメージ

4.1.4　個々の描画アルゴリズム

4.1.3 節では，描画するための各トポロジーの大きさをそれぞれ決定した．この節では，その決定された大きさから流線を描画するアルゴリズムについて説明する．このアルゴリズムの前提条件は 4.1.3 節と同様で，ある言語によらない一般的なアルゴリズムである．そのため，円を描く関数 (DRAW_CIRCLE) などはその意味

のみを記述し実装や内部処理については説明しない．これは，これを実装する言語やライブラリによって実装方法が大きく異なるためである．描画処理は draw メソッドで行われる．4.1.3 節の__init__メソッドではそれぞれのインスタンスが作成され，インスタンス変数として大きさの情報が保持された．draw メソッドはこれらの情報を用いて，根から順に流線の描画を行う．各クラスは親ノードから占有領域の中心点を draw メソッドの引数として渡され，その中心点をもとに流線が描画されるべき位置を計算する．ただ，C 系のみは中心点だけでは描画できないため独自のデータ構造を用いる．また，draw メソッドで描画処理を行わない Cons クラスがある．Cons クラスは，4.1.3 節での処理とは逆に，親ノードから与えられたリストを子ノードに分配する処理を行う．Cons クラスの draw メソッドはアルゴリズム 12 である．また，アルゴリズムの変数の流れを図 4.28 に図式化した．

アルゴリズム 12 Cons クラスの draw メソッド

条件:

$data_list$：描画のための情報をまとめたリスト (親ノードのインスタンス変数・リスト型)
$data1$：$head$ の描画に必要な情報を保持するためのインスタンス変数
$data2$：$tail$ の描画に必要な情報を保持するためのインスタンス変数

```
1: procedure DRAW(data)
2:     data1 ← data_list.pop          ▷ pop はリストの先頭の値を取り出す処理
3:     data2 ← data_list
```

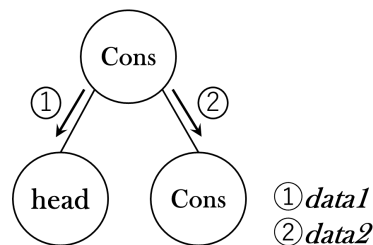

図 4.28　Cons が子ノードに渡す情報

a_ϕ の描画

a_ϕ は開始記号であるので，描画処理では一番最初にこのクラスの draw メソッドが呼ばれる．このクラスは 4.1.3 節で子孫ノードの大きさの情報をまとめた [(占有領域の半径, トポロジーの種類), ...] というリストをそのインスタンスに保持している．draw メソッドはこれを用いて，この流線を描画する．a_ϕ は $cons$ で繋がれている A 系の子孫ノードの数だけ一様流を描画する．また，それら子孫ノードが描画されるべき占有領域の中心点の座標を計算し，それをインスタンス変数にリスト型としてまとめ，保持する．リスト型とするのは，このノードの $head$ が $cons$ であるためで，直接子孫ノードにその描画位置を指定するのではなく $cons$ を経由するからである．以上より，A0 の draw メソッドはアルゴリズム 13 になる．このアルゴリズムの変数の流れを図式化したものが図 4.29 であり，このアルゴリズムによる描画処理を図式化したものが図 4.30 である．

アルゴリズム 13 A0 クラスの draw メソッド

条件:

$children$：__init__メソッドでインスタンスが保持した $head$ の占有領域の情報

$data1$：子ノードの描画に必要な情報を保持するためのインスタンス変数 (リスト型)

関数 DRAW_LINE は座標平面上の 2 点を引数にとり，それを結ぶ線分を描画する関数

```
1:  procedure DRAW
2:      r ← 0
3:      data1 ← []
4:      longest_r ← 0
5:      for i = 0 . . . data.length do
6:          child ← data[i]
7:          if child[0] > longest_r then                    ▷ child[0] は占有領域の半径に対応する
8:              longest_r ← child[0]
9:      edge ← longest_r + 1
10:     for i = 0 . . . data.length do
11:         r ← r + child[0] + 1
12:         data1.add((0, −r))                              ▷ add は引数の値をリストに追加する処理
13:         child ← data[i]
14:         if child[1] = A2 then                           ▷ child[1] は占有領域の種類に対応する
15:             DRAW_LINE((−edge, −r), (−child[0], −r))
16:             DRAW_LINE((child[0], −r), (edge, −r))
17:         else if child[1] = A_minus then
18:             DRAW_LINE((−edge, −r + child[0]), (edge, −r + child[0]))
19:         else
20:             DRAW_LINE((−edge, −r − child[0]), (edge, −r − child[0]))
21:         r ← r + child[0] + 1
```

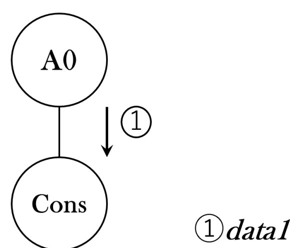

図 4.29　A0 が子ノードに渡す情報

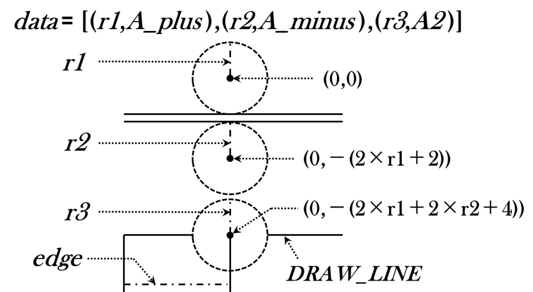

図 4.30　A0 の描画のイメージ

$b_{\phi+}$ 及び $b_{\phi-}$ の描画

$b_{\phi+}(b_{\phi-})$ は開始記号なので，a_ϕ と同様に描画処理では一番最初にこのクラスの draw メソッドが呼ばれる．このクラスは 4.1.3 節で $head$ の占有領域の情報と $tail$ の占有領域の情報から求めた自身の大きさを表す円の半径をそのインスタンスに保持している．B0 クラスの draw メソッドでは自身の大きさの情報から最外境界部の円を描画し，また，子ノードが描画されるべき座標を計算する．ここで，$head$ にはその占有領域の中心

点を，$tail$ には C 系を描画するために用いるデータ構造を与える．C 系を描画するために必要なデータ構造は，(基準点からの円周上の距離, 接する物理境界の半径, 接する物理境界の中心, 接する物理境界が $B0$ かどうか) のタプル型である．B0 クラスでは，このデータ構造を求め，これをリストにして $cons$ に渡す．以上より，B0_plus の draw メソッドはアルゴリズム 14 になる．このアルゴリズムの変数の流れを図式化したものが図 4.31 であり，このアルゴリズムによる描画処理を図式化したものが図 4.32 である．

アルゴリズム 14 B0_plus(B0_minus) クラスの draw メソッド

条件:

r：__init__メソッドでインスタンスが保持した自身の大きさを表す円の半径

$children$：$tail$ の占有領域の情報 ($tail$ クラスのインスタンス変数・リスト型)

$data1$：$head$ の描画に必要な情報を保持するためのインスタンス変数 (タプル型)

$data2$：$tail$ の描画に必要な情報を保持するためのインスタンス変数 (リスト型)

関数 DRAW_CIRCLE は座標と実数値を引数にとり，点を中心に実数値を半径とした円を描画する関数

関数 MAKE_LIST_C は C 系の描画に必要なデータ構造を作成する関数

```
1: procedure DRAW(children)
2:     DRAW_CIRCLE(r, (0, 0))
3:     data1 ← (0, 0)
4:     data2 ←MAKE_LIST_C(children, r, (0, 0), True)
```

アルゴリズム 15 関数 make_list_c

条件:

$children$：C 系の情報をまとめたリスト ([(占有領域の高さ, 占有領域の底辺長), ...])

r：C 系が接する物理境界の半径

$center$：C 系が接する物理境界の中心の座標 (タプル型)

$type$：接する物理境界が B0 か否か (Bool 値)

L：基準点からの円周上の距離 (デフォルト値:0)

保証:

返り値は C 系を描画するために必要なデータ構造

```
1: function MAKE_LIST_C(children, r, center, type, L)
2:     data ← []
3:     l ← L
4:     for i = 0 ... children.length do
5:         l ← l + 1
6:         data[i].add(l, r, center, type)
7:         child ← children[i]
8:         l ← l + child[1]                ▷ child[1] は占有領域の底辺長に対応する
9:     return data
```

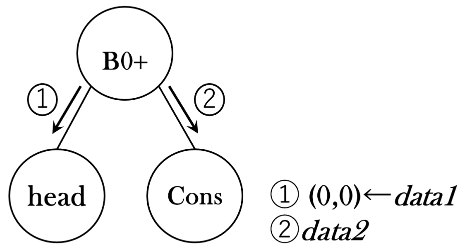

図 4.31　B0_plus が子ノードに渡す情報

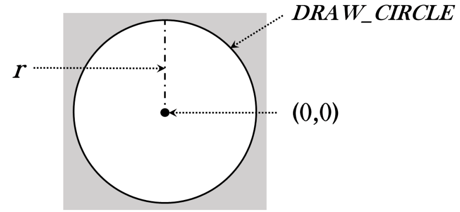

図 4.32　B0_plus の描画のイメージ

a_+ 及び a_- の描画

$a_+(a_-)$ は，親クラスで計算された占有領域の中心点の座標と 4.1.3 節で決定された自身の大きさを用いて，1 つの homoclinic saddle connection からなる流線を描画する．また，子ノードを描画するために，その占有領域の中心点の座標を計算する．以上より，A_plus の draw メソッドはアルゴリズム 16 になる．このアルゴリズムの変数の流れを図式化したものが図 4.33 であり，このアルゴリズムによる描画処理を図式化したものが図 4.34 である．

アルゴリズム 16 A_plus(A_minus) クラスの draw メソッド

条件:

r：__init__メソッドでインスタンスが保持した自身の占有領域を表す円の半径

$center$：このインスタンスの占有領域の中心点の座標 (親クラスのインスタンス変数)

$data1$:$head$ の描画に必要な情報を保持するためのインスタンス変数 (タプル型)

関数 DRAW_CIRCLE は座標と実数値を引数にとり，点を中心に実数値を半径とした円を描画する関数

1: **procedure** DRAW($center$)

2: 　DRAW_CIRCLE($r, center$)

3: 　$data1 \leftarrow center$

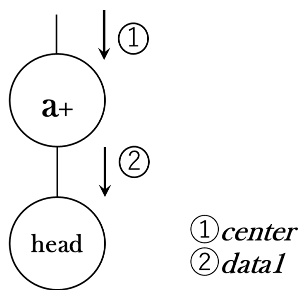

図 4.33　A_plus が子ノードに渡す情報

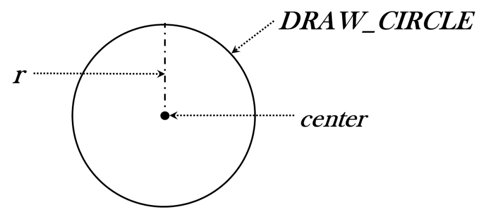

図 4.34　A_plus の描画のイメージ

a_2 の描画

a_2 は，親ノードの draw メソッドで計算された占有領域の中心点の座標と 4.1.3 節で決定された自身の情報をを用いて，物理境界と一様流の一部を描画する．また，子ノードを描画するために，$head$ と $tail$ それぞれについて C 系を描画するために用いるデータ構造を計算する．以上より，A2 の draw メソッドはアルゴリズ

ム 17 になる．このアルゴリズムの変数の流れを図式化したものが図 4.35 であり，このアルゴリズムによる描画処理を図式化したものが図 4.36 である．

アルゴリズム 17 A2 クラスの draw メソッド

条件:

R：__init__メソッドでインスタンスが保持した自身の物理境界を表す円の半径
r：__init__メソッドでインスタンスが保持した自身の占有領域を表す円の半径
$center$：このインスタンスの占有領域の中心点の座標 (親クラスのインスタンス変数)
$children1$：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数・リスト型)
$children2$：$tail$ の占有領域の情報 ($tail$ クラスのインスタンス変数・リスト型)
$data1$：$head$ の描画に必要な情報を保持するためのインスタンス変数 (リスト型)
$data2$：$tail$ の描画に必要な情報を保持するためのインスタンス変数 (リスト型)
関数 DRAW_CIRCLE は座標と実数値を引数にとり，点を中心に実数値を半径とした円を描画する関数
関数 DRAW_LINE は座標平面上の 2 点を引数にとり，それを結ぶ線分を描画する関数
関数 MAKE_LIST_C は C 系の描画に必要なデータ構造を作成する関数

```
1: procedure DRAW(center, children1, children2)
2:     x ← center[0]                    ▷ center[0] は中心点の x 座標に対応する
3:     y ← center[1]                    ▷ center[1] は中心点の y 座標に対応する
4:     DRAW_CIRCLE(R, center)
5:     DRAW_LINE((x − r, y), (x − R, y))
6:     DRAW_LINE((x + R, y), (x + r, y))
7:     data1 ← MAKE_LIST_C(children1, R, center, False)
8:     data2 ← MAKE_LIST_C(children2, R, center, False)
```

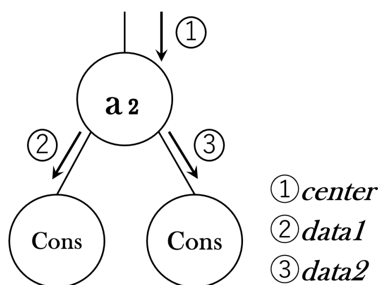

図 4.35　A2 が子ノードに渡す情報

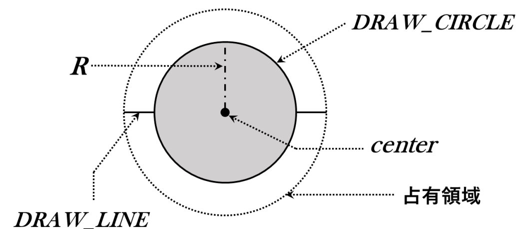

図 4.36　A2 の描画のイメージ

b_{++} 及び b_{--} の描画

$b_{++}(b_{--})$ は，親クラスで計算された占有領域の中心点の座標と 4.1.3 節で決定された自身の大きさを用いて，2 つの homoclinic saddle connection からなる流線を描画する．また，子ノードを描画するために，$head$ と $tail$ それぞれについてそれらの占有領域の中心点の座標を計算する．以上より，B_plus_plus の draw メソッドはアルゴリズム 18 になる．このアルゴリズムの変数の流れを図式化したものが図 4.37 であり，このアルゴリズムによる描画処理を図式化したものが図 4.38 である．

アルゴリズム 18 B_plus_plus(B_minus_minus) クラスの draw メソッド

条件:

$R1$：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数・実数型)

$R2$：$tail$ の占有領域の情報 ($tail$ クラスのインスタンス変数・実数型)

$center$：このインスタンスの占有領域の中心点の座標 (親クラスのインスタンス変数)

$data1$：$head$ の描画に必要な情報を保持するためのインスタンス変数 (タプル型)

$data2$：$tail$ の描画に必要な情報を保持するためのインスタンス変数 (タプル型)

関数 DRAW_CIRCLE は座標と実数値を引数にとり，点を中心に実数値を半径とした円を描画する関数

1: **procedure** DRAW($center, R1, R2$)
2: $x \leftarrow center[0]$
3: $y \leftarrow conter[1]$
4: DRAW_CIRCLE($R1, (x, R2 + y)$)
5: DRAW_CIRCLE($R2, (x, -R1 + y)$)
6: $data1 \leftarrow (x, R2 + y)$
7: $data2 \leftarrow (x, -R1 + y)$

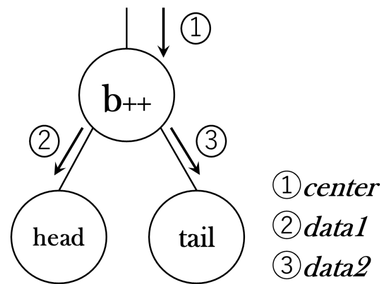

図 4.37　B_plus_plus が子ノードに渡す情報

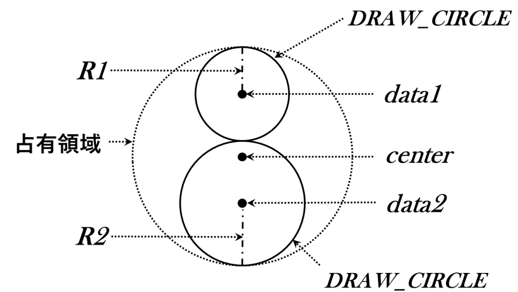

図 4.38　B_plus_plus の描画のイメージ

b_{+-} 及び b_{-+} の描画

$b_{+-}(b_{-+})$ は，親クラスで計算された占有領域の中心点の座標と 4.1.3 節で決定された自身の大きさを用いて 2 つの homoclinic saddle connection からなる流線を描画する．また，子ノードを描画するために，$head$ と $tail$ それぞれについてそれらの占有領域の中心点の座標を計算する．以上より，B_plus_minus の draw メソッドはアルゴリズム 19 になる．このアルゴリズムの変数の流れを図式化したものが図 4.37 であり，このアルゴリズムによる描画処理を図式化したものが図 4.40 である．

アルゴリズム 19 B_plus_minus(B_minus_plus) クラスの draw メソッド

条件:

$R1$：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数・実数型)
$R2$：$tail$ の占有領域の情報 ($tail$ クラスのインスタンス変数・実数型)
$center$：このインスタンスの占有領域の中心点の座標 (親クラスのインスタンス変数)
$data1$：$head$ の描画に必要な情報を保持するためのインスタンス変数 (タプル型)
$data2$：$tail$ の描画に必要な情報を保持するためのインスタンス変数 (タプル型)
関数 DRAW_CIRCLE は座標と実数値を引数にとり，点を中心に実数値を半径とした円を描画する関数

1: **procedure** DRAW($center, R1, R2$)
2: $x \leftarrow center[0]$
3: $y \leftarrow conter[1]$
4: DRAW_CIRCLE($R1, (x, R2 + y)$)
5: DRAW_CIRCLE($R1 + (R2 \times 2), (x, y)$)
6: $data1 \leftarrow (x, R2 + y)$
7: $data2 \leftarrow (x, -R1 + y)$

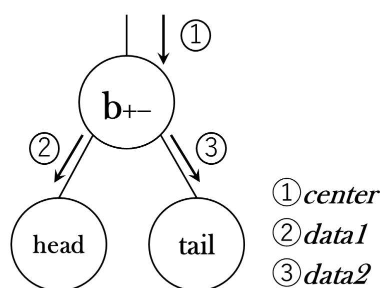

図 4.39　B_plus_minus が子ノードに渡す情報

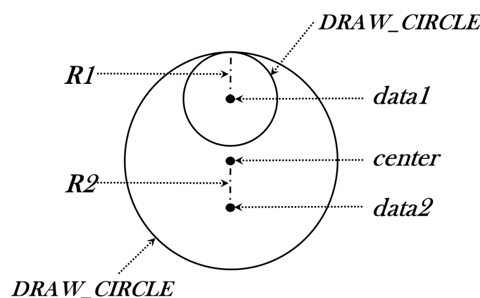

図 4.40　B_plus_minus の描画のイメージ

β_+ 及び β_- の描画

$\beta_+(\beta_-)$ は，親クラスで計算された占有領域の中心点の座標と 4.1.3 節で決定された自身の大きさを用いて物理境界を描画する．また，子ノードを描画するために，$head$ について C 系を描画するために用いるデータ構造を計算する．以上より，Beta_plus の draw メソッドはアルゴリズム 20 になる．このアルゴリズムの変数の流れを図式化したものが図 4.41 であり，このアルゴリズムによる描画処理を図式化したものが図 4.42 である．

アルゴリズム 20 Beta_plus(Beta_minus) クラスの draw メソッド

条件:

R：__init__メソッドでインスタンスが保持した自身の物理境界を表す円の半径
$center$：このインスタンスの占有領域の中心点の座標 (親クラスのインスタンス変数)
$children$：$head$ の占有領域の情報 ($head$ クラスのインスタンス変数・リスト型)
$data1$：$head$ の描画に必要な情報を保持するためのインスタンス変数 (リスト型)
関数 DRAW_CIRCLE は座標と実数値を引数にとり，点を中心に実数値を半径とした円を描画する関数
関数 MAKE_LIST_C は C 系の描画に必要なデータ構造を作成する関数

1: **procedure** DRAW($center, children$)
2: DRAW_CIRCLE($R, center$)
3: $data1 \leftarrow$MAKE_LIST_C($children, R, center$, False)

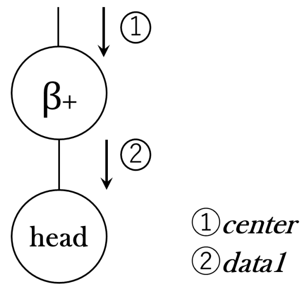

図 4.41　Beta_plus が子ノードに渡す情報

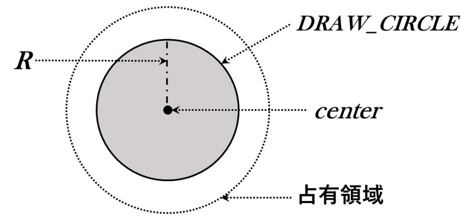

図 4.42　Beta_plus の描画のイメージ

c_+ 及び c_- の描画

$c_+(c_-)$ は，親クラスで計算された C 系用のデータ構造と 4.1.3 節で決定された自身の大きさを用いて物理境界を描画する．C 系は物理境界に接する形状をしているため，単純に占有領域の中心点の座標を与えることで描画することはできない．そのため，情報の受け渡しには独自のデータ構造を用いる．C 系の描画に用いるデータ構造は C 系が接する物理境界を持つクラスで作成され，(基準点からの円周上の距離, 接する物理境界の半径, 接する物理境界の中心, 接する物理境界が $B0$ かどうか) の情報をまとめたタプル型である．これらの情報から物理境界の周りに接する C 系の流線を点でモデル化する．各点の座標の計算には三角関数を用いる．また，子ノードを描画するために，$head$ にはその占有領域の中心点を，$tail$ には C 系を描画するために用いるデータ構造を計算してインスタンスに保持する．以上より，C_plus の draw メソッドはアルゴリズム 21 になる．このアルゴリズムの変数の流れを図式化したものが図 4.43 であり，このアルゴリズムによる描画処理を図式化したものが図 4.44–4.46 である．

アルゴリズム 21 C_plus(C_minus) クラスの draw メソッド

条件:

R：*head* の占有領域の情報 (*head* クラスのインスタンス変数・実数型)
children：*tail* の占有領域の情報 (*tail* クラスのインスタンス変数・リスト型)
h：__init__メソッドでインスタンスが保持した自身の占有領域の高さ
w：__init__メソッドでインスタンスが保持した自身の占有領域の底辺長
data：C 系を描画するための情報をまとめたリスト (親ノードのインスタンス変数・リスト型)
*data*1：*head* の描画に必要な情報を保持するためのインスタンス変数 (タプル型)
*data*2：*tail* の描画に必要な情報を保持するためのインスタンス変数 (リスト型)
関数 HIGH_C は C 系の情報をまとめたリストから最も高い C 系の占有領域の高さを返す関数
関数 THETA_POINT は線分の長さ，角度，座標を引数として座標平面上の位置を返す関数
関数 DRAW_SPLINE は座標のリストを結ぶスプライン補間を行いそれを描画する関数
関数 MAKE_LIST_C は C 系の描画に必要なデータ構造を作成する関数

1: **procedure** DRAW($data, R, children$)
2: $l \leftarrow data[0]$ ▷ 接する物理境界の円周の基準点からの円周上の距離に対応する
3: $r \leftarrow data[1]$ ▷ 接する物理境界の半径に対応する
4: $center \leftarrow data[2]$ ▷ 接する物理境界の中心座標に対応する
5: $bool \leftarrow data[3]$ ▷ 接する物理境界が B0 かどうかの bool 値に対応する
6: $high_c \leftarrow$ HIGH_C($children$)
7: $\theta_1 \leftarrow l \div r$
8: $\theta_2 \leftarrow (l + w) \div r$
9: $\theta_3 \leftarrow ((\theta_2 - \theta_1) \div 2) + \theta_1$
10: $point_1 \leftarrow$ THETA_POINT($\theta_1, r, center$)
11: $point_2 \leftarrow$ THETA_POINT($\theta_2, r, center$)
12: **if** $bool$ = True **then** ▷ 親クラスが B0 だった場合，親の物理境界の内側に C 系を描画する
13: $point_3 \leftarrow$ THETA_POINT($\theta_3, r - h, center$)
14: $center_b \leftarrow$ THETA_POINT($\theta_3, r - R - high_c - 1, center$)
15: **else**
16: $point_3 \leftarrow$ THETA_POINT($\theta_3, r + h, center$)
17: $center_b \leftarrow$ THETA_POINT($\theta_3, r + R + high_c + 1, center$)
18: **if** $r = 0$ **then** ▷ B 系の子を持つかどうか調べる
19: DRAW_SPLINE($[point_1, point_3, point_2]$)
20: **else**
21: $\theta_b \leftarrow \pi - ((\pi \div 2) + \theta_3)$
22: $point_{b1} \leftarrow$ THETA_POINT($-\theta_b, R + 1, center_b$)
23: $point_{b2} \leftarrow$ THETA_POINT($\pi - \theta_b, R + 1, center_b$)
24: $point_{b3} \leftarrow$ THETA_POINT($-\theta_b - (\pi \div 6), R + 1, center_b$)
25: $point_{b4} \leftarrow$ THETA_POINT($\pi - \theta_b + (\pi \div 6), R + 1, center_b$)
26: DRAW_SPLINE($[point_1, point_{b3}, point_{b1}, point_3, point_{b2}, point_{b4}, point_2]$)
27: $data \leftarrow$ MAKE_LIST_C($children, r, center, bool, l$)
28: HEAD.DRAW($center_b$)
29: TAIL.DRAW(data)

アルゴリズム 22 関数 THETA_POINT

条件:

θ：角度を表す値 (ラジアン)

r：円の半径を表す値

$center$：円の中心の座標を表す値

保証:

返り値は半径 r の円周上の点 (x,y)

1: **procedure** THETA_POINT($\theta, r, center$)

2: $x \leftarrow r \times \cos\theta + center[0]$ ▷ $center[0]$ は円の中心の x 座標に対応する

3: $y \leftarrow r \times \sin\theta + center[1]$ ▷ $center[1]$ は円の中心の y 座標に対応する

4: **return** (x, y)

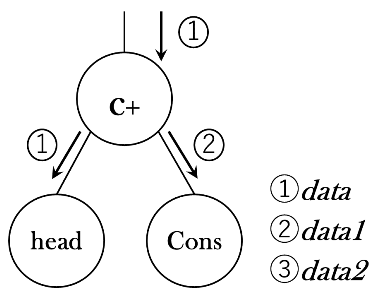

図 4.43 C_plus が子ノードに渡す情報

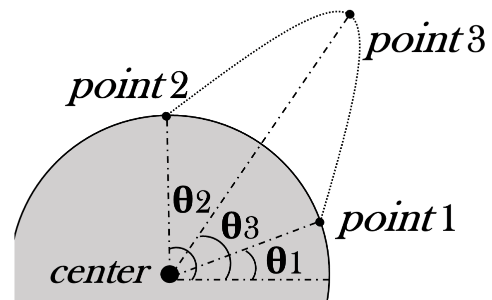

図 4.44 C_plus の描画のイメージ

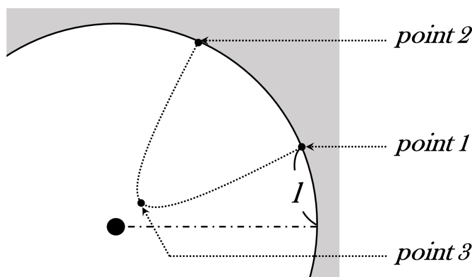

図 4.45 C_plus の描画のイメージ (親クラスが B0 の場合)

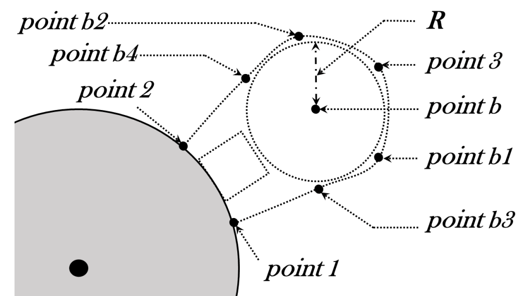

図 4.46 C_plus の描画のイメージ (c_+ が B 系の子ノードを持つ場合)

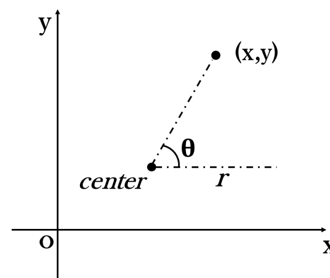

図 4.47 関数 THETA_POINT が座標 (x, y) を計算するイメージ

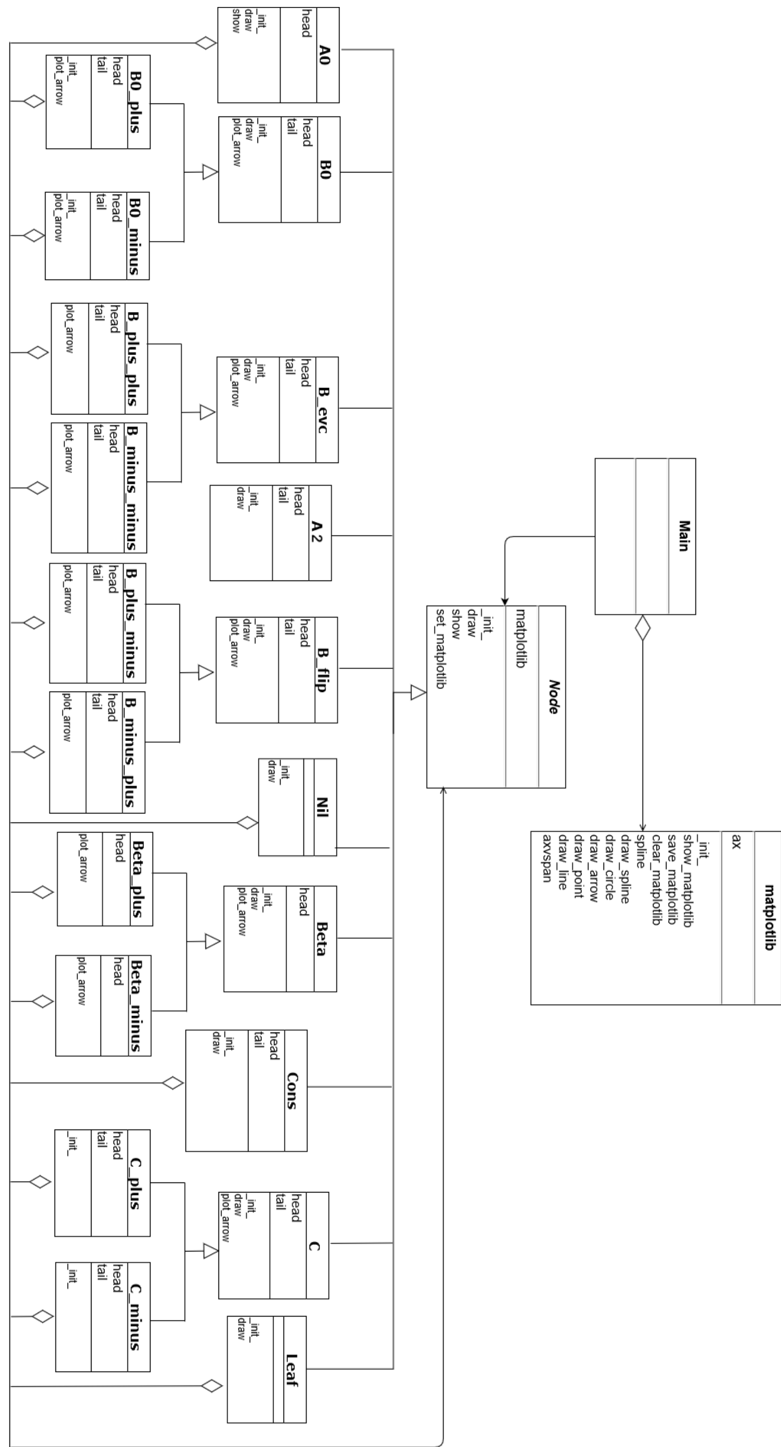

図 4.48　Python のクラス図

4.1.5　Python で実装するクラス図

Python によって実装されるクラス図は，3.6 節によって作成したクラス図を拡張したクラス図である．図 4.48 は Matplotlib クラス・show メソッド・set_matplotlib メソッドを追加した Python のクラス図である．

- Matplotlib クラス
  描画に必要なモジュールをまとめたクラスである．

set_matplotlib メソッドによって, インスタンス作成時に新しい図の下地を作成する, これによって, プログラムを連続で実行した場合にも描画可能になった．show メソッドによって入力した木構造を表示する．

4.1.6 抽象構文木の作成と大きさの決定

コンストラクタメソッドによって抽象構文木の生成と描画の大きさを決定した．コンストラクタはインスタンスを生成した時に必ず呼び出され，オブジェクトの生成とデータの初期化を行う．本研究では，yacc モジュールが LALR 法を用いて構文解析を行っているため，字句ごとにクラスを作成し，クラス内で抽象構文木の生成と同時に大きさの定義 4.1.1 を行った．構文解析は葉から順に実行され，同時に大きさの定義を行うことで木表現の葉から大きさの決定を可能とした．文字列がトークンに変換され，文法ルールに従って yacc 内のメソッドが順に実行されていく．このとき yacc 内のメソッドが，それぞれの対応するクラスのインスタンスを生成し．コンストラクタメソッドである__init__メソッドが呼び出され抽象構文木の生成と大きさの決定を行う．コンストラクタメソッドは，葉から実行されてるため，葉から徐々に大きくなるように，大きさを決めていく．それぞれのクラスのコンストラクタメソッドは引数に子のクラスを持っている．そのため引数を使うことで子のクラスのメソッドを呼び出すことができ，根から順に描画を行うことが可能である．

4.1.7 描画機能の実装

この節では，4.1.4 節のアルゴリズムを Python で実装する方法について説明する．4.1.4 節のアルゴリズムでは，各クラスの draw メソッドの引数に親ノードで決定されるインスタンス変数を必要としていた．しかし，4.1.6 節にあるように各インスタンスは子ノードのインスタンスを保持している．そのため，各インスタンス内から直接子ノードの draw メソッドを呼び出すことが可能である．また，実装に当たっては DRAW_CIRCLE などの関数を定義する必要がある．Matplotlib には，あらかじめ円や長方形などの基本的な図形を描画できる機能が備わっている．DRAW_LINE や DRAW_CIRCLE，saddle を表す点や向きを表す矢印などはこの Matplotlib の機能で再現できる．Matplotlib では，最初にキャンパスのようなものを作成し，このキャンパス上に関数などをプロットしていくことになる．そして，最後にこのキャンパスを表示することでプロットされた関数を見ることができるのである．その仕様上，各クラスの DRAW_LINE は同じキャンパス上にその図形を描画する必要がある．そのため，実装では描画処理の前に Matplotlib クラスのインスタンスを main クラスで作成する．この Matplotlib クラスでは__init__時にキャンパスを作成する．つまり，Matplotlib インスタンスは 1 つ 1 つのキャンパスである．そして，各クラスの set_matplotlib メソッドを呼び出し，作成したインスタンスを引数として与える．これにより，各トポロジーのインスタンスでは引数として与えられたキャンパスに対しての描画が可能となる．以上より，4.1.4 節のアルゴリズムの Python への実装が完了した．

4.1.8 描画の限界と可読性

本研究において，描画する流線の大きさによって個々の流れの図が小さくなってしまい，線と線の間隔が非常に短くなってしまう場合がある．私たちは，実際に描画を行うことで，描画の限界を調べた．

可読性の定義 3.3 節を満たしていないと判断した場合に描画の限界とした．Matplotlib では，キャンパスを満たすように描画されるため，流線の大きさに関係なく，描画できる．そのため，大きさが大きい流線を表示させた場合，線が重なってしまい視覚的に流線の判断が難しくなる．しかし，Matplotlib の機能には，ズーム機能が存在するためズームすることで視覚的にも判断可能になる．さらに，ズームされたものも保存することが可能であるため複雑な木構造を扱った場合にも, 部分的な流れであれば保存することができる．そのため，実際には線は重ならず描画することは可能であり，可読性を満たしているが，実際に人が見ると重なって見え

てしまうため，評価指標の妥当性は確かめられなかった．

4.2 Asymptote による実装

4.2.1 目的達成設定

本研究での目的達成の成果物は，アプローチ次第で変わる可能性がある．Asymptote による実装では，与えられた木表現に対して 3.3 節の定義を満たす図を作成するためのライブラリを作成することで本研究の目的の達成したと言えることとする．

4.2.2 提供するライブラリ

ライブラリ内の流線図には，図に配置の基準となる点や大きさなど，利用者が指定しなければならないパラメータが必ず付与されている．3.3 節の定義を満たす図を作図するためには，ずべての流線図の特徴とコードを把握する必要がある．

パラメータのデータ型

- pair ｘ座標とｙ座標の二次元座標における複素数を表すデータ型であり，本研究では図の基準点として扱う．例えば実部ｘに 0，虚部ｙに 0 を代入すると（0，0）を基準点として作図される．
- real 実数を表すパラメータであり，本研究ではの流線構造の大きさを決めるデータ型となる．例を示すと a_+ の図では homoclinic saddle connection の半径を示している．すなわち，homoclinic saddle connection の大きさは real 型に与えられた実数に比例する．homoclinic saddle connection の中には他の流線図が入る場合もため，その場合は中の流線図より大きい値を real 型に与えなければ，3.3 節の定義を満たない図となるため作図するたびに調節する必要性がある．
- bool3 値として true，default，false の 3 つを与えることができる．本研究では A 系，B 系，C 系の流線図を作成するライブラリの作成を目標としており，それぞれの下付き文字の種類は＋（プラス），ー（マイナス），2 の最大 3 種類である．bool3 型はそれらを区別するために採用したデータ型である．似たデータ型で bool 型が存在するが，bool 型は true，false の 2 種類しか与えることができない．すべての流線図を網羅するためには 3 種類の値を与えられる bool3 型を使用する必要があった．本研究では bool3 に true が与えられると a_+ や b_{++} などの homoclinic saddle connection が反時計回りの軌道での流れを示すプラスの流線図が表現される．false が与えられたのならば，a_- や b_{--} などの homoclinic saddle connection が時計回りの軌道での流れを示すマイナスの流線図が表現される．また，default が与えられたならば $a2$ などの流線図が表現される．

関数

- draw 描画するための線を描く命令をする関数である．draw((x1,y1)–(x2,y2)) により，座標 (x1,y1) から (x2,y2) までの直線を描くことができる．またラベルや矢印を付与したり線の色を変更したりする特性を持つことも可能である．本研究では，矢印を付与する arrow と円を描く circle の特性を持つ draw 関数を使用した．
- fill 指定した場所に，指定した色を塗りつぶす命令をする関数である．本研究では障害物を灰色で塗りつぶすことで，流線との違いを明確にしている．障害物は円状にしているので circle 関数と共に使用される．
- arrow 本研究では矢印を流線上に表記することで，流線の軌道の進む向きを表現している．arrow 関数はその際に使用する矢印を表現する関数である．draw 関数の内部に arrow 関数を与えることで流線上に矢印付与することができ，流線図の拡大縮小の際にも整合性を保つことができる．矢印の場所は

draw 関数で描く線分の始点，中間点，終点の 3 種類である．図 4.49 はその 3 種類を実際に実行した図である．

図 4.49　左から BeginArrow，MidArrow，EndArrow

この 3 つの場所以外に書く場合は Relative 関数をともに使用して描くことが可能である．また，矢印の種類は変更することも可能であり，4 種類の矢印が存在する．図 4.50 はその 4 種類を実際に実行した図である．

図 4.50　左から DefaultaHead，SimpleHead，HookHead，TeXHead

SimpleHead や TeXHead などの微細な矢印では拡大縮小の際に矢印が確認できなくなり流線の軌道の進む向きが判断できなくなる恐れがある．本研究では，DefaultHead や HookHead を使用して流線の完全性を保てるようにする．

- circle 円を描く命令する関数である．中心点となる座標と半径となる大きさを与えることで任意の円を描くことができる．本研究では流線の homoclinic saddle connection と障害物を表現する際に使用される．
- dot 点に近い一定の大きさの円を与えられた座標点に描く命令をする関数である．本研究では淀み点を表現するために dot 関数を使用する．淀み点は黒丸なので fill 関数と circle 関数を用いて表現することも可能であるが，dot 関数を用いることで簡潔に表現できるので使用した．また，大きさが一定であるため流線の拡大縮小の際にも完全性を保つ流線を描くことが可能である．
- Relative Relative の引数には 0 から 1 までの実数が与えられる．arrow 関数の引数内に Relative 引数によって矢印を Arrow(Relative(0.0)) は BeginArrow，Arrow(Relative(0.5)) は MidArrow，Arrow(Relative(1.0)) は EndArrow とそれぞれプログラムに同等の影響を与えるものである．すなわち，Relative の引数の数字は，書き始めを 0，書き終わりを 1 として扱い，その間の数字を与えることで，任意の場所をしていすることが可能である．
- scale scale は指定した流線の大きさを変える関数である．引数には等倍を 1 とする実数が与えられ，実数値に比例して指定した流線の大きさは変わる．

4.2.3　流線のアルゴリズム

この節では，3.3 節での流線構造を描画する際の手順を各流線ごとに記す．Asymptote での実装では A 系と B 系それぞれに抽象データ型を用いて実装しており，中には流線の場所や大きさを示すパラメータや描画関数がそれぞれにまとめられている．また，条件分岐文である if 文をそれぞれの抽象データ型の中で用いており，引数内に指定のパラメータを入れることで目的の流線を描けるようにしている．抽象データ型を用いることで，関数の引数の数を最小限に抑えられ実行速度を早くする効果が期待される．さらに抽象データは情報隠蔽の特性を持っているため，内部のデータと手続きは利用者から隠蔽され，内部の変更を容易にしたり，利用者は外部インターフェースのみでプログラムを活用できる利点を持つ．流線構造を描く際に，流線の homoclinic saddle connection の中に他の流線構造が入る場合がある．流線構造とその中に入る流線構造を描く実行をそれぞれ行うことで描画することもできるが，中に入る流線構造を自動的に実行されるようにすることにより，一回の実行で描画することを可能にした．下記にはアルゴリズムをより把握するために流線ごとにコードを記す．

A 系の実装

A 系は a_+ ，a_- ，$a2$ の 3 パターンの描画が可能である．それぞれ与えられるパラメータにより，種類や大きさが決められる．流線の基準点となる pair 型は変数 x に置き換えられ，a_+ と a_- は saddle point を示し，$a2$ は物理境界の中心を示す．real 型は変数 r に置き換えられ，a_+ と a_- は homoclinic saddle connection の半径，$a2$ は物理境界の半径を示す．bool 型は変数 d に置き換えられ，ture なら a_+ ，false なら a_- ，それ以外なら $a2$ が描画される．a_+ と a_- は Arrow 関数を付与した draw 関数により，基準点 x から X 軸座標の正と負の方向に，変数 r の大きさ分に比例した矢印付きの直線が描画される．そして，circle 関数を共に用いて基準点 x から a_+ は Y 軸座標の負の方向，a_- は Y 軸座標の正の方向に変数 r の大きさ分に比例した矢印付きの homoclinic saddle connection が描画される．$a2$ は物理境界から X 軸座標の正と負の方向に変数 r の大きさ分に比例した矢印付きの直線が draw 関数によって表現される．周囲に流線軌道を持つ物理境界は基準点 x を中心とした，半径 r の circle 関数と Arrow 関数を付与した draw 関数と，circle 関数で示した中をを灰色に塗りつぶす fill 関数によって表している．$a2$ の物理境界の周りの流線軌道には 2 つの矢印を表現する必要があるが，1 つの draw 関数の中に Arrow 関数は 1 つしか付与することができないため，2 つの矢印を表現する場合は同じ draw 関数を記述する必要がある．さらに，この物理境界は X 軸を線対称としているため，reverse 関数を付与した逆向きに描く draw 関数を記述した．また，homoclinic saddle connection の中には他の流線が入る場合は，scale 関数を用いて流線が入るスペースを作る．図 4.51 は場合分けのイメージ図，図 4.52 は描画のイメージ図である．

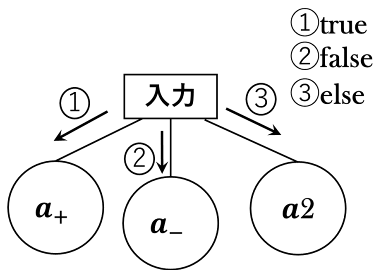

図 4.51　A 系の場合分けイメージ図

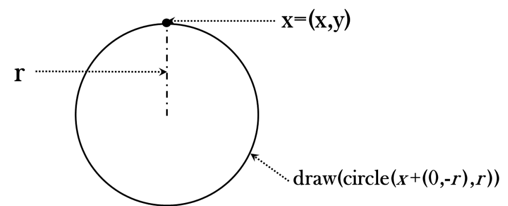

図 4.52　A 系の描画イメージ図

プログラム 4.1　A 系

```
1
2  struct A {
3      pair x;
4      real r;
5      bool3 d;
6      B in;
7
8
9      void operator init(pair xx, real rr) {
10         x = xx;
11         r = rr;
12     }
13
14     void operator init(pair xx, real rr, bool3 dd) {
15         x = xx;
16         r = rr;
```

```
17          d = dd;
18      }
19
20      void mydraw() {
21          if (d == false) {
22              draw(x+(-r,0) -- x+(0,0), arrow=MidArrow(HookHead));
23             draw(x+(0,0) -- x+(r,0), arrow=MidArrow(HookHead));
24              draw(reverse(circle(x+(0,r), r)), arrow=MidArrow(HookHead));
25              dot(x+(0,r));
26              dot(x);
27          } else if (d == true) {
28              draw(x+(-r,0) -- x+(0,0), arrow=MidArrow(HookHead));
29              draw(x+(0,0) -- x+(r,0), arrow=MidArrow(HookHead));
30              draw((circle(x+(0,-r), r)), arrow=MidArrow(HookHead));
31              dot(x+(0,-r));
32              dot(x);
33          } else {
34              draw(x+(-2r,0) -- x+(-r,0),arrow=MidArrow(HookHead));
35              draw(x+(r,0) -- x+(2r,0),arrow=MidArrow(HookHead));
36              fill(circle(x, r),gray);
37              draw(circle(x, r), Arrow(Relative(0.77)));
38              draw(reverse(circle(x, r)), Arrow(Relative(0.77)));
39          }
40          if (in != null) {
41              in.mydraw();
42          }
43      }
44      void scale(real a) {
45          r *= a;
46          in.scale(a);
47      }
48  }
```

B 系の実装

B 系は b_{++}，b_{--}，b_{+-}，b_{-+} の 4 パターンの描画が可能である．流線の基準点となる pair 型は変数 x に置き換えられ，saddle point を示す．real 型は変数 r に置き換えられ，b_{++} と b_{--} は homoclinic saddle connection の半径を示し，b_{+-} と b_{+-} は外側の homoclinic saddle connection の半径を示す．2 つの bool 型は 1 つめは変数 d，2 つ目は変数 d2 に置き換えられ，両方 ture なら b_{++}，両方 false なら b_{--}，b が ture で b2 が false なら b_{+-}，b が false で b2 が true なら b_{-+} が描画される．すべてのパターンは共通して circle 関数と arrow 関数を持つ 2 つの draw 関数で構成されてる．b_{++}，b_{--} は基準点 x から Y 軸座標の正と負の方向に変数 r の距離分移動した点を中心点とし，半径 r の circle 関数により 2 つの homoclinic saddle connection が表現される．b_{+-}，b_{-+} は基準点 x から Y 軸座標の正の方向に，変数 r の距離分移動した点を中心点とし，半径 r の circle 関数と変数 r を 0.6 倍した circle 関数により 2 つの homoclinic saddle connection が表現される．整合性を保つために，これらのパターンすべての時計回りの軌道を示す circle 関数には reverse 関数を付与させている．また，homoclinic saddle connection の中に他の流線が入る場合は，scale 関数を用いて流線が入るスペースを作る．図 4.51 は場合分けのイメージ図，図 4.52 は描画のイメージ図である．

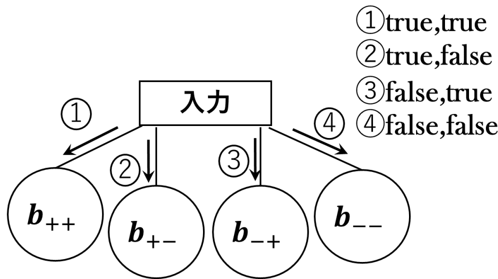

図 4.53　B 系の場合分けイメージ図

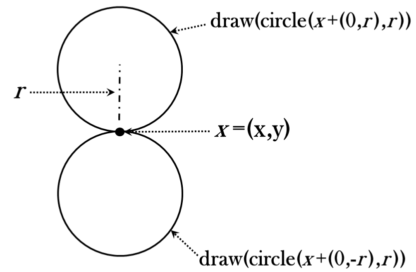

図 4.54　B 系の描画イメージ図

プログラム 4.2　B 系

```
struct B {
    pair x;
    real r;
    bool3 d,d2;
    B in;

    void operator init(pair xx, real rr) {
        x = xx;
        r = rr;
    }

     void operator init(pair xx, real rr,bool3 dd) {
        x = xx;
        r = rr;
        d = dd;
    }

    void operator init(pair xx, real rr, bool3 dd, bool3 ddd) {
        x = xx;
        r = rr;
        d = dd;
        d2 = ddd;
    }


    void mydraw() {
        if (r != 0) {
            if (d == false && d2 ==false) { /* b-- */
                draw(reverse(circle(x+(0,r), r)), arrow=MidArrow(HookHead));
                draw(reverse(circle(x+(0,-r), r)), arrow=MidArrow(HookHead));
            } else if (d == false && d2 == true) { /* b-+ */
                draw(reverse(circle(x+(0,r), r)), arrow=MidArrow(HookHead));
                draw((circle(x+(0,r*0.6), r*0.6)), arrow=MidArrow(HookHead));
            } else if (d == true && d2 == false) { /* b-+ */
                draw((circle(x+(0,r), r)), arrow=MidArrow(HookHead));
                draw(reverse(circle(x+(0,r*0.6), r*0.6)), arrow=MidArrow(HookHead));
```

```
37            } else { /* b++ */
38                draw((circle(x+(0,r), r)), arrow=MidArrow(HookHead));
39                draw((circle(x+(0,-r), r)), arrow=MidArrow(HookHead));
40            }
41            dot(x);
42            if (in != null) {
43                in.mydraw();
44            }
45        }
46    }
47    void scale(real a) {
48        r *= a;
49    }
50 }
```

4.2.4 クラス図との対応部分

この節では，3.6 節で記したクラス図との対応部分について記す．Asymptote では実装の際にリストを用いてはいないので，Cons クラスや内部の head や tail などのリストを用いることによって生じるクラスや変数は対応していない．また，C 系も Asymptote の実装上の問題によりクラス図とは対応していない．しかし，他の流線構造をはじめ draw 関数やコンストラクタはクラス図と対応しているため，構造理解のためのモデリングとして扱うことが可能である．

4.2.5 Asymptote による実行方法

Asymptote ではプログラムを実行する方法として，Ubuntu の端末や Windows のコマンドプロンプト上に直接コードを記述する interactive mode(対話モード) という方法が存在する．そこで上記のライブラリを読み込み，画面上に表示させたい流線図に対応する関数を記述することで，画面上にトポロジー化された流線図を表示させる．例として，a_+ の中に b_{++} が入った流線構造を描く．4.3 は実行のコード，図 4.55 は描画された流線図である．

プログラム 4.3　実行方法

```
1 $ asy /* interactive mode */
2 Welcome to Asymptote version 2.41 (to view the manual, type help)
3 > input fin;  /* ライブラリの呼び出し */
4 > A ap = A((0,0),3,true);  /* $a_{+}$ */
5 > ap.in = B((0,-3),1.2,true,true); /* 中に入る$b_{++}$ */
6 > ap.mydraw();  /* 描画関数実行 */
```

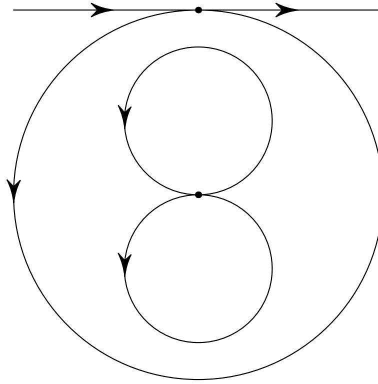

図 4.55　例

4.2.6　描画の有効性

本研究では流線構造を任意の場所に任意の流線が描画できる．すなわち，すべての流線の場所や大きさを理解して，パラメータを調節しなければ 3.3 節の定義を損ねた流線になる可能性がある．また，C 系の実装を試みたところ，文法規則上複数個描かれる場合があり，なおかつ物理境界円上の二点を弧を描いて結ぶような流線であるため，パラメータの調節が難儀になり 3.3 節の定義を損ねる原因になりかねない．よって本研究では，C 系を除いた流線を実装することにした．解決策としては，字句解析を搭載することが挙げられる．流線の基準点だけ整え，大きさは他の流線に合わせて調節されるようになれば 3.3 節の定義が損なわれない流線が描かれ，より文法規則に則った流線構造が描けると考える．また，本研究では場合分けで図の描画を試みたが，クラスなどを用いて図自体に大きさや場所を変動させる特性を持たせることで，よりモジュラリティのある描画も期待できる．

第 5 章

おわりに

本研究では，木表現で表された構造安定な非圧縮流を 2 次元上のトポロジー表現へ変換するアルゴリズムを設計し，Python と Asymptote を用いて実装した．流れを計算機で扱う場合は木表現で扱いやすい．一方，図で表現することで専門知識がない人でも直感的に流れの特徴を把握することが可能である．計算機で扱った木表現を図に変換することで，それぞれのメリットを活用することが可能である．また，自動化することによって変換のミスを防止が期待できる．

Python において，設計した描画アルゴリズムは，占有領域という概念を用いて流線図の大きさを葉から順に大きくすることによって流線間の距離を一定に保つことができ，線を必ず重ならせない．また，それぞれの流線を円か点でモデル化することによって滑らかな線を表現した．以上より，考案したアルゴリズムは 3.3 節で定めた条件を満たした図を，下で述べる欠点以外では作成できるものであると言える．実装したプログラムにより，木表現で表されたトポロジーを速やかに 2 次元上のトポロジーへ間違いなく正確に変換することが可能となった．実際，約 4000 の流れの木表現を本プログラムで変換するテストを行ったところ，全て正確に 2 次元の図が表示された．したがって，本アルゴリズムとプログラムを用いることで，流体解析者が木表現で表現されたトポロジーを，少なくともテストケースの範囲内では間違いなく速やかに直観的に理解できるようになったと言える．しかし，**??**節で定義した図の条件を完全に満たすことができたかどうかについては検討の余地があると考える．本研究で製作したアルゴリズムの占有領域は設計を単純化するため，各流線を覆う円で表されている．これは，b_{++} なども 1 つの円として表しているため，実際の流線よりも占有領域の面積が大きい．このため，b_{++} など実際の流線よりもその占有領域の面積が大きい流線が増えれば増えるほど空白が多くなり，一見して無駄な部分が多く見づらい．従って適切な距離を完全に保てているとは言い難い．解決には，占有領域をより柔軟に定義するアルゴリズムの設計が求められる．

Asymptote では，流線を描く際，場所と大きさを把握してパラメータを調節することにより 3.3 節の定義を満たす流線を描くことが可能である．しかし，4.2.6 節で記したとおり流線の特性上，C 系を描くことは 3.3 節の定義を損ねる可能性があるため，A 系や B 系を描く場合であるならこのプログラムは有益であると考えられる．また，前節で記したように字句解析を実装することでこの問題も解決できる可能性を持ち，図自体に大きさや場所を変動させるの特性を持たせることで，よりモジュラリティのある図の描画も期待できるため，Asymptote は流線を描画するプログラミング言語としてはさらなる品質の向上が期待できる．

謝辞

本研究を進めるにあたり，ご指導いただいた指導教員の横山哲郎准教授に心より感謝いたします．深くお礼を申し上げ，感謝の意を表します．

参考文献

[1] 荒井迅：トポロジカルな流れ構造の理解へ向けて，ながれ，Vol.33，No.1，pp.23–28 (2014).

[2] Yokoyama, T. and Sakajo, T.: Word representation of streamline topologies for structurally stable vortex flows in multiply connected domains. *Proc. Roy. Soc. A*,Vol.469,No.2150,pp.1–18 (2013).

[3] 加藤舞，内藤綾香，横山哲郎，横山知郎：円盤上の非圧縮流の反転の解析，情報処理学会　第 81 回全国大会講演論文集，Vol.1900，pp.319–120(2019).

[4] 横山知郎，坂上貴之，澤村陽一：二次元多重連結領域内における構造安定な非圧縮流れの文字列表現アルゴリズム，数理解析研究所講究録，Vol.J101-D．pp.11–25(2014).

[5] 横山哲郎，横山知郎：ハミルトン曲面流に対応する語の列挙アルゴリズム，電子情報通信学会論文誌 D，Vol.100，No.10，pp.892–894(2017).

[6] 横山哲郎，横山知郎：ハミルトン曲面流に対応する流れの向きを考慮した極大語の列挙アルゴリズム，電子情報通信学会論文誌 D，Vol.101，No.8，pp.1220–1222(2018).

[7] 加藤舞，内藤綾香：多重連結領域上の安定非圧縮流の解析，南山大学 2018 年度卒業論文 (2019).

[8] 梅田裕平：データの形が教えてくれること-トポロジカル・データ・アナリシスとその応用-，情報処理，Vol.57，No.11，pp. 1122–1127(2016).

[9] 結城浩：Java 言語で学ぶデザインパターン入門，SB クリエイティブ (2001).

付録 A

プログラムリスト

Python のプログラム

以下は自動描画を実現する Python のプログラムである．プログラム A.1 は木構造を生成する yacc モジュールを扱うプログラムである．プログラム A.2 は字句解析を行うプログラムである．プログラム A.3 は main クラス以外のクラスを扱うプログラムである．プログラム A.4 は main クラスを扱い，プログラム実行時にはこれが呼び出される．

プログラム A.1　yacc.py

```
1  # Yacc. PLYを用いる
2
3  import ply.yacc as yacc
4  from .lex import tokens
5  import sys
6  from . import flow
7
8  def p_s(p):
9       '''s : A0 '(' as ')'
10           | B0_PLUS '(' b_plus ',' '(' cs_minus ')' ')'
11           | B0_MINUS '(' b_minus ',' '(' cs_plus ')' ')' '''
12      if p[1] == 'a0': p[0] = flow.A0(p[3])
13      elif p[1] == 'b0+': p[0] = flow.B0_plus(p[3], p[6])
14      elif p[1] == 'b0-': p[0] = flow.B0_minus(p[3], p[6])
15
16 def p_as(p):
17      '''as : NIL
18            | CONS '(' a ',' as ')' '''
19      if p[1] == 'n': p[0] = flow.Nil()
20      elif p[1] == 'cons': p[0] = flow.Cons(p[3], p[5])
21
22 def p_a(p):
23      '''a : A_PLUS '(' b_plus ')'
24           | A_MINUS '(' b_minus ')'
25           | A2 '(' cs_plus ',' cs_minus ')' '''
26      if p[1] == 'a+': p[0] = flow.A_plus(p[3])
27      elif p[1] == 'a-': p[0] = flow.A_minus(p[3])
28      elif p[1] == 'a2': p[0] = flow.A2(p[3],p[5])
29
```

```
30 def p_b_plus(p):
31     '''b_plus : LEAF
32               | B_PLUS_PLUS '(' b_plus ',' b_plus ')'
33               | B_PLUS_MINUS '(' b_plus ',' b_minus ')'
34               | BETA_PLUS '(' cs_plus ')' '''
35     if p[1] == 'l': p[0] = flow.Leaf()
36     elif p[1] == 'b++': p[0] = flow.B_plus_plus(p[3], p[5])
37     elif p[1] == 'b+-': p[0] = flow.B_plus_minus(p[3], p[5])
38     elif p[1] == 'be+': p[0] = flow.Beta_plus(p[3])
39
40 def p_b_minus(p):
41     '''b_minus : LEAF
42                | B_MINUS_MINUS '(' b_minus ',' b_minus ')'
43                | B_MINUS_PLUS '(' b_minus ',' b_plus ')'
44                | BETA_MINUS '(' cs_minus ')' '''
45     if p[1] == 'l': p[0] = flow.Leaf()
46     elif p[1] == 'b--': p[0] = flow.B_minus_minus(p[3], p[5])
47     elif p[1] == 'b-+': p[0] = flow.B_minus_plus(p[3], p[5])
48     elif p[1] == 'be-': p[0] = flow.Beta_minus(p[3])
49
50 def p_c_plus(p):
51     '''c_plus : C_PLUS '(' b_plus ',' cs_minus ')' '''
52     p[0] = flow.C_plus(p[3], p[5])
53
54 def p_c_minus(p):
55     '''c_minus : C_MINUS '(' b_minus ',' cs_plus ')' '''
56     p[0] = flow.C_minus(p[3], p[5])
57
58 def p_cs_plus(p):
59     '''cs_plus : NIL
60                | CONS '(' c_plus ',' cs_plus ')' '''
61     if p[1] == 'n': p[0] = flow.Nil()
62     elif p[1] == 'cons': p[0] = flow.Cons(p[3], p[5])
63
64 def p_cs_minus(p):
65     '''cs_minus : NIL
66                 | CONS '(' c_minus ',' cs_minus ')' '''
67     if p[1] == 'n': p[0] = flow.Nil()
68     elif p[1] == 'cons': p[0] = flow.Cons(p[3], p[5])
69
70 def p_error(p):
71     print ('Syntax error in input %s' %p)
72
73 parser = yacc.yacc()
74
75 def parse(data):
76     return yacc.parse(data)
77
78 if __name__ == '__main__':
```

```
79      while True:
80          try:
81              s = input('>>> ')
82          except EOFError:
83              break
84          parser.parse(s).draw()
85          break
```

プログラム A.2　lex.py

```
1  # lexerを生成する. PLYを用いる.
2
3  import ply.lex as lex
4
5  tokens = (
6      'A0', 'B0_PLUS', 'B0_MINUS',
7      'A_PLUS', 'A_MINUS', 'A2',
8      'B_PLUS_PLUS', 'B_PLUS_MINUS', 'B_MINUS_PLUS', 'B_MINUS_MINUS',
9      'BETA_PLUS', 'BETA_MINUS',
10     'C_PLUS', 'C_MINUS',
11     'CONS', 'NIL', 'LEAF',
12 )
13
14 literals = "(),"
15
16 t_A0 = r'a0'
17 t_B0_PLUS = r'b0\+'
18 t_B0_MINUS = r'b0\-'
19
20 t_A_PLUS = r'a\+'
21 t_A_MINUS = r'a\-'
22 t_A2 = r'a2'
23
24 t_B_PLUS_PLUS = r'b\+\+'
25 t_B_PLUS_MINUS = r'b\+\-'
26 t_B_MINUS_PLUS = r'b\-\+'
27 t_B_MINUS_MINUS = r'b\-\-'
28
29 t_BETA_PLUS = r'be\+'
30 t_BETA_MINUS = r'be\-'
31
32 t_C_PLUS = r'c\+'
33 t_C_MINUS = r'c\-'
34
35 t_CONS = r'cons'
36 t_NIL = r'n'
37 t_LEAF = r'l'
38
39 t_ignore = ' \t\n' # 入力を無視する
40 t_ignore_COMMENT = r'\#.*' # コメントを無視する
```

```
41
42 # error handling
43 def t_error(t):
44     print("不正な文字 '%s'" % t.value[0])
45     t.lexer.skip(1)
46
47 lexer = lex.lex()
48
49 if __name__ == '__main__':
50     lex.runmain()
```

プログラム A.3 flow.py

```
1 # -*- coding: utf-8 -*-
2
3 import abc
4
5 import math
6 import numpy as np
7 import matplotlib.pyplot as plt
8 import matplotlib as mpl
9 from scipy import interpolate
10
11 def theta_point(theta, r, center):
12     """
13     半径とthetaと中心点を使って二次元上の点の位置を求める関数
14     """
15     return (r * math.cos(theta) + center[0], r * math.sin(theta) + center[1])
16
17 def c_list_high(children_ocu_list):
18     """
19     C系の配列[(self.high,self.bottom_length),...]から最も大きい高さを求める関数
20     """
21     return max(map(lambda x: x[0], children_ocu_list))
22
23 def c_list_circ_length(children_ocu_list, margin):
24     """
25     C系の配列[(self.high,self.bottom_length),...]から円周を求める関数
26     """
27     # marginはc同士の間に空けたいスペース
28     circ_length = 0 # 円周を保存する変数
29     longest_child = 0 # 最も長い子供の長さを保存する変数
30     for child in children_ocu_list:
31         circ_length += child[1]+margin # スペース分円周を伸ばす
32         if longest_child < child[1]:
33             longest_child = child[1]
34     # もし円周の半分以上の長さを持つ子供がいれば、円周の長さをその子供に合わせる
35     #（C系とb系が重なることを避ける）
36     if circ_length/2 <= longest_child:
37         circ_length = longest_child*2
```

```
38     return circ_length
39
40 def make_list_for_c(children_ocu_list, parent_r, parent_center, parent_type, margin,
       parent_length=0, first_child=False):
41     """
42     Cをdrawするための配列[[基準点からの距離,親の半径,親の中心,親がB0かどうか],...]を作成する関数
43     """
44     # parent_lengthは親の円の特定の位置から書き始めたいとき用(a2など)
45     c_list = []
46     length = parent_length
47     if parent_type and first_child:
48         length += 0.3
49         for child in children_ocu_list:
50             # 子供それぞれについて円周の基準点からどれだけ離れているかと、betaの半径、betaの中心、
                   親がB0かどうか
51             c_list.append({"length":length, "parent_r":parent_r, "parent_center":
                   parent_center, "parent_type":parent_type})
52             if length+(margin/len(children_ocu_list))-child[1] < length:
53                 length += 1.5
54             else:
55                 length += (margin/len(children_ocu_list))-child[1]+1
56     else:
57         for child in children_ocu_list:
58             length += margin
59             c_list.append({"length":length, "parent_r":parent_r, "parent_center":
                   parent_center, "parent_type":parent_type})
60             length += child[1]
61     return c_list
62
63 class Canvas:
64     """
65     図を描画する領域
66     """
67     def __init__(self):
68         """
69         matplotlibの初期化設定
70         """
71         self.ax = plt.axes()
72         plt.axis('off')
73         self.ax.set_aspect('equal')
74
75     def show_canvas(self):
76         """
77         作成された図を表示
78         """
79         plt.tight_layout()
80         plt.show()
81
82     def save_canvas(self, file_name):
```

```
83          """
84          作成された画像を保存
85          """
86          print("save picture! ")
87          plt.tight_layout()
88          plt.savefig(file_name)
89
90      def clear_canvas(self):
91          """
92          matplotlibのデータ削除
93          """
94          plt.close("all")
95          plt.cla()
96          plt.axis('off')
97          self.ax.set_aspect('equal')
98
99      def spline(self, x, y, point, deg):
100         """
101         スプライン補間
102         """
103         tck, u = interpolate.splprep([x, y], k=deg, s=0)
104         u = np.linspace(0, 1, num=point, endpoint=True)
105         spline = interpolate.splev(u, tck)
106         return spline[0], spline[1]
107
108     def draw_spline(self, xy):
109         """
110         スプライン補間関数、引数はx座標y座標のタプルのリスト
111         """
112         count = len(xy)
113         x = []
114         y = []
115         for i in range(0, count):
116             a_xy = xy[i]
117             x.append(a_xy[0])
118             y.append(a_xy[1])
119         if count >= 4:
120             a, b = self.spline(x, y, 100, 3)
121         elif count == 3:
122             a, b = self.spline(x, y, 100, 2)
123         plt.plot(a, b, color="black")
124
125     def draw_circle(self, r, center=(0, 0), circle_fill=False, fc="grey"):
126         """
127         円描画、引数centerはタプル
128         """
129         if circle_fill:
130             circ = plt.Circle(center, r, ec="black", fc=fc, linewidth=1.5)
131         else:
```

```
132                circ = plt.Circle(center, r, ec="black", fill=False, linewidth=1.5)
133            self.ax.add_patch(circ)
134            self.ax.plot()
135
136        def draw_arrow(self, center, theta=0):
137            """
138            矢印を描画する
139            """
140            # theta=0で右向きの矢印
141            col = 'k'
142            arst = 'wedge,tail_width=0.6,shrink_factor=0.5'
143            plt.annotate('', xy=(center[0]+(0.1*math.cos(theta)), center[1]+(0.05*math.sin
                   (theta))), xytext=(center[0]+(0.1*math.cos(math.pi+theta)), center
                   [1]+(0.1*math.sin(math.pi+theta))), arrowprops=dict(arrowstyle=arst,
                   connectionstyle='arc3', facecolor=col, edgecolor=col, shrinkA=0, shrinkB=0))
144
145        def draw_point(self, center):
146            """
147            zoomした際大きさが変化する点をプロットする関数
148            """
149            plt.plot([center[0]], [center[1]], 'k.')
150
151        def draw_line(self, xy_1, xy_2):
152            """
153            xy_1からxy_2まで直線を引く関数
154            """
155            plt.plot([xy_1[0], xy_2[0]], [xy_1[1], xy_2[1]], 'k-')
156
157        def axvspan(self, r):
158            """
159            半径rの周りを塗りつぶす関数
160            """
161            self.ax.axvspan(-r, r, -r, r, color="gray", alpha=0.5)
162
163    class Node(object, metaclass=abc.ABCMeta):
164        @abc.abstractmethod
165        def __init__(self):
166            self.canvas = None
167            self.head = None
168            self.tail = None
169
170        def draw(self, *arg):
171            """
172            描画処理を行うメソッド
173            """
174            pass
175
176        def plot_arrow(self, *arg):
177            """
```

```
178         矢印などの描画を行う
179         """
180         pass
181 
182     def set_canvas(self, canvas):
183         """
184         描画を行うキャンバスを指定する
185         """
186         self.canvas = canvas
187         if self.head is not None:
188             self.head.set_canvas(canvas)
189         if self.tail is not None:
190             self.tail.set_canvas(canvas)
191 
192 class A0(Node):
193     """
194     A0を扱うクラス
195     """
196     def __init__(self, head): # 子の半径を定義
197         super().__init__()
198         self.type = "A0"
199         self.head = head # 抽象構文木の作成
200         self.margin = 0.5
201 
202     def draw(self):
203         long_child = 0
204         childrens_info = [] # drawで引数として渡す
205         count_r = 0
206         if self.head.type == "Nil":
207             # 一様流を書く
208             self.canvas.draw_line((-1, 0), (1, 0))
209             self.canvas.draw_arrow((0, 0), math.pi)
210         else:
211             for child in self.head.occupation: # 子供達の中で一番長いrを求める
212                 if child[0] > long_child:
213                     long_child = child[0]
214             edge = long_child + self.margin
215             for child in self.head.occupation:
216                 # 次の子供の中心点をy軸に-r*2して繰り返す
217                 count_r += child[0] + self.margin
218                 # 子供それぞれについて中心点を作成して配列に格納
219                 childrens_info.append({"center":(0, -count_r), "edge":edge})
220                 count_r += child[0] + self.margin
221         self.head.draw(childrens_info)
222 
223 class B0(Node):
224     """
225     B0+,B0-の抽象クラス
226     """
```

```
227     def __init__(self, head, tail):
228         super().__init__()
229         self.head = head
230         self.tail = tail
231         self.margin = 0.5
232         high_children = c_list_high(tail.occupation)
233         children_length = c_list_circ_length(tail.occupation, self.margin)
234         self.r = max(children_length / (2 * math.pi), head.r + high_children + self.
            margin)
235 
236     def draw(self):
237         side_r = self.r + self.margin
238         self.canvas.axvspan(side_r)
239         self.canvas.draw_circle(self.r, (0, 0), circle_fill=True, fc="white")
240         self.plot_arrow()
241         for_children = make_list_for_c(self.tail.occupation, self.r, (0, 0), True, 2*
            self.r*math.pi, first_child=True)
242         self.head.draw((0, 0))
243         self.tail.draw(for_children)
244 
245 class B0_plus(B0):
246     """
247     B0+を扱うクラス
248     """
249     def plot_arrow(self):
250         self.canvas.draw_arrow((self.r, 0), math.pi/2)
251 
252 class B0_minus(B0):
253     """
254     B0-を扱うクラス
255     """
256     def plot_arrow(self):
257         self.canvas.draw_arrow((self.r, 0), math.pi*1.5)
258 
259 class A_Flip(Node):
260     """
261     a+,a-の抽象クラス
262     """
263     def __init__(self, head):
264         super().__init__()
265         self.head = head
266         self.margin = 0.5 # 子の専有領域と親の領域の余白
267         self.r = head.r + self.margin
268 
269     def draw(self, info_dic): # 描画する際に親から与える中心点
270         center = info_dic["center"]
271         edge = info_dic["edge"]
272         self.canvas.draw_circle(self.r, center)
273         self.plot_arrow(center,edge)
```

```
274         self.head.draw(center)
275
276 class A_plus(A_Flip):
277     """
278     a+を扱うクラス
279     """
280     def __init__(self, head):
281         super().__init__(head)
282         self.type = "A_plus"
283         self.occupation = [(self.r, self.type)]
284
285     def plot_arrow(self, center, edge):
286         self.canvas.draw_point((center[0], center[1]-self.r))
287         self.canvas.draw_arrow((center[0]-self.r, center[1]), theta=math.pi*1.5)
288         self.canvas.draw_arrow((center[0]+self.r, center[1]), theta=math.pi/2)
289         self.canvas.draw_line((-edge, center[1]-self.r), (edge, center[1]-self.r))
290         self.canvas.draw_arrow((-edge/2, center[1]-self.r), math.pi)
291         self.canvas.draw_arrow((edge/2, center[1]-self.r), math.pi)
292
293 class A_minus(A_Flip):
294     """
295     a-を扱うクラス
296     """
297     def __init__(self, head):
298         super().__init__(head)
299         self.type = "A_minus"
300         self.occupation = [(self.r, self.type)]
301
302     def plot_arrow(self, center, edge):
303         self.canvas.draw_point((center[0], center[1]+self.r))
304         self.canvas.draw_arrow((center[0]-self.r, center[1]), theta=math.pi/2)
305         self.canvas.draw_arrow((center[0]+self.r, center[1]), theta=math.pi*1.5)
306         self.canvas.draw_line((-edge, center[1]+self.r), (edge, center[1]+self.r))
307         self.canvas.draw_arrow((-edge/2, center[1]+self.r), math.pi)
308         self.canvas.draw_arrow((edge/2, center[1]+self.r), math.pi)
309
310 class A2(Node):
311     """
312     a2を扱うクラス
313     """
314     def __init__(self, head, tail):
315         super().__init__()
316         self.type = "A2"
317         self.head = head
318         self.tail = tail
319         self.margin = 0.5 # 子同士のスペースの定義
320         self.high = max(c_list_high(head.occupation), c_list_high(tail.occupation))
321         len_of_plus_circ = c_list_circ_length(head.occupation, self.margin) + self.margin
                # plus回りの長さ
```

```
322         len_of_minus_circ = c_list_circ_length(tail.occupation, self.margin) + self.
                margin # minus回りの長さ
323         if len_of_plus_circ >= len_of_minus_circ:
324             self.len_of_circ = len_of_plus_circ * 2
325         else:
326             self.len_of_circ = len_of_minus_circ * 2
327         self.center_r = self.len_of_circ / (2 * math.pi) # a_2の円の半径
328         self.r = self.center_r + self.high # 専有領域の半径
329         self.occupation = [(self.r, self.type)]
330 
331     def draw(self, info_dic):
332         center = info_dic["center"]
333         edge = info_dic["edge"]
334         self.canvas.draw_circle(self.center_r, center, circle_fill=True) # a_2の描画
335         self.canvas.draw_point((center[0]+self.center_r, center[1])) # 一様流との交点の描
                画(右)
336         self.canvas.draw_point((center[0]-self.center_r, center[1])) # 一様流との交点の描
                画(左)
337         self.canvas.draw_line((center[0]-self.r, center[1]), (center[0]-self.center_r,
                center[1]))
338         self.canvas.draw_line((center[0]+self.center_r, center[1]), (center[0]+self.r,
                center[1]))
339         self.canvas.draw_line((-edge, center[1]), (-self.r, center[1]))
340         self.canvas.draw_line((self.r, center[1]), (edge, center[1]))
341         self.canvas.draw_arrow(((-edge-self.r)/2, center[1]), math.pi)
342         self.canvas.draw_arrow(((self.r+edge)/2, center[1]), math.pi)
343         for_plus_children = make_list_for_c(self.head.occupation, self.center_r, center,
                False, self.margin)
344         for_minus_children = make_list_for_c(self.tail.occupation, self.center_r, center,
                 False, self.margin, parent_length=self.len_of_circ/2)
345         self.head.draw(for_plus_children)
346         self.tail.draw(for_minus_children)
347 
348 class Cons(Node):
349     """
350     consを扱うクラス
351     """
352     def __init__(self, head, tail):
353         super().__init__()
354         self.head = head
355         self.tail = tail
356         self.type = head.type
357         head_child = [s for s in head.occupation if s != (0, 0)]
358         tail_child = [s for s in tail.occupation if s != (0, 0)]
359         self.occupation = []
360         for child in head_child:
361                 self.occupation.append(child)
362         for child in tail_child:
363                 self.occupation.append(child)
```

```
364
365     def draw(self, children_list):
366         self.head.draw(children_list.pop(0))
367         if len(children_list) != 0:
368             self.tail.draw(children_list)
369
370 class Nil(Node):
371     """
372     nilを扱うクラス
373     """
374     def __init__(self):
375         super().__init__()
376         self.type = "Nil"
377         self.occupation = [(0, 0)]
378
379 class Leaf(Node):
380     """
381     leafを扱うクラス
382     """
383     def __init__(self):
384         super().__init__()
385         self.r = 0
386
387 class B_Evc(Node):
388     """
389     b++,b--の抽象クラス
390     """
391     def __init__(self, head, tail):
392         super().__init__()
393         # headは上の円の半径、tailは下の円の半径
394         self.head = head
395         self.tail = tail
396         self.margin = 0.5 # 子の専有領域と親の領域の余白
397         self.l_up_r = head.r # 上の図の占有領域(半径)
398         self.l_down_r = tail.r # 下の図の占有領域(半径)
399         self.r = (2 * self.l_up_r + 2 * self.l_down_r + 4 * self.margin) / 2 # 全体の占
                有領域(半径)
400
401     def draw(self, center=(0, 0)): # 描画する際に親から与える中心点
402         self.canvas.draw_point((center[0], self.l_down_r+center[1]-self.l_up_r)) # 2つ
                の円の交点
403         self.canvas.draw_circle(self.l_up_r+self.margin, (center[0], self.l_down_r+self.
                margin+center[1])) # 上の円
404         self.canvas.draw_circle(self.l_down_r+self.margin, (center[0], -self.l_up_r-self
                .margin+center[1])) # 下の円
405         self.plot_arrow(center)
406         self.head.draw((center[0], self.l_down_r+self.margin+center[1]))
407         self.tail.draw((center[0], -self.l_up_r-self.margin+center[1]))
408
```

```
409 class B_Flip(Node):
410     """
411     b+-,b--の抽象クラス
412     """
413     def __init__(self, head, tail):
414         super().__init__()
415         self.head = head
416         self.tail = tail
417         self.margin = 0.5 # 子の専有領域と親の領域の余白
418         self.l_up_r = head.r # 上の図の占有領域(半径)
419         self.l_down_r = tail.r # 下の図の占有領域(半径)
420         self.r = (2 * self.l_up_r + 2 * self.l_down_r + 4 * self.margin) / 2
421 
422     def draw(self, center=(0, 0)): # 描画する際に親から与える中心点
423         self.canvas.draw_circle(self.l_up_r+self.margin, (center[0], self.l_down_r+self.
                margin+center[1]))
424         self.canvas.draw_circle(self.l_up_r+self.l_down_r+2*self.margin, center)
425         self.canvas.draw_point((center[0], self.l_down_r+self.margin+center[1]+self.
                l_up_r+self.margin))
426         self.plot_arrow(center)
427         self.head.draw((center[0], self.l_down_r+self.margin+center[1]))
428         self.tail.draw((center[0], -self.l_up_r-self.margin+center[1]))
429 
430 class Beta(Node):
431     """
432     beta+,beta-の抽象クラス
433     """
434     def __init__(self, head):
435         super().__init__()
436         self.head = head
437         self.margin = 0.5 # 要素の両脇に作るスペースの大きさ
438         high_children = c_list_high(head.occupation)
439         children_length = c_list_circ_length(head.occupation, self.margin)
440         self.center_r = children_length / (2 * math.pi) # betaの円
441         if children_length < 1:
442             self.center_r = 7 / (2 * math.pi)
443         self.r = self.center_r + high_children # 親に渡す全体の大きさ
444 
445     def draw(self, center):
446         self.canvas.draw_circle(self.center_r, center, circle_fill=True)
447         for_children = make_list_for_c(self.head.occupation, self.center_r, center, False
                , self.margin)
448         self.plot_arrow(center)
449         self.head.draw(for_children)
450 
451 class B_plus_plus(B_Evc):
452     """
453     b++を扱うクラス
454     """
```

```
455     def plot_arrow(self, center):
456         # 上の円の矢印
457         self.canvas.draw_arrow((center[0], self.l_down_r+2*self.margin+center[1]+self.
            l_up_r), math.pi)
458         # 下の円の矢印
459         self.canvas.draw_arrow((center[0], -self.l_up_r-2*self.margin+center[1]-self.
            l_down_r), 0)
460
461 class B_plus_minus(B_Flip):
462     """
463     b+-を扱うクラス
464     """
465     def plot_arrow(self, center):
466         self.canvas.draw_arrow((center[0], self.l_down_r+self.margin+center[1]-self.
            l_up_r-self.margin), theta=math.pi)
467         self.canvas.draw_arrow((center[0], center[1]-(self.l_up_r+self.l_down_r+2*self.
            margin)))
468
469 class Beta_plus(Beta):
470     """
471     beta+を扱うクラス
472     """
473     def plot_arrow(self, center):
474         self.canvas.draw_arrow((center[0]+self.center_r, center[1]), math.pi/2)
475
476 class B_minus_minus(B_Evc):
477     """
478     b--を扱うクラス
479     """
480     def plot_arrow(self, center):
481         self.canvas.draw_arrow((center[0], self.tail.r+2*self.margin+center[1]+self.head
            .r), 0)
482         self.canvas.draw_arrow((center[0],-self.head.r-2*self.margin+center[1]-self.tail
            .r), math.pi)
483
484 class B_minus_plus(B_Flip):
485     """
486     b-+を扱うクラス
487     """
488     def plot_arrow(self, center):
489         self.canvas.draw_arrow((center[0], self.l_down_r+self.margin+center[1]-self.
            l_up_r-self.margin), theta=0)
490         self.canvas.draw_arrow((center[0], center[1]-(self.l_up_r+self.l_down_r+2*self.
            margin)), theta=math.pi)
491
492 class Beta_minus(Beta):
493     """
494     beta-を扱うクラス
495     """
```

```
496     def plot_arrow(self, center):
497         self.canvas.draw_arrow((center[0]+self.center_r, center[1]), math.pi*1.5)
498 
499 class C(Node):
500     """
501     c+,c-の抽象クラス
502     """
503     def __init__(self, head, tail):
504         super().__init__()
505         self.head = head
506         self.tail = tail
507         self.margin = 1 # c系の要素の両脇に作るスペースの大きさ
508         self.circ_margin = 0.5 # 子のb系の要素と親の間の距離
509         self.high_children = c_list_high(tail.occupation)
510         self.children_length = c_list_circ_length(tail.occupation, self.margin)
511         bottom_length = max(head.r*2, self.children_length)
512         self.high = 2 * head.r + self.high_children + self.margin
513         if (self.head.r == 0) and (len(self.tail.occupation) != 1):
514             self.high += len(self.tail.occupation) * 1
515         self.occupation = [(self.high, bottom_length)]
516 
517     def draw(self, c_data):
518         if self.high_children == 0:
519             self.high_children = 0.3
520         length = c_data["length"]
521         center_r = c_data["parent_r"]
522         center = c_data["parent_center"]
523         bool_b0 = c_data["parent_type"]
524         start_theta = length / center_r
525         start_point = theta_point(start_theta, center_r, center)
526         end_theta = (length + self.children_length) / center_r
527         end_point = theta_point(end_theta, center_r, center)
528         high_theta = (end_theta-start_theta) / 2 + start_theta
529         if bool_b0:
530             high_point = theta_point(high_theta, center_r-self.high, center)
531             b_center = theta_point(high_theta, center_r-self.high_children-self.
                    circ_margin-self.head.r, center)
532         else:
533             high_point = theta_point(high_theta, center_r+self.high, center)
534             b_center = theta_point(high_theta, center_r+self.high_children+self.
                    circ_margin+self.head.r, center)
535         self.plot_arrow(bool_b0, high_point, high_theta)
536         if self.head.r != 0:
537             # 180-(90+high_theta)bの専有領域の中心を基準に三角関数を適用するための準備
538             b_r_theta = math.pi - (math.pi/2+high_theta)
539             # 0度の点
540             b_r_center = theta_point(-b_r_theta, self.head.r+self.circ_margin, b_center)
541             # 180度の点
542             b_l_center = theta_point(math.pi-b_r_theta, self.head.r+self.circ_margin,
```

```
                b_center)
543             if self.head.r * 2 < self.children_length / 2:
544                 self.canvas.draw_spline([start_point, high_point, end_point])
545             else:
546                 self.canvas.draw_spline([start_point, b_r_center, high_point, b_l_center,
                        end_point])
547         else:
548             self.canvas.draw_spline([start_point, high_point, end_point])
549         self.canvas.draw_point(start_point)
550         self.canvas.draw_point(end_point)
551         for_children = make_list_for_c(self.tail.occupation, center_r, center, bool_b0,
                self.margin/1.5, parent_length=length)
552         self.head.draw(b_center)
553         self.tail.draw(for_children)
554 
555 class C_plus(C):
556     """
557     c+を扱うクラス
558     """
559     def __init__(self, head, tail):
560         super().__init__(head, tail)
561         self.type = "C_plus"
562 
563     def plot_arrow(self, bool_b0, high_point, high_theta):
564         self.canvas.draw_arrow(high_point, high_theta+ math.pi*(1.5 if bool_b0 else 0.5)
                )
565 
566 class C_minus(C):
567     """
568     c-を扱うクラス
569     """
570     def __init__(self, head, tail):
571         super().__init__(head, tail)
572         self.type = "C_minus"
573 
574     def plot_arrow(self, bool_b0, high_point, high_theta):
575         self.canvas.draw_arrow(high_point, high_theta+ math.pi*(0.5 if bool_b0 else 1.5)
                )
```

プログラム A.4 main.py

```
1 """
2 Visualization program of tree representation of structurally stable incompressible flow
      in two dimensional multiply-connected domain
3 """
4 # -*- coding: utf-8 -*-
5 
6 import os
7 from src import flow, yacc
8 from src.flow import Canvas
```

```
9
10 """
11 入力した木表現に対する流線を表示
12 """
13 def main():
14     while True:
15         try:
16             s = input('>>> ')
17             object = yacc.parser.parse(s)
18             canvas = Canvas()
19             object.set_canvas(canvas)
20             object.draw()
21             print("draw successful!")
22             print("You can save picture or watch in matplotlib:"+"\n"+"If you want to
                    save, please type \"save\"."+"\n"+"If you want to watch, please type \"
                    watch\".")
23             type = input(':')
24             if type == "save":
25                 dirname = "flow_picture/"
26                 os.makedirs(dirname, exist_ok=True)
27                 filename = dirname + s + '.png'
28                 canvas.save_canvas(filename)
29             elif type == "watch":
30                 canvas.show_canvas()
31             else:
32                 pass
33             canvas.clear_canvas()
34         except AttributeError:
35             print("please type correct syntax.")
36         except EOFError:
37             break
38
39 if __name__ == "__main__":
40     main()
```

Asymptote のプログラム

プログラム A.5 fin.asy

```
1 settings.outformat = "pdf";
2 unitsize(1cm);
3
4 struct B {
5     pair x;
6     real r;
7     bool3 d,d2;
8     B in;
9
10     void operator init(pair xx, real rr) {
```

```
11          x = xx;
12          r = rr;
13      }
14
15       void operator init(pair xx, real rr,bool3 dd) {
16          x = xx;
17          r = rr;
18          d = dd;
19      }
20
21      void operator init(pair xx, real rr, bool3 dd, bool3 ddd) {
22          x = xx;
23          r = rr;
24          d = dd;
25          d2 = ddd;
26      }
27
28
29      void mydraw() {
30          if (r != 0) {
31              if(d == false && d2 ==false){ /* b-- */
32              draw(reverse(circle(x+(0,r), r)), arrow=MidArrow(HookHead));
33              draw(reverse(circle(x+(0,-r), r)), arrow=MidArrow(HookHead));
34              }
35              else if(d == false && d2 == true){ /* b-+ */
36              draw(reverse(circle(x+(0,r), r)), arrow=MidArrow(HookHead));
37              draw((circle(x+(0,r*0.6), r*0.6)), arrow=MidArrow(HookHead));
38              }
39              else if(d == true && d2 == false){ /* b-+ */
40              draw((circle(x+(0,r), r)), arrow=MidArrow(HookHead));
41              draw(reverse(circle(x+(0,r*0.6), r*0.6)), arrow=MidArrow(HookHead));
42              }
43              else { /* b++ */
44              draw((circle(x+(0,r), r)), arrow=MidArrow(HookHead));
45              draw((circle(x+(0,-r), r)), arrow=MidArrow(HookHead));
46              }
47              dot(x);
48              if (in != null) {
49              in.mydraw();
50              }
51
52          }
53      }
54
55      void scale(real a) {
56          r *= a;
57      }
58  }
59
```

```
60 struct A{
61     pair x;
62     real r;
63     bool3 d;
64     B in;
65 
66 
67     void operator init(pair xx, real rr) {
68         x = xx;
69         r = rr;
70     }
71 
72     void operator init(pair xx, real rr, bool3 dd) {
73         x = xx;
74         r = rr;
75         d = dd;
76     }
77 
78     void mydraw() {
79         if(d == false){
80         draw(x+(-r,0) -- x+(0,0), arrow=MidArrow(HookHead));
81         draw(x+(0,0) -- x+(r,0), arrow=MidArrow(HookHead));
82         draw(reverse(circle(x+(0,r), r)), arrow=MidArrow(HookHead));
83         dot(x+(0,r));
84         dot(x);
85 
86         } else if(d == true){
87         draw(x+(-r,0) -- x+(0,0), arrow=MidArrow(HookHead));
88         draw(x+(0,0) -- x+(r,0), arrow=MidArrow(HookHead));
89         draw((circle(x+(0,-r), r)), arrow=MidArrow(HookHead));
90         dot(x+(0,-r));
91         dot(x);
92         } else {
93         draw(x+(-2r,0) -- x+(-r,0),arrow=MidArrow(HookHead));
94         draw(x+(r,0) -- x+(2r,0),arrow=MidArrow(HookHead));
95         fill(circle(x, r),gray);
96         draw(circle(x, r), Arrow(Relative(0.77)));
97         draw(reverse(circle(x, r)), Arrow(Relative(0.77)));
98         }
99         if (in != null) {
100            in.mydraw();
101        }
102    }
103    void scale(real a) {
104        r *= a;
105        in.scale(a);
106    }
107 }
```

付録 B

実行例

Python の実行例

入力する文字列を容易にするため, ホール部分をそれぞれ置き換える．よって，実際に入力する文字列は以下のように変換し入力する．

表 B.1 文字列の対応

文字列	置き換え後の文字列	文字列	置き換え後の文字列
{	(	}	)
a_ϕ	a0	$b_{\phi-}$	b0
a_+	a+	a_-	a-
a_2	a2	b_{++}	b++
b_{+-}	b+-	b_{-+}	b-+
b_{--}	b–	β_+	be+
β_-	be-	c_+	c+
c_-	c-		

$a_\phi(cons(a_-(b_{--}(b_{-+}(l,\ l),\ l)),\ n))$ で表現された流線構造を描く例を下記に示す．

プログラム B.1 Python 実行例 1

```
1 >>> a0(cons(a-(b--(b-+(l,l),l)),n))  /* 木表現の入力*/
2 a0(cons(a-(b--(b-+(l,l),l)), n))  /* 木表現の表示*/
3 draw successful!
4 You can save picture or watch in matplotlib:  /* 保存か表示の選択 */
5 If you want to save, please type "save".
6 If you want to watch, please type "watch".
7 :watch
```

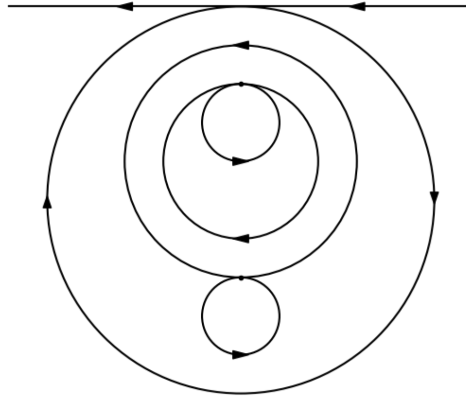

図 B.1　a0(cons(a-(b–(b-+(l,l),l)),n))

$b_{\phi+}(b_{+-}(b_{++}(l,\ l),\ l),\ (cons(\ c_{-}(l,\ n),\ cons(c_{-}(l,\ n),\ n))))$ で表現された流線構造を描く例を下記に示す.

プログラム B.2　Python 実行例 2

```
>>> b0+(b+-(b++(l,l),l),(cons(c-(l,n),cons(c-(l,n),n))))
b0+(b+-(b++(l,l),l),cons(c-(l,n), cons(c-(l,n), n)))
draw successful!
You can save picture or watch in matplotlib:
If you want to save, please type "save".
If you want to watch, please type "watch".
:watch
```

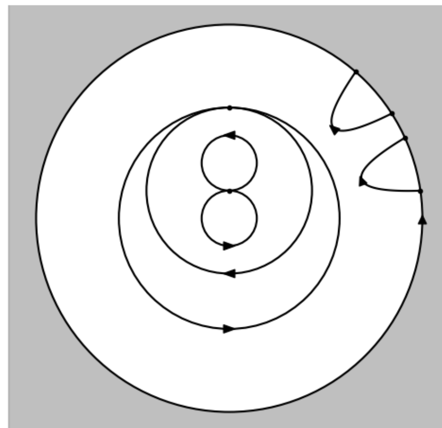

図 B.2　b0+(b+-(b++(l,l),l),(cons(c-(l,n),cons(c-(l,n),n))))

Asymptote の実行例

a_{+} の中に b_{++} が入った流線構造を描く例を下記に示す.

プログラム B.3　実行例

```
1 $ asy /* interactive mode */
2 Welcome to Asymptote version 2.41 (to view the manual, type help)
3 > input fin;  /* ライブラリの呼び出し */
4 > A ap = A((0,0),3,true);  /* $a_{+}$ */
5 > ap.in = B((0,-3),1.2,true,true); /* 中に入る$b_{++}$ */
6 > ap.mydraw();  /* 描画関数実行 */
```

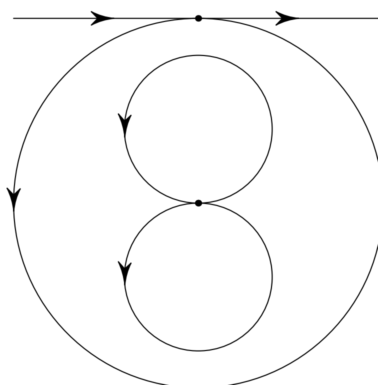

図 B.3　例