2004年度 卒論論文

幾何制約におけるCassowaryとハイブリッド並行制約プログラミングの性能評価

提出日: 2005年2月2日
指導: 上田和紀 教授

早稲田大学
理工学部 情報学科
学籍番号: 1G01P018-1
宇佐美　智史

目 次

図 目 次

表 目 次

概 要

(仮)2003 年度に石井氏、若槻氏により作成されたアニメーションツール Grifon は、従来のアニメーションツールとは異なり、ハイブリッド並行制約と幾何制約によってアニメーションを表現するものである。ハイブリッド並行制約は時間軸上の連続的・離散的変化を表すことが表すことが出来るため、時間的な処理を受け持つ。幾何制約は、例えば図形同士の幾何的な関係を表現する。Grifon における幾何制約は Cassowary という主にユーザインターフェイスを利用するために開発された幾何制約処理系を用いて実現しているのだが、ハイブリッド並行制約でも、オプションと指定によって簡単な幾何制約が実現出来ることが出来る。本研究では、このことに注目し、Cassowary で実装されている幾何制約がハイブリッド並行制約プログラミングによっても実装出来れば、幾何制約もハイブリッド並行制約プログラミングに処理を任せられるのではないかという考えの下、Cassowary に実装されている幾何制約をハイブリッド並行制約プログラミングで実装してみた（実際はまだ実装中ですが、なんとか出来そうかと思ってます）

本研究において、幾何制約がハイブリッド並行制約プログラミングによっても実現出来たことにより、アニメーションの観点からすると、幾何制約をハイブリッド並行制約プログラミングでも十分に実現出来ることがわかった。さらに、ハイブリッド並行制約プログラミングでは、自分で新たな制約を作り出すことが出来、そういう意味では柔軟なのではないかと考える。（ただし、時間のことも考えて制約をつくったりしないといけないので大変ではあるが … ）しかし、Cassowary は、制約に強さをつけることが出来る。つまり、制約に優先度があり、制約がすべて満たされなくても解を求められる。よって、すべての幾何制約をハイブリッド並行制約に任せてよいわけではなく、今後の Grifon ではそういうことを考えていきたいと思う。

第1章　研究の背景と目的*

1.1　研究の背景と目的

2003年度に石井氏、若槻氏によって高級な計算モデルに基づきアニメーションを柔軟に表現することを目指して、アニメーションツールGrifonが作成された。

そのGrifonでは、制約に基づいてアニメーションを表現している。制約は専門性の高い内容をシンプルかつ直感的に表現できる、モジュール化が容易で階層化や部品化を実現できる、といった特徴を持っている。

Grifonでは、アニメーション中の時間的変化をハイブリッド並行制約によって、グラフィックオブジェクトの空間的な関係を幾何制約によって表現している。ハイブリッドへ行く制約は時間軸に沿った変化の表現が可能なように並行制約を拡張した枠組みである。幾何制約はグラフィックオブジェクトの位置座標やサイズ、色などの属性値を変数として、それらが持つべき数値的な関係を表現している。

制約処理系はハイブリッド並行制約がハイブリッド並行制約プログラミング、幾何制約がCassowaryによって実現されている。しかし、ハイブリッド並行制約プログラミングでも幾何制約が実現できることがわかり、そのことに着目をしてみた。

本研究では、Cassowaryによって実現されている幾何制約をハイブリッド並行制約プログラミングによって、実現してみた。実現した上で、Cassowaryとそのハイブリッド並行制約プログラミングによって幾何制約を実現したものを比較することによって、Grifonが制約充足系において、今の環境をどのようにしたらよりよく出来るかを検証してみた。

1.2　本論文の構成

第2章　関連研究（仮）

2.0.1　Juno-2

インターフェイスは、マウス操作とテキストによる描画言語の表現が用意されている。どちらの操作を行っても、すぐにもう片方の表現反映される。どちらかというと言語表現の方に重点があり、専用の描画言語によって図形を表現する。

2.0.2　Udraw

1996 年に吉田法茂氏と内藤頼光氏により作成され、それを基にして 1998 年に柳吾郎氏と坂東雄人氏によって、改良が加えられた幾何学的制約処理に基づく描画ツールである。一度図形同士に制約をかけることによって、その後の処理は、ユーザは図形の幾何学的な関係を気にすることなく、自動的に図形をその制約にしたがって、変形させることが可能である。

2.0.3　Constrator

Constrator とは、２００２年に石井大輔氏が Udraw を基に、一から作成した幾何学的制約処理に基づく描画ツールである。つまり、ユーザが一度、図形に制約をかけることによって、その後、図形を移動したり、ドラッグしたりしてもその制約によって、図形同士の相互関係は保つことができるという描画ツールである。

第3章　Grifon（仮）

2003 年度に石井氏、若槻氏によって、高級な計算モデルに基づきアニメーションを柔軟に表現することを目指し、制約に基づくアニメーションツールとして作成されたのが Grifon である。Grifon では、アニメーションを、図形の属性が持つ規則を表す制約と、図形を表すデータ構造である PSVG を組み合わせて表現したものである。制約充足系ではアニメーションを実行するときの時間変化をハイブリッド並行制約で表現して、図形同士の空間的な関係を幾何制約で表現している。制約のドメインとして PSVG を用いている。 また、制約と PSVG を抽象化して取り出し、アニメーションライブラリとして管理するための設計を行っている。Grifon は JHotDraw、Rhino、BSF、Log4j、Cassowary 既存実装を利用している。Grifon では、ハイブリッド並行制約はハイブリッド並行制約プログラミングによって、幾何制約は Cassowary によって実現されている。

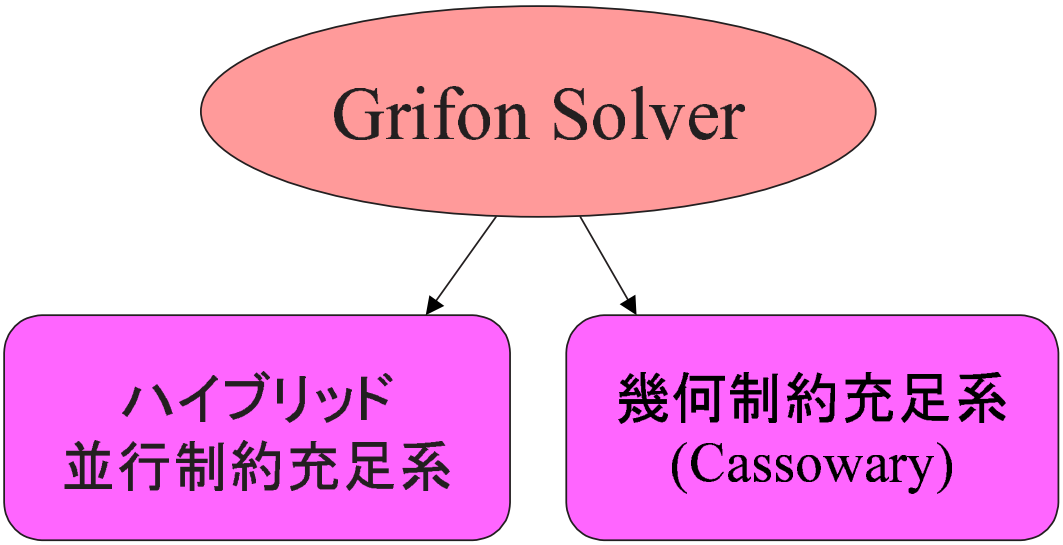

図 3.1: Grifon の制約処理系

3.1 JHotDraw

描画アプリケーションのフレームワークである。キャンバスや内部データモデルの実装に利用している。多くのデザインパターンにより構成されており、Smalltalk-80で開発されたグラフィックス・エディタのためのフレームワークであるHotDrawのJavaへのポートしたものである。また、LGPLライセンスでソースコードを公開している。

3.2 Rhino

JavaScriptのJavaによる実装。すべてがJavaで記述されたJavaScriptのオープンソースな実装である。JavaScriptのコンパイラやJavaアプリケーションのスクリプティングを行う機能が含まれている。一般的にはJavaアプリケーションへ組み込まれ、エンドユーザーによるスクリプトの記述が可能になる。

3.3 BSF

Java用スクリプト言語のラッパである。デバッグのためのスクリプティング機能を持たせるために使用している。

3.4 Log4j

Jakartaプロジェクトで開発が進められているJavaプログラム用のログAPIである。Log4Jを利用することで、様々なロギングが可能となる。またパフォーマンスにも十分に配慮されて設計されているので、ログの記録に用いるコードをプログラムソースに残したまま出荷しても、特に問題は起こらないようになっている。他にもLog4Jの優れた機能があり。ログを出力する部分をグループ化しておくことで、プログラム中の特定部分の動作情報だけを出力したり、出力先にファイル、OutputStream、java.io.Writer、リモートLog4Jサーバ、リモートUnix Syslogデーモン、NTイベントログなどを簡単に指定することができます。

3.5 Cassowary

おもにユーザインタフェースの制御を目的とした線形の実数地制約充足系である。幾何制約充足に利用している。詳しくは○章で述べる。

第4章　Cassowary（仮）

Cassowary とは、おもにユーザインターフェースを利用するために開発された、線型の等式、不等式制約の充足系である。制約充足は数値的に処理されるが、アルゴリズムとしてはシンプレックス法が用いられており、繰り返し処理を高速に扱うことができる手法が使われている。制約には強さというパラメータが付けられており、制約がすべて満たされない場合でも最適解を求めることが出来る。その強さは、required、strong、medium、weak の４種類があり、required の強さをもった制約は必ず満たされなければならないということになっている。

線形方程式・不等式制約は自然とユーザインターフェイス（特にレイアウトや他の幾何的な関係）を表現するのに用いられる。不等式制約は主に ”〜の内側 ”、”〜の上 ”、”〜の下 ”、”〜の左 ”、”〜の右 ”、といった関係を表現するのに必要となる。

例えば簡単な例を出すと、Web ドキュメントで、figure1 と figure2 があって、figure1 は figure2 の左側になるという要求を上記の不等式制約で表すと次のようになる。

$$\text{figure1.rightSide} \leqq \text{figure2.leftSide}$$

この不等式制約を説明すると、figure1 の右側が figure2 の左側の左にあれば、figure1 は figure2 の左側にあることが保証されるということを意味している。

制約システム（系？）では、要求を表現するのと同様に優位性を表現するのも重要である。

その一つの用途には、例えばある一つのイメージがあって、そのある部分が動いている場合に、その他の部分が動かないということを表現することである。その部分が動くをいう理由がない限りその部分はそのままの場所にいるということを表現できる。

※その二つ目の用途には優雅な方法で潜在的に無効のユーザ入力を処理することです。例えば、ユーザが制限の窓における外部の図を動かそうとするなら、図がまさしく窓の側面に直面して突き当たって、誤りを与えるよりむしろ止まるのは、妥当です。

※その三つ目の用途はグラフを広げる際に、衝突している要求のバランスをとることです。

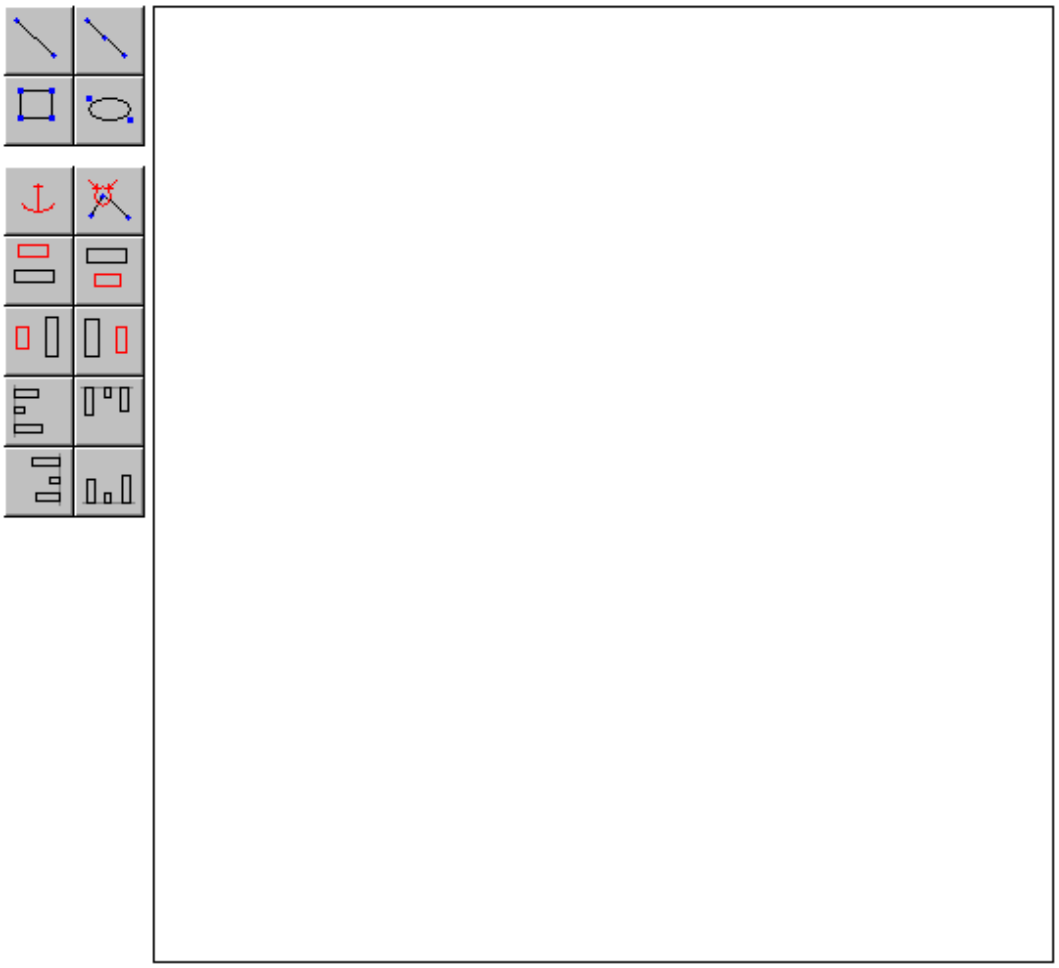

図 4.1: Cassowary

4.1 Cassowary で実装されている制約

Cassowary では大きく分けて 2 種類の幾何制約が実装されている。

オブジェクトとしては、Line(直線)、Midpoint Line(中点持ち直線)、Rectangle(長方形)、Oval(楕円) が実装されている。これらのオブジェクトはいずれも 2 点を指定することにより実現できる。

制約としては、Anchor(アンカー制約)、Colloccation(連結制約)、Adjacency(隣接制約)、Alignment(整列制約) が実装されている。Anchor は点に対してかける制約で、その他の制約は 2 つ以上のオブジェクト間にかける制約である。

4.1.1 オブジェクト

Line

オブジェクトとして、ある二点を指定することにより、直線をつくる制約である。

Midpoint Line

オブジェクトとして、ある二点を指定することにより。直線とその直線の中点をつくる制約である。

Rectangle

オブジェクトとして、ある二点を指定することにより、その二点の中点を対角線の交点とする長方形（正方形）をつくる制約である。

Oval

オブジェクトとして、ある二点を指定することにより、その二点の中点を中心として、その二点の中点を対角線の交点とする長方形（正方形）に接するような楕円をつくる制約である。

4.1.2 制約

Anchor(アンカー制約)

ある点を現在の位置に据えつけ続ける制約である。

Collocation(連結制約)

ある二点以上の点の中で確立される制約であり、その二点以上の点をある一点に集める制約である。

Adjacency(隣接制約)

２つのオブジェクト間で確立される制約である。上記の図の位置を保証するものである。

Alignment(整列制約)

２つ以上のオブジェクトが留まることを保証する制約である。そのオブジェクトの中のどれかが移動されたものとして、並べられます。

4.2 主要なクラス

主なクラスとその関係は以下の通りである。

```
Object
     ClAbstractVariable
  ClDummyVariable
  ClObjectiveVariable
  ClSlackVariable
  ClVariable
     ClConstraint
  ClEditOrStayConstraint
       ClEditConstraint
       ClStayConstraint
  ClLinearConstraint
       ClLinearEqualityConstraint
       ClLinearInequalityConstraint
             ClLinearExpression
     ClTableau
          ClSimplexSolver
             ClStrength
             ClSymbolicWeight
```

4.3 主要なメソッド

第5章　Hybrid並行制約プログラミング(仮)

Grifonの特徴の１つは、ハイブリッド並行制約プログラミングによって、アニメーション中の時間変化を表現したことである。本章では、ハイブリッド並行制約プログラミングの基礎となっている並行制約プログラミング、デフォルト並行制約プログラミングについて述べ、それに次いで、ハイブリッド並行制約プログラミングについて述べていく。

5.1　hybrid cc

5.1.1　並行制約プログラミング

並行制約プログラミングとは宣言的で並行なモデルを持ったプログラミングの枠組みのことである。並行制約プログラミングの各々の構成は論理的な式から成る。並行制約プログラミングは伝統的な考えである変数の評価としてのストアを、変数の正の値の制約としてのストアに取り替えた。そのストアは変数の正の値を制限する情報の集合で構成されている。並行制約プログラミングは並行に動作し、保有しているストアに影響を与えるエージェントのセットからなる。エージェントにはtellエージェントとaskエージェントの２種類があり、 tellエージェントはストアに情報を与えるもので、askエージェントは情報の妥当性を尋ね、他のエージェントが妥当であったら、そのエージェントに縮小させるものである。これは次の文法に従う

A::=c | if c then A |　A,A | new X in A

計算は単純で、情報はストアに与えられるだけである。ask動作は同期のために用いられる。欠点としては、情報が欠如している状態の認識ができないということにある。

5.1.2　デフォルト並行制約プログラミング

デフォルト並行制約プログラミングとは、並行制約プログラミングに否定情報を扱うための機能が追加されたものである構文は以下のようになる。

A::=c | if c then A | A,Anew X in A | if c else A

否定情報は　if c else A という構文からなる negative ask エージェントが扱う。negative ask エージェントが行う negative ask 操作は、計算時に有効であるかどうか決定することができないため、デフォルト並行制約プログラミングでは、negative ask の結果を予測し、計算を行うことになっている。予測が正しければ、実際に等しい出力が得られる。異なった場合は計算を取り消し、場合によりバックトラックなどを行う。

5.1.3　ハイブリッド並行制約プログラミング

ハイブリッド並行制約プログラミングは、ハイブリッドシステムのモデリングと検証のための言語として考案されたものである。ハイブリッドシステムは離散的・連続的な動作からなるシステムで、
↓
例をここに書く。
↓
ハイブリッド並行制約プログラミングは時間軸に沿った表現ができるようにデフォルト並行制約プログラミングを拡張した枠組みとなっている。各時点においてデフォルト並行制約プログラミングの計算が繰り返し行われるようなモデルを持つ。プログラムはある時点での処理についての記述とある時点から次の時点までの間の処理についての記述からなる。
ハイブリッド並行制約プログラミングの構文は以下のようになる。
↓
↓
↓ここに構文を入れる
↓
↓
次にハイブリッド並行制約プログラミング言語による簡単な例を示す。

```
t = 65,
do hence t'=1 until t = 75,
when t=75 do hence t'=0,
sample(t)
```

表 5.1: 温度変化のシミュレーションのプログラム

このプログラムは温度変化をシミュレーションしたものである。初期設定として温度 (t) を 65 とする。このプログラムでは、温度 (t) は時間 1 につき 1 増加す

る。(時間と温度はプログラム上のものなのであえて単位はつけない。) そうしていくと、温度はずっと上昇してしまうので、温度 (t) が 75 となったら、温度を一定に保つというようなものである。

以下に実行結果を示す。左の行がこのプログラム内の時間であり、右の行が温度 (t) の値である。

```
$ hybrid -r 100 ../examples/aircon.hcc
t::
0.000000e+00    6.500000e+01
1.000000e-01    6.510000e+01
6.000000e-01    6.560000e+01
3.100000e+00    6.810000e+01
1.000000e+01    7.500000e+01
1.000000e+01    7.500000e+01
1.010000e+01    7.500000e+01
1.060000e+01    7.500000e+01
1.310000e+01    7.500000e+01
2.560000e+01    7.500000e+01
8.810000e+01    7.500000e+01
1.000000e+02    7.500000e+01
Max time!
```

表 5.2: 温度変化のシミュレーションのプログラムの実行結果

第6章　シンプレックス法(仮)

6.1　線形計画問題

シンプレックス法の説明をする前に、まずは線形計画問題という問題についてふれておく。線形計画問題とは、目的関数とそのすべての制約条件が設計変数 $x^T = (x_1, x_2, \ldots, x_n)^T$ の線形関数で表される問題である。また、すべての設計変数は非負でなければならない。$(x_i \geq 0)$

線形計画問題は、例を挙げると次のようになる。

$$\begin{aligned} x_1 + x_2 &\leq 8 \\ 2x_1 + x_2 &\geq 6 \\ x_1 + 2x_2 &\geq 6 \end{aligned} \tag{6.1}$$

$$x_1, x_2 \geq 0 \tag{6.2}$$

$$f(x_1, x_2) = 3x_1 + 2x_2 \tag{6.3}$$

グラフを用いてこの問題を解くと、図なんちゃらに示すように、最適解は明らかに点Cであって、$(x_1, x_2) = (2, 2)$ の時でそのときの値は $f = 10$ となる。

線形計画問題の特徴をまとめると次のようになる。

- 設計変数はすべて非負である。$(x_i \geq 0, i = 1, \ldots, n)$
- 実行可能領域が凸多面体となる。
- 最適解は実行可能領域の端点に存在して、その解は大域的な最適解である。

6.2　シンプレックス法

設計変数が２個の場合は、図？のように図に描いて最適解を求めることができるが、一般的な３次元以上の場合にはこのような解法は不可能である。そのような場合にシンプレックス法が用いられる。

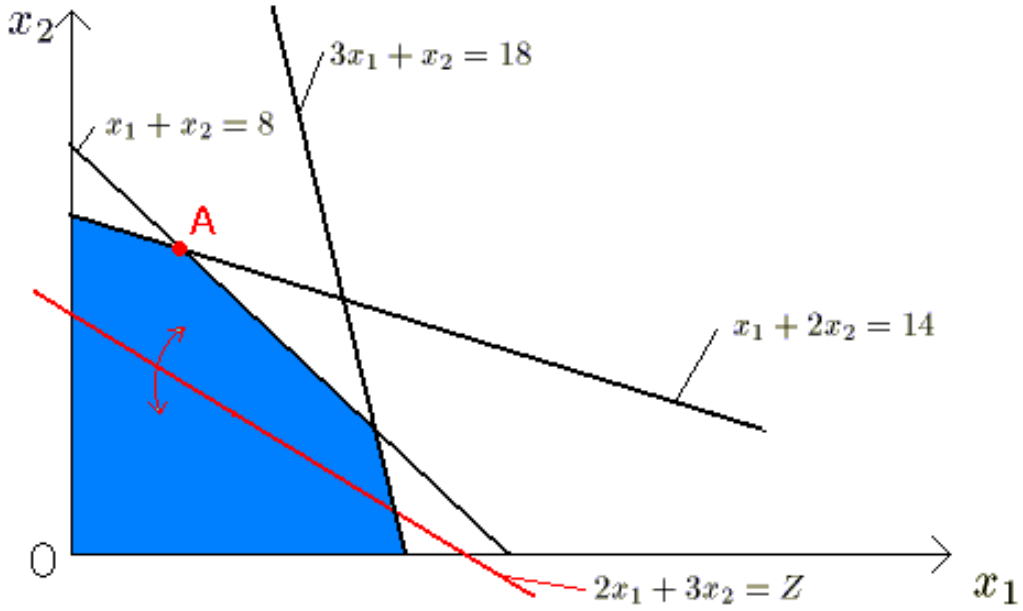

図 6.1: Grifon の制約処理系

6.2.1 特徴

『最適解は必ず端点に存在する』という線型計画問題の特徴を最大限利用した手法である。まず、可能基底解となる端点を求め、その後に、目的関数の値が改善されるような端点に移動する。その後、さらに目的関数が改善される端点へと移動を繰り返し、最終的に目的関数が改善されなくなった端点が見つかった場合は、それを最適解とみなす。

この手法ではすべての短点を調べる必要がなく、一部の端点を求めるだけで最適解に到達できる、非常に効率的な手法であり、現在でも広く使われている。

6.2.2 シンプレックス法の手順

シンプレックス法の手順は以下のようになる。

1. 線型計画問題にすラック変数を導入して、標準形に書き換える。
2. 可能基底解となる端点を出発点として選ぶ。
3. 端点間を移動し、目的関数を改善する点を選ぶ。
4. 最適解となる端点を求める

標準形とスラック変数

線形計画法の標準形とは、すべての制約条件が等式制約として与えられ、目的関数が最小化となるような形式を意味する。しかし、実際の問題の制約条件は式？に示したように、不等式制約で表されることが多いため、スラック変数と呼ばれる非負の変数を導入し、等式制約に置き換える。式？は次式のようになる。

$$\begin{aligned} x_1 + x_2 + x_3 \qquad\quad &= 8 \\ 2x_1 + x_2 \qquad - x_4 \qquad &= 6 \\ x_1 + 2x_2 \qquad\quad - x_5 &= 6 \end{aligned} \tag{6.4}$$

スラック変数として導入された x_3, x_4, x_5 もすべて非負でなければならない。もともとの不等式制約と等価になるようにこれらのスラック変数の前の符号が定まる。

また、目的関数は費用最小化、重量最小化のように『最小化』の形式で表される場合だけでなく、利益最大化、固有振動数最大化のように『最大化』の形式で表される場合もある。線形計画法の標準形は必ず、『最小化』の形式でなければならないので、元の問題が『最大化』の場合、目的関数自身に -1 をかけて符号を変え、『最小化』となるように変換する。こうすると、線形計画問題の一般形は次式で表される。

$$\left.\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m \end{aligned}\right\} \quad (n > m) \tag{6.5}$$

$$x_1, x_2, \ldots, x_n \geq 0 \tag{6.6}$$

$$z = f(x) = c_1x_1 + c_2x_2 + \cdots + c_nx_n \rightarrow \text{最小} \tag{6.7}$$

ここで示すように、制約条件数 m は設計変数の数 n よりも小さい。制約式？および非負の条件？を満たす解（$x_1, x_2, x_3, \ldots$）を実行可能解または可能解と呼ぶ。制約領域はすべて可能解の集合なので、実行可能領域とも呼ばれる。

なお、式 (なんちゃら) のように、設計変数の取り得る範囲を定める制約条件のことを側面制約と呼んで、普通の制約条件と区別して扱うことがある。

6.2.3 出発点となる可能基底解の選び方

標準形に変形できたら、次に出発点となる可能基底解を選ばなければならない。

端点の移動法

１、出発点としての可能基底解

設計変数の数がｎ、制約式が m 個 $(n \succ m)$ の線形計画問題を考える。簡単のために、$(x_1, x_2, x_3, \dots)$ が可能基底解である場合で説明を始める。ｍ個の制約式をベクトル形式で書くと、次式で表される。

$$\begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} x_1 + \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} x_2 + \cdots + \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} x_n = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

つまり、

$$\boldsymbol{a_1} x_1 + \boldsymbol{a_2} x_2 + \cdots + \boldsymbol{a_n} x_n = \boldsymbol{b} \tag{6.8}$$

ここで、$x_1, \dots, x_m$ を基底変数として、$x_{m+1}, x_{m+2}, \dots, x_n$ を非基底変数として、0 とおく。そして、基底変数 $(x_1, \dots, x_m)$ に対する m 元連立方程式を解く。

$$\boldsymbol{a_1} x_1 + \boldsymbol{a_2} x_2 + \cdots + \boldsymbol{a_n} x_n = \boldsymbol{b}$$

つまり、

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} \tag{6.9}$$

これを解いて、$(x_1, x_2, \dots, x_m)$ がすべて正であったと仮定する。このとき $(x_1, x_2, \dots, x_m, 0, 0, \dots, 0)$ は一つの可能基底解である。

２、基底変数の入れ替えー新しい基底変数の選択

次に非基底解の一つ $x_k, (k = m+1, m+2, \dots, n)$ を新しく基底解に入れることを考える。その係数ベクトル$\boldsymbol{a_k}$を基底ベクトルを用いて次式で表す。

$$\boldsymbol{a_k} = \boldsymbol{a_1} {\lambda_1}^k + \boldsymbol{a_2} {\lambda_2}^k + \cdots + \boldsymbol{a_m} {\lambda_2}^m$$

つまり、

$$\begin{pmatrix} a_{k1} \\ a_{k2} \\ \vdots \\ a_{km} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mm} \end{pmatrix} \begin{pmatrix} {\lambda_1}^k \\ {\lambda_2}^k \\ \vdots \\ {\lambda_m}^k \end{pmatrix} \tag{6.10}$$

これを解いて ${\lambda_j}^k, (j = 1, \ldots, m)$ を求める。

３、基底変数の入れ替えーどの変数を基底変数から取り除くか

式 (なんちゃら) を θ 倍したものを、式 (なんちゃら) から引くと次式が得られる。

$$\boldsymbol{a_1}(x_1 - {\lambda_1}^k\theta) + \boldsymbol{a_2}(x_2 - {\lambda_2}^k\theta) + \cdots + a_m(x_m - {\lambda_m}^k\theta) + \boldsymbol{a_k}\theta = \boldsymbol{b} \tag{6.11}$$

この式では未知数は θ のみである。

ここで $(x_j - {\lambda_j}^k\theta), (j = 1, \ldots, m)$ のいずれかが 0 となるまで、θ を大きくすることによって基底変数を入れ替えることができる。

ここで、θ を 0 から徐々に大きくしていき、$\theta = \theta_p$ において、最初に $x_p - {\lambda_p}^k\theta_k = 0$ になるとする。

つまり、

$$\theta_k = \min_j \frac{x_j}{{\lambda_j}^k} = \frac{x_p}{{\lambda_p}^k} \quad ({\lambda_j}^k > 0) \tag{6.12}$$

このとき、元の基底変数から x_p が基底から外れ ($x_p = 0$ となる)、x_k が新たに基底変数となる。その値は $x_k = \theta_k$ である。

４．基底変数を入れ替えるべきかどうか

基底変数を x_p から x_k に入れ替えることで、端点が移動する。それにより、目的関数が改善されるかどうかを調べる。そのために、$0 \leq \theta \leq \theta_k$ に対して、目的関数がどのように変化するのか調べる。

$$\begin{aligned} f(\theta) &= \boldsymbol{c_1}(x_1 - {\lambda_1}^k\theta) + \boldsymbol{c_2}(x_2 - {\lambda_2}^k\theta) + \cdots + \boldsymbol{c_m}(x_m - {\lambda_m}^k\theta) + c_k\theta \\ &= (c_1x_1 + c_2x_2 + \cdots + c_mx_m) + (c_k - c_1{\lambda_1}^k - c_2{\lambda_2}^k - \cdots - c_k{\lambda_m}^k)\theta \end{aligned} \tag{6.13}$$

この第１項はもとの目的関数の値で、第２項が基底変数の入れ替えに伴う目的関数の変化分を表す。したがって、θ の係数が負であれば、この移動によって目的関数の値は改善される。この係数を相対費用係数と呼ぶ。つまり、

$$\bar{c}_k = c_k - c_1{\lambda_1}^k - c_2{\lambda_2}^k - \cdots - c_k{\lambda_m}^k \tag{6.14}$$

すべての非基底変数 ($n - m$ 個) に対して相対費用係数を計算し、それが負の値をもち絶対値が最大となるような x k を新しい基底変数に選べばよい。

５．収束判定

上記の手順を繰り返すことで、規定変数がどんどん代わっていき、端点が移動することになる。最後に、相対費用係数 $\bar{c}_k$ がすべて正である場合、もはやどの基底変数の入れ替えを行っても目的関数は改善されない。つまり、そのときの端点が最適解である。

第7章　Cassowaryにおける幾何制約の実装(仮)

この章ではCassowaryで実装されている幾何制約をハイブリッド並行制約プログラミングで実装する。Cassowaryで実装されている幾何制約は大きく分けて２種類あり、１つは幾何学的オブジェクトを生成する制約であり、もう１つはオブジェクト、またはオブジェクト同士の幾何学的関係の制約である。本章では、まず幾何学的オブジェクトを生成するために必要なPointの概念を表現するクラスの設計と実装、次にそのPoint の概念を利用して、Line、Midpoint Line、Rectangle、Ovalの概念を表現するクラスの設計と実装、最後に幾何学的関係を表現する制約の設計と実装について説明する。

7.1　Hccで幾何制約を実装する上での問題点

7.2　点オブジェクトの表現

7.2.1　設計

幾何学的図形のオブジェクトをつくるためには、ある点を表現するようなクラスがあると、表現しやすかったのでまず、最初にPointClassという点を表現するクラスを実装した。このPointClassで表現する点は２次元上のものとして考える。このPointClassは２つの引数をとってそれを保持し続けるように設計した。ただし、Cassowaryで実装されているものでは、点が動くという振る舞いをするので、もし、点を移動させたら、その点はその移動させた場所の座標を保持するようにした。その点の移動を表現するJAVAでいうメソッド的なもの(ハイブリッド並行制約プログラミングではこのことをclosureと呼ぶ) であるPointClassの中に実装した。

7.2.2　実装

以下がPointClassのプログラムである。

```
PointClass = (initx,inity)[x,y,MovePoint, Change]{

                x = initx, y = inity,

        always {
                MovePoint = (newx,newy){Self.Change,x=newx,y=newy},
                lcont(x),lcont(y),
                unless Change then {cont(x),cont(y),x'=0,y'=0}
                }
      },
```

表 7.1: PointClass のプログラム

7.3 幾何学的オブジェクトの表現

7.3.1 Line オブジェクト

設計

実装

以下が LineClass のプログラムである。

```
LineClass = (X,Y)[x1,y1,x2,y2]{
        x1 = X.x, y1 = X.y,
        x2 = Y.x,y2 = Y.y,
        always{
                lcont(x1),lcont(y1),lcont(x2),lcont(y2),
                unless (X.Change) then {cont(x1),cont(y1),x1'=0,y1'=0},
                unless (Y.Change) then {cont(x2),cont(y2),x2'=0,y2'=0},
                if(X.Change) then {x1 = X.x, y1 = X.y},
                if(Y.Change) then {x2 = Y.x, y2 = Y.y}
        }
  },
```

表 7.2: LineClass のプログラム

7.3.2 Midpoint Lineオブジェクト

設計

実装

以下がMidPointLineClassのプログラムである。

```
MidPointLineClass = (X,Y)[mx1,my1,mx2,my2,midx,midy]{
        mx1 = X.x, my1 = X.y,
        mx2 = Y.x,my2 = Y.y,
        midx = (X.x + Y.x)/2,
        midy = (X.y + Y.y)/2,
        always{
                lcont(mx1),lcont(my1),lcont(mx2),lcont(my2),
                lcont(midx),lcont(midy),
                unless (X.Change) then {cont(mx1),cont(my1),mx1'=0,my1'=0},
                unless (Y.Change) then {cont(mx2),cont(my2),mx2'=0,my2'=0},
                unless (X.Change || Y.Change)then
                         {cont(midx),cont(midy),midx'=0,midy'=0},
                if(X.Change) then {mx1 = X.x, my1 = X.y},
                if(Y.Change) then {mx2 = Y.x, my2 = Y.y},
                if (X.Change || Y.Change) then
                         {midx = (X.x + Y.x)/2,midy = (X.y + Y.y)/2}
            }
 },
```

表 7.3: MidpointLineClassのプログラム

7.3.3 Rectangleオブジェクト

設計

実装

以下がInitRectangleClassのプログラムである。以下がRectangleClassのプログラムである。

```
InitRectangleClass = (X,Y)[rx1,ry1,rx2,ry2,rx3,ry3,rx4,ry4]{
                rx1 = X.x, ry1 = X.y,rx2 = X.x,ry2 = Y.y,
                rx3 = Y.x, ry3 = Y.y,rx4 = Y.x,ry4 = X.y
            },
```

表 7.4: InitRectangleClass のプログラム

7.3.4 Oval オブジェクト

設計

実装

以下が OvalClass のプログラムである。

7.4 幾何学的関係の表現

7.4.1 設計

7.4.2 実装

```
RectangleClass = (W,X,Y,Z)[rx1,ry1,rx2,ry2,rx3,ry3,rx4,ry4]{
        rx1 = W.x,ry1 = W.y,rx2 = X.x,ry2 = X.y,
        rx3 = Y.x,ry3 = Y.y,rx4 = Z.x, ry4 = Z.y,
        always{
                lcont(rx1),lcont(ry1),lcont(rx2),lcont(ry2),
                lcont(rx3),lcont(ry3),lcont(rx4),lcont(ry4),
                if (W.Change) then
                {rx1 = W.x, ry1 = W.y, rx2 = W.x, ry4= W.y,
                 cont(rx3), cont(ry3),cont(rx4), cont(ry2),
                 rx3'=0, ry3'=0,  rx4'=0, ry2'=0},
                if (X.Change) then
                {rx2 = X.x, ry2 = X.y, rx1 = X.x, ry3= X.y,
                 cont(rx4), cont(ry4),cont(rx3), cont(ry1),
                 rx4'=0, ry4'=0,  rx3'=0, ry1'=0},
                if (Y.Change) then
                {rx3 = Y.x, ry3 = Y.y, rx4 = Y.x, ry2= Y.y,
                 cont(rx1), cont(ry1),cont(rx2), cont(ry4),
                 rx1'=0, ry1'=0,  rx2'=0, ry4'=0},
                if (Z.Change) then
                {rx4 = Z.x, ry4 = Z.y, rx3 = Z.x, ry1= Z.y,
                 cont(rx2), cont(ry2),cont(rx1), cont(ry3),
                 rx2'=0, ry2'=0,  rx1'=0, ry3'=0},

                unless (W.Change || X.Change || Y.Change ||Z.Change)then
                {cont(rx1),cont(ry1),rx1'=0,ry1'=0,
                 cont(rx2),cont(ry2),rx2'=0,ry2'=0,
                 cont(rx3),cont(ry3),rx3'=0,ry3'=0,
                 cont(rx4),cont(ry4),rx4'=0,ry4'=0
                                }
                }
      },
```

表 7.5: RectangleClass のプログラム

```
OvalClass = (X,Y)[ox1,oy1,ox2,oy2]{
        ox1 = X.x,oy1 = X.y,
        ox2 = Y.x,oy2 = Y.y,
        always{
                lcont(ox1),lcont(oy1),lcont(ox2),lcont(oy2),
                unless (X.Change) then
                {cont(ox1),cont(oy1),ox1'=0,oy1'=0},
                unless (Y.Change) then
                {cont(ox2),cont(oy2),ox2'=0,oy2'=0},
                if(X.Change) then {ox1 = X.x, oy1 = X.y},
                if(Y.Change) then {ox2 = Y.x, oy2 = Y.y}
           }
    },
```

表 7.6: OvalClass のプログラム

第8章　これまでのまとめと今後の課題(仮)

8.1　幾何制約における比較

8.1.1　Cassowary で実装されている幾何制約

- 制約に強さという概念があるため、強い制約が満たされない場合でも最適解が求められる。
- 制約 (オブジェクト、関係) が実装されたものしか使えないので、アニメーションが制限される。

8.1.2　ハイブリッド並行制約プログラミングで実装した幾何制約

- プログラムによって幾何制約を作っていけばよいので、いろいろな制約が実現可能でアニメーションの幅が広がる。
- プログラミング自体が難しいのが難点である。

8.2　今後の課題と目標

謝辞

参考文献

[1] 石井大輔:制約に基づくアニメーション作成システム Grifon の設計と実装,
早稲田大学大学院理工学研究科情報科学専攻修士論文 (2003)

[2] 若槻聡一郎:制約に基づくアニメーション作成システム Grifon におけるデータ　構造の設計と実装,
早稲田大学理工学部情報学科卒業論文 (2003)

[3] Borning A., Marriott K., Stuckey P., and Xiao Y.: Solving Linear Arithmetic Constraints for User Interface Applications,
In *Proceedings of the 1997 ACM Symposium on UIST*, 1997, pp. 87–96.

[4] Vineet Gupta.,Radha Jagadeesan.,Vijay Saraswat.,Daniel G Bobrow.: Programming in hybrid constraint languages,
1997.

[5] Vineet Gupta.,Radha Jagadeesan.,Vijay Saraswat.,Daniel G Bobrow.: Computing with continuous change,
Technical report,Xerox Palo Alto Research Center,May 1995.

[6] 細部博史:モジュール機構を備えた幾何制約解消系,
WISS 2000

[7] Scalable Vector Graphics1.1 Specification-Japanese translation.
`http://www.hcn.zaq.ne.jp/`$_{/REC-SVG11-20030114/}$$GuardedHornClauses(GHC)$、および並行論理プログラミング (

[8] 細部博史:ユーザーインターフェースにおける制約解消法の研究動向,
`http://www.ueda.info.waseda.ac.jp/ ueda/ghc.html`

[9] SICStus Prolog Homepage
http://www.sics.se/isl/sicstus.html

[10] 平川正人・安村通晃編:bit 別冊 ビジュアルインターフェース共立出版 (1996)

[11] Juno-2 Homepage
http://www.research.digital.com/SRC/juno-2/juno-2-home.html

[12] Erich Gamma : “Design Pattern”, Addison-Wesley Publishing Company (1995). オブジェクト指向における再利用のためのデザインパターン, 本位田 真一, 吉田 和樹 監訳 (ソフトバンク, 1995)