学　士　論　文

題　目

イベント駆動型制御とその表現
—動的環境下における多自由度ロボットの
運動学習を目指して—

指導教員　松山 隆司 教授

京都大学工学部 電気電子工学科

氏　名　山口 明彦

平成 18 年 2 月 9 日

目 次

第 1 章　序論 **1**
1.1　背景と目的 1
1.2　動的環境下でのロボットの運動学習における課題 1
1.2.1　運動学習の必要性 2
1.2.2　運動学習における課題 2
1.2.3　探索空間の構成と制御則 2
1.3　提案手法の概要 3
1.3.1　制御則とその表現 3
1.3.2　学習手法 4

第 2 章　イベント駆動型制御 **5**
2.1　イベント駆動型制御の定義 5
2.2　プリミティブの出力タイミングの決定 5
2.3　プリミティブの形状 6
2.4　イベント駆動型制御における入出力要素 7

第 3 章　EventGraph: イベント駆動型制御の表現 **8**
3.1　依存関係と時間関係の表現 8
3.1.1　依存関係及びイベント間の関係の表現 8
3.1.2　時間的な位置関係の表現 8
3.1.3　時間的な距離の表現 9
3.1.4　メトリックの表現 9
3.2　プリミティブの表現 10
3.3　EventGraph の定義 10

第 4 章　EventGraph に基づく制御アルゴリズム **12**
4.1　制御アルゴリズムの基本要素 12
4.1.1　プリミティブの出力 12
4.1.2　ポイントの監視 12
4.1.3　イベントの予測 13
4.1.4　ポイント生起時刻の保存と更新 13
4.2　EventGraph に基づく制御アルゴリズム 13
4.3　制御アルゴリズムの効率化 14

第 5 章　イベント駆動型制御に基づく運動学習とその評価 **15**
5.1　目的と方針 15
5.2　EventGraph の構造を固定したパラメタ最適化の手法 15
5.2.1　最適化問題の定式化 15
5.2.2　最適化手法 16
5.3　EventGraph の構造を固定した最適化の評価実験 16
5.3.1　状況設定 16
5.3.2　多自由度ロボットの構成 17
5.3.3　EventGraph の構造 17
5.3.4　評価関数 17
5.3.5　実験結果と考察 18
5.4　EventGraph の構造最適化の基礎的検討 18

第 6 章　結論 **19**
6.1　イベント駆動型制御の適性に関する議論 19
6.2　結論と課題 19

謝 辞 **21**

参 考 文 献 **22**

付 録 A　メトリックの拡張とイベントの予測 **24**
A.1　メトリックの学習と意義 24
A.2　メトリックの表現と予測 24

付 録 B　EventGraph の制約 **26**
B.1　生成可能にするための制約 26
B.2　計算量軽減のための制約 26

付 録 C　アルゴリズムの詳細 **28**
C.1　メトリックとプリミティブパラメタの推定値 28
C.2　サブアルゴリズム 28
C.2.1　導入 28
C.2.2　ポイント 29
C.2.3　エッジ 30
C.2.4　イベントノード 31
C.2.5　制御ノード 32
C.3　メインアルゴリズム 34

付 録 D　EventGraph の構造を固定した最適化 **36**
D.1　EventGraph に対する演算 36
D.2　SPX と MGG 36

付 録 E シミュレーションの詳細 **38**
E.1 多自由度ロボットの構成の詳細 38
E.2 順動力学計算アルゴリズム 38
E.3 トランポリンのモデル 40
E.3.1 トランポリンの順動力学 40
E.3.2 トランポリンの反作用モデル 41

Abstract

Motion acquisition of multi-degree-of-freedom (multi-DOF) robot in dynamic environment, such as sandy soil and trampoline, is very difficult since the state of environment is changed by the robot motion. There are some difficulty in a control law based on differential equation: (1) large DOF, (2) discrete event dynamics, (3) underactuated system, (4) error in modeling of environment. For such problem, motion acquisition by robot itself, namely, motion learning is one of efficient approach since the acquired motion may be stable in the environment and such learning requires little cost.

In this study, we attempt to obtain an efficient method of motion learning for multi-DOF robot in dynamic environment. Value function is given for the evaluation of the motion that the robot learned, so the learning motion is optimizing the control law. Because of the difficulty of obtaining differential information between control law and value function, the learning is the searching, and how to construct the search space is most important in the searching problem; we have to construct it as small as we can. The existence of a solution to realize the motion depends on the control law, therefore, the construction of search space is closely related with the control law; but there are few research about motion learning focused on this point.

We designed a control law based on the finding about "invariant features" of motion (Kuniyoshi, et al.[1)]); energy input timing is important for success of the motion. This control law is similar to human qualitative knowledge about motion, that is, a energy input procedure that involves *event*; for example, 'bend legs before landing event.' Therefore, we named the control law as *event-driven control* that can be described in a little parameters, and we also designed a graph structure as its representation that describes the dependence and temporal relation among event and local control signal, and is named EventGraph.

Motion learning based on event-driven control consists of (a) optimizing the parameters in a fixed graph structure, (b) optimizing the graph structure. In this paper, we focused on (a); from simulation, we found that the motion learning based on event-driven control is efficient for multi-DOF robot in dynamic environment. In this simulation, the structure of EventGraph is given from the qualitative knowledge of human, hence it may be optimal to the robot that has a similar embodiment to human. Furthermore, we took a primary study about (b); as the result, a better graph structure is found for the robot that has different embodiment from human.

Realization of optimizing that includes graph structure may enable multi-DOF robot to learn motion in unfamiliar dynamic-environment, *learn by watching*[2)], and create new motion. The method suggested in this paper is a fundamental technology for them.

第1章 序論

1.1 背景と目的

近年ヒューマノイド型をはじめとする多自由度ロボットが開発され，2足歩行など，かなり高度な運動が実現されている．多自由度ロボットの利点はその汎用性の高さにあり，ひとつのロボットにさまざまな行動を取らせることできる．その半面，自由度の高さのために，制御の不安定さや困難さを生んでいる．

従来，多自由度ロボット—ここでは，環境に固定されていない複数のリンクから成るロボットを想定している—の研究は床や地面の上といった静的な環境のもとで行われることが多かったが，現実の環境は動的であり，ロボットの行動によって環境の状態（例えば砂地やトランポリンのような環境を想定）やロボットの身体と環境の関わり方（接触状態など）が変わる．すなわち身体の力学的な拘束条件が変化する．さらに，ロボットの身体性（形状，リンクの質量など）は，設計者が存在するため事前に知ることができるのに対し，環境の状態や特性（例えば床の摩擦係数，床反力など）は必ずしも事前に知ることができるとは限らない．例えば惑星にロボットを送り込むような状況である．

よって，多自由度ロボットの個々の運動（歩行，跳躍など）に対して制御則を見出すコストはかなり大きく，また，個々のロボットにさまざまな環境下で運動を実現できるようチューニングすることも同様に困難である．つまりロボットの汎用性を活かせない．

そこで，ロボット自身に運動を学習させることによって，これらの困難さの解決を試みる．ロボット自身が制御則を環境に合わせて最適化するため，その環境に対して安定であり，人手によるチューニングコストが少ないからである．もちろんロボットが学習すると言っても，何の知識も無しに運動を実現することは困難であるから，運動に関する何らかの知識を与えることは必要である．このような知識として，例えばロボットが実現した軌道（以下「実現軌道」）に対する評価関数で与えることが考えられる [3)4)]．

本研究で目指すところは，実現軌道に対する評価関数が与えられたとき，動的な環境下に置かれた多自由度ロボットに評価関数を最大にするような運動を学習させること，言い替えると制御信号を得る方法を見出すことである．評価関数としては，与えられた参照軌道との誤差を用いる方法もあるが，参照軌道がロボットにとって最適である保証はない．そこで本研究では，「跳躍では高さが高いほど評価がよい」といった運動の目的や特徴を表現するような評価関数が与えられることを前提とする．

次節以降で論じるように，このような学習は探索空間の爆発という困難さを含み，制御則と探索空間の両方の観点から設計して行かなければならない．そこで本研究では「運動の制御則とその表現としてふさわしいものは何か」ということに焦点を絞る．

1.2 動的環境下でのロボットの運動学習における課題

本節では，動的環境下での多自由度ロボットの制御における困難さについて述べ，続いて運動学習における課題を考察する．

1.2.1　運動学習の必要性

まず，解析的な制御（系を微分方程式によって記述し，それに基づいて制御信号を計算するような方法）の困難さについて述べ，運動学習の必要性について論じる．

制御信号と実現軌道の関係を Fig.1.1 に示す. 制御信号は多自由度ロボットの身体 (body) に送られ（具体的には関節のアクチュエータに入力され），環境と相互作用し，軌道が実現される．この順方向（制御信号 → 実現軌道）の関係は一般に微分方程式で記述可能であるが，微分方程式に基づいた制御はこの逆問題であり，このときの困難さとして，(1) ロボットの自由度が大きい，(2) 身体の力学的拘束条件が変化（ロボットと環境の接触状態やロボットのリンク間での接触状態，あるいは可動範囲が限定されている関節角の変化によって生じ，微分方程式が不連続に変化する），(3) 劣駆動系（ロボットの自由度よりも系の一般化座標*の次元数が大きい[5)6)]），(4) 環境のモデル化誤差（床や地面の反作用モデルを正確に作ることは難しい[1)]）などが考えられる．

微分方程式に基づいた制御を実現するために，歩行などにおいては，上半身のバランスなどを実現軌道の制約として加えて，多自由度ロボットがもつ冗長性を軽減している[7)]．その半面，制約を常に満たすように制御しながら歩行するのでエネルギロスが大きく，ヒトのような無駄のない動きを獲得するために学習を使う研究がなされている[3)]．

以上より，動的環境下で軌道の評価関数を最大にするような多自由度ロボットの制御信号を求めることは困難な問題であり，ロボットの運動学習というアプローチが手法のひとつとして有効であると結論される†.

1.2.2　運動学習における課題

ここで想定している運動学習とは，実現軌道に対する評価関数が与えられたときに，それを達成するための制御則を学習することである.「制御則を学習する」とは，ある時刻，身体及び環境状態において，評価関数を最大にするような制御信号を出力するためのルールの獲得である‡.

よってこの場合の学習とは制御則の最適化であるが，前項で述べたような解析的な制御が困難な状況下では，制御則と評価関数の関係から微分情報を求めることは難しく，山登り法のような最適化手法は使えない．故に学習とは探索であり，探索においてもっとも重要となる点は,「いかにコンパクトに探索空間を構成するか」である．

1.2.3　探索空間の構成と制御則

運動学習において探索空間を構成するとき,「理論的に運動を実現する解が存在すること」が最大の要請であり，探索の成功を左右する．運動を実現できるか否かは制御則に依存していることを考えると，探索空間の構成は制御則と密接に結び付いていることがわかる．よって運動学習を考える上では，制御則を考慮しつつ探索空間を構成して行く必要がある．

この観点から考えて，探索手法の選択（遺伝アルゴリズム[4)]，強化学習[9)] など）よりも，

---

* 系を記述するために必要な最小の数の座標．動的環境と多自由度ロボットから成る系の一般化座標の次元数は，ロボットの自由度（=アクチュエータの数）に加えて，ロボットの位置と姿勢を記述するために 6 次元，さらに動的な環境を記述するために何次元か必要である．

† 解析的な制御からアプローチする例としては，Ishikawa らのように，ハイブリッドシステムの枠組で動的環境下における制御を実現しようとする研究もある[8)]．

‡ 評価関数が各時刻ごとに与えられるわけではないことに注意されたい．

探索空間の構成を第一に議論すべきである．

探索空間の構成において，目標軌道を探索するか，制御信号側を探索するか，2 通りの選択が考えられる．目標軌道の探索とは実現軌道の目標系列を獲得することであり，そこから逆運動学問題を解いて制御信号を計算し，ロボットに適用すると言う手法である．

しかし動的環境下では軌道側から制御信号を求めることは一般的に困難であり，何より，実現される軌道には身体と環境がつくり出す複雑さが含まれている．よって，情報量（ここでは運動の記述に必要な空間の次元数）としては軌道側の方が多く，コンパクトな探索空間を構成する上では，より情報量が少ない制御信号側が適していると言える．

例えば，目標軌道空間をグリッドで分割（時間軸方向に等間隔で分割）して運動学習を実現している研究 [3] がある．しかしグリッドによる探索空間の分割は，分割の度合を粗くすると運動を実現できる解が存在しなくなる可能性が出てくる一方で，細かくすると学習に時間がかかるという問題がある．

以上より，従来の研究ではコンパクトな探索空間の構成と制御則の選択に関して十分な議論がなされておらず，このため実用的な運動学習手法の獲得には至っていないと考えられる．

1.3 提案手法の概要

前節の考察に基づき，運動学習のための制御則とその表現方法の概要を述べ，それに基づく探索手法について考察する．

1.3.1 制御則とその表現

コンパクトな探索空間の構成と制御則の選択に関して，Kuniyoshi らの “invariant features” に関する研究 [1] から得られた知見をもとにした制御則を提案する．この知見とは

K1 身体の力学的拘束条件を利用すれば，常に精密な制御をしなくても，ツボ (knacks) を押えた制御で大域的にはタスク（運動）が実現できる

K2 タスクを成功させるために重要なのは，トータルエネルギや関節角ではなく，局所的なエネルギを出力するタイミングである

というものであり，K1 から，運動が連続信号（実現軌道や制御信号）よりもはるかに少ない情報量で表現可能であることが示唆される．そしてそれは，運動の実現可能性を保持したまま探索空間の縮小につながり，運動学習におけるコンパクトな探索空間の構成という目的にも合致する．

身体の力学的拘束条件とは，例えば地面から受ける反力や関節角の制約などのことで，ツボを押えた制御とは，要所要所（ツボ）で適切に力を加え，ほかの（時間）領域では慣性に身を任せて積極的に制御しないような制御法のことである．類似の考え方として，「人間の歩行の ballistic モデル」と言った表現をしている研究もある [10]．“ballistic” とは「弾道 (学) の」という意味で，弾丸のように最初だけ制御し，後は積極的に制御しないようなモデルである．このような受動運動を規範にしている制御形式 [11] ではエネルギの消費量の少なさなどが注目されているが，本研究で着目するのは「情報量の少なさ」である．

そこで本研究では，K1, K2 の知見に基づいた制御則の実現として，人間の定性的な運動知識，すなわち身体の各部位に力を加えて行く手順に類似した制御則を設計する．その概

要は，ある身体の部位に対する局所的な制御信号—プリミティブ—を出力するタイミングを，ほかのプリミティブや，観測軌道から安定して抽出される点—イベント—との関係から決定して出力するというものであり,「イベント駆動型制御」と名付けた．また，イベント駆動型制御の表現として，イベントとプリミティブ，プリミティブとプリミティブの依存関係及び時間関係を記述したグラフ構造—EventGraph—を設計した．

1.3.2 学習手法

イベント駆動型制御の表現である EventGraph の学習は,

1. グラフの構造を固定したパラメタ（プリミティブの形状など）の最適化
2. グラフの構造自体の最適化

の 2 段階に分けられる（Fig.1.2 参照）.

一度最適化されたグラフの構造は環境や身体の状態の変動に対して大きく変化しないため，異なった環境下でもグラフ構造を固定した最適化のみ行えばよく，効率的であると考えられる．また，身体性が同じロボットはもとより，身体性が異なるロボット（ただしリンクの構造は同じという想定）に対しても，あるロボットが学習した制御表現 (EventGraph) を初期値にして探索することで効率的な学習が行えると考えられる．

本論文では，上記 1 のグラフの構造を固定した学習を中心に議論を深め，2 の，グラフの構造自体の最適化に関しては基礎的な検討を行う．なお，グラフの構造については人間の定性的な運動知識に基づいて与えており，人間と類似した身体性をもつロボットに対しては，ある程度最適であると考えられる．

グラフの構造を含めた最適化を実現することで，未知の（学習していない）動的環境への適応や，人間とは異なる身体性をもつロボットが運動学習をすることにつながるだろうと期待される．また，グラフの構造はある程度運動の「型」を決めるため，グラフの構造の最適化を含めた運動学習は，模倣学習 [2]（他者の運動の観測軌道から主体の身体に合わせた制御則を獲得する）や運動創発（既に学習した運動から新たな運動を創り出す）などに発展して行くと考えられる．本研究で提案している手法は，そのための基盤となる技術である．

イベント駆動型制御の詳細は 2 章で，EventGraph の詳細は 3 章でそれぞれ述べる．また，4 章では EventGraph に基づいた運動生成アルゴリズムを設計し，5 章ではシミュレータを用いた評価実験を行う．

第2章 イベント駆動型制御

2.1 イベント駆動型制御の定義

本章では，イベント駆動型制御，すなわち，人間の定性的な運動知識（身体の各部位に力を加えて行く手順）に類似した制御則について述べる．

「人間の定性的な運動知識」について,「跳び箱運動（開脚跳び）」を例に取って説明する．開脚跳びにおける身体制御は，おおよそ次のような手順である．

> 踏切板まで助走し，踏切板を蹴って跳ぶ．跳び箱で手を着いて，手で体を前方に加速させると同時に足を開く．体が跳び箱の真上に来たら跳び箱を手で押して前方に跳び，空中でバランスを整えつつ着地する．

この例からわかることは，制御（体の各部位に力を加えること）の手順は,〔踏切板に到達する〕や〔跳び箱に手が着く〕と言った「イベント（運動の実現軌道上の点）」と関係していることである．この関係とは，イベントよりも前に制御が位置したり，イベントと同期，あるいはイベントより後に制御が位置するといったものである．また,「手で体を前方に加速させると同時に足を開く」など，体の各部位に対する局所的な制御の間でも同期や遅延の関係がある．

このような手順に類似した制御則—すなわちイベント駆動型制御—とは，ある身体の部位に対する局所的な制御信号を出力するタイミングを，ほかの局所的な制御信号やイベントとの関係から決定して出力するという制御方法である．この局所的な制御信号のことを「プリミティブ」と呼ぶ．イベント駆動型制御は，Kuniyoshi らの研究から得られた知見—運動を実現するために重要なのは局所的なエネルギを出力するタイミングである—を繁栄させたものになっている．

以下では，Kuniyoshi らの知見をもとに，プリミティブを出力するタイミング及びプリミティブの形状について議論し，最後にイベント駆動型制御の入出力要素について述べる．

2.2 プリミティブの出力タイミングの決定

あるプリミティブが出力されるべきタイミングは，当然環境や身体の状態によって変わる．しかし，前節で述べた制御の「手順」は変わらないと考えられ，したがってプリミティブやイベントの間に，何らかの，環境や身体状態の変動に依らない運動の構造が存在すると考えられる*．以下，プリミティブの出力タイミング決定の指針を定めるために，この構造を中心に議論する．

前節で述べた跳び箱運動の例からわかるように，あるプリミティブを出力するタイミングに関係しているものには，ほかのプリミティブとイベントがある．ここで,「ほかのプリミティブ」には，同じ関節に対するほかのプリミティブだけでなく，異なる関節に対するプリミティブも含まれている．すべてのリンクが連結している以上，関節が異なっていてもそこに何らかの因果関係が存在すると考えられるからである．

---

* もちろん，環境や身体の状態が大幅に変化したらこの構造も変える必要があると思われるが，ここでは変わらないことを仮定して議論する．

イベントとは,〔踏切板に到達する〕や〔跳び箱に手が着く〕と言った，運動の実現軌道上の点（瞬間[†]）である．これらのイベントは，床から踏切板，あるいは空中から跳び箱に身体の一部が移行する瞬間であるから，その物理的意味は「力学的拘束条件の変化」である．

よって，このようなイベントを基準にして制御をするタイミングを決める目的は，身体の力学的拘束を安定に利用するためであると考えられる．この目的を「プリミティブを出力するタイミングの安定化」に拡張し，これに合わせてイベントの定義を「ある運動の実現軌道上から，安定して検出される点（瞬間）」とする．

以上で「環境や身体状態の変動に依らない運動の構造」の要素，すなわち全関節に対するプリミティブ及びイベントについて述べた．次に，これら要素間の関係について論じる．

前述した「手順」は，要素（プリミティブやイベント）の間の時間的な位置関係（前・後・同期）を定めるものである．この関係には時間的な距離が含まれていないが，実際に制御する場合には時間的な距離が必要になる．

時間的な距離は，基本的に，環境や身体の状態によって変動する．時間的な位置関係は,「踏切板に到達する前に助走する」といった変動しない（変動してはならない）関係もあれば,「空中でバランスを整える」ときの各関節に加える制御信号のように変動する（変動しても問題ない）関係もある．

プリミティブやイベントと言った要素間に時間関係（位置関係+距離）が与えられたとき，その関係を使ってプリミティブの出力タイミングを決めるわけだから，両者の間にはどちらを基準にするか，決められている必要がある．つまり，時間関係に伴って依存関係が存在する．

プリミティブはイベントに依存しており[‡]，この依存関係は環境や身体の状態によって変動しない．これに対して，プリミティブとプリミティブの依存関係は，一般的に変動し得る．しかし，あるプリミティブ P_1 がほかのプリミティブ P_2 ただひとつに依存しているような場合を考えると，その依存関係は変動しない（変動すると，P_1 が基準を失い，出力するタイミングを決められない，つまり変動させてはならない）．逆に，依存関係が変動する場合は，どちらを基準にしても他方のタイミングを決められるということだから，変動させなくても問題はないと考えられる．このようなことから，プリミティブとプリミティブの依存関係も変動しないものとして規定する．つまり要素間の依存関係は不変である．

以上をまとめると，次のようになる．運動の要素，すなわち全関節に対するプリミティブやイベントは，環境や身体状態の変動に依らず，依存関係によって構造化されている．また，一部の要素間では時間的な位置関係が環境や身体の状態に依らずに不変で，そのほかの要素間の時間的な位置関係，あるいは時間的な距離関係は環境や身体の状態によって変化する．このような関係をもとに，プリミティブの出力タイミングを決定する．

2.3 プリミティブの形状

Kuniyoshi らの研究から得た知見より，運動タスクを実現する上では,（局所的な）制御信号の波形よりもむしろそのエネルギが重要なファクタとなっており，細かな形状を定める必要はないと考えられる．また，エネルギ投入のタイミングが重要であるから，信号の

[†] 物理的にはおそらく連続的に変化していると考えられるが，この場合，瞬間として捉えても問題はないと思われる（変化が急峻で，その区間幅が十分に短いから）．

[‡] この逆は考えない．6 章を参照

ピークを明示できることが望ましい．これらのことから，トルクの時間積分の大きさを明示でき，かつ波形のピークを明示できるシンプルな制御信号の波形として，本研究では，Fig.2.1 に示すような釣鐘型のトルク波形を用いることにする．この波形を特徴付けるパラメタは，後方向の分散 Δt_{b}，前方向の分散 Δt_{f}，及びピーク $peak$ である．

このようにプリミティブを定めたとき，プリミティブの数が増減するとエネルギに大きく影響を与える．よって，Kuniyoshi らの知見から，プリミティブの数を固定しても運動の達成において問題にならないと思われる．

2.4 イベント駆動型制御における入出力要素

多自由度ロボットと環境から構成されるシステム，及び構成要素間の入出力の関係を Fig.2.2, 2.3 に示す[§]．個々の構成要素について，以下で詳細を述べる．

観測状態: 入力とは観測状態のことである．観測状態は，瞬時的な物理量（例えば座標，速度，関節角など）を指し，多自由度ロボットの状態と，環境状態の両方を表す．すなわち観測状態 Q は，Fig.2.2 において

$$\mathsf{Q} = \{p_1,\, p_2,\, p_3,\, \ldots,\, q_1,\, q_2,\, \ldots,\, \dot{p}_1,\, \dot{p}_2,\, \dot{p}_3,\, \ldots,\, \dot{q}_1,\, \dot{q}_2,\, \ldots\} \tag{2.1}$$

を表す．

観測状態空間: Q が取り得るすべての値を元とする空間であり，$\mathbb{Q}$ で表す，

状態変化の軌跡: $\mathbf{Q}(t) : t(\in \mathbb{R}) \to \mathbb{Q}$ で表す．

イベント条件とその集合: イベント駆動型制御において，実現軌道は入力 $\mathbf{Q}(t)$ としてフィードバックされ，そこからイベントが，発生したかしていないかの真理値として抽出される．そこで，イベントが発生したかどうかを表すために,「イベント条件」を $\mathbb{Q}$ から $\{true, false\}$ への写像として定義する．また，イベント条件の集合を $\mathsf{EventCond}$ として定義しておく．

出力ポートと入力接続: 出力は，最終的には多自由度ロボットの制御入力（具体的にはアクチュエータに送られるトルク信号）$u_1, u_2, \ldots$ に変換されるが，いくつかのプリミティブの重複（時間軸上での重複）を許すために「出力ポート」$\mathbb{O}_1, \mathbb{O}_2, \ldots$ を設け，それらの入力（例えば Fig.2.3 の $\mathbb{O}_1$ に対する in_1, in_2）のひとつに対して出力する，という設計にする．この入力のひとつを「入力接続」と呼ぶ．このとき，出力ポートの集合を，次のように定義する．

$$\mathsf{OutputPortSet} = \left\{ OutputPort_k \mid k \subset \mathbb{N} \right\} \tag{2.2}$$

$$\text{where} \quad OutputPort_k = \left\{ \{in\},\, out \mid \{in\} \in (\mathbb{O}_k)^{N_{\mathrm{O}}^{(k)}},\, out \in \mathbb{O}_k,\, N_{\mathrm{O}}^{(k)} \in \mathbb{N} \right\} \tag{2.3}$$

ここで $\{in\}$ は入力接続の集合，out は出力である．$\mathbb{O}_k$ は出力空間で，例えば出力先が関節のトルクなら $\mathbb{R}$ となる[¶]．

[§] なお，Fig.2.3 における “memory”，“predict” は，多自由度ロボットの記憶装置や，予測装置（例えば観測した状態に基づき，次に何が起こるか予測する）を表しており，以下で述べる観測状態に含まれる．つまりイベント駆動型制御では，これらの装置からの出力も「入力」として扱う．

[¶] 方向は考慮していない（関節の向きで決定される）．また実際にはトルクは実数空間 $\mathbb{R}$ 全体を値域としないので，トルクの空間は $\mathbb{T} \subset \mathbb{R}$ である．

第3章 EventGraph: イベント駆動型制御の表現

2章の議論をもとに，本章ではイベント駆動型制御の表現方法について述べる．

3.1 依存関係と時間関係の表現

3.1.1 依存関係及びイベント間の関係の表現

すべてのプリミティブは，ほかの要素（イベントやプリミティブ）に依存している．どの要素にも依存していないプリミティブは，出力タイミングが決められないため存在しない．そこで，この依存関係をエッジ（後で述べるグラフ構造の要素）で表現することにする．

ところで，プリミティブは区間幅をもつ．また，2.3節で述べたように釣鐘型の形状でピークの位置を明示するため，プリミティブは開始点，ピーク点，終了点の3つの点（これらを「制御点」と呼ぶことにする）から構成される．よって「プリミティブがイベントに依存する」といった表現は正確でなく,「プリミティブのピーク点がイベントに依存する」などと表現する必要がある．これより，エッジがつなぐのはプリミティブの個々の制御点である．

さて，イベントを中心にエッジを辿ることで，多くの要素間の依存関係，つまりどちらがどちらに依存しているかを知ることができる．しかし，あるプリミティブが複数の要素に依存しているような場合（一例をFig.3.1に示す），エッジを辿るだけでは依存関係はわからず，エッジに明示しておく必要がある．そこで，エッジに向きを設けることによって依存関係を表現する．このとき，矢の先端に位置する点（プリミティブの制御点）がその反対側に位置する点（=基準点，ほかのプリミティブの制御点やイベント）に依存しているとする．

なお，2 章では詳細に述べなかったが，イベントとイベントの間にも時間的な位置関係がある．一連の運動からイベントのみを抽出したとき，このイベントの集合は何らかの構造（単純な場合ではリストだが枝分かれする場合もある）を形成していると考えられ，安定したイベントの検出のためには，イベント条件が満たされるだけでなく，イベントの履歴，つまりイベントとイベントの順序関係も利用した方がよい．そこで，この関係もエッジで表現することにする．このときのエッジの方向は，イベントが発生する順序を表すとする*.

3.1.2 時間的な位置関係の表現

時間的な位置関係（時間的な距離を含まない関係）としては，プリミティブとプリミティブの関係がもっとも複雑である．区間と区間の時間的な位置関係は，開始点と終了点の位置関係（前・後・同期）で分類すると13通りになるが，プリミティブの場合，区間に中間点（=ピーク点）をひとつ含むため，Fig.3.2に示すように63通りの分類ができる．

しかし，プリミティブとプリミティブの時間的な位置関係は，例えばピーク点とピーク点の位置関係（前･後･同期の3通りである）のみ考えればよい場合が多く，また，2.2節で述

* 正確には，以下で述べる初期ノード（運動開始時に発生するイベントを表す）を定めることによって，イベント間のエッジは，方向を定めなくても順序がわかるが，ほかのエッジと表現を統一するため，また部分的にイベント間の連結を見ても順序が読み取られるようにするため，敢えて方向をもたせる．

べたように位置関係が変化しても問題ない場合もあり，上記のようなタイプ分けによる表現は繁雑である．そこで「点と点の関係」を基本要素として時間的な位置関係を表現する．このとき「点と点」の間の関係として，既にエッジによって依存関係が表されており，このエッジに時間的な位置関係を表す情報を付加することにする．これにより，上記 63 通りの関係を表現可能で，ピークとピークのみといった場合も簡潔に表現される．なお，エッジにどのようにして時間的な位置関係を表す情報を付加するかについては，以下で述べる．

3.1.3 時間的な距離の表現

時間的な距離はプリミティブを出力するタイミングを決める上で必要である．このときプリミティブはほかのプリミティブやイベントといった要素に依存しており，この依存関係はエッジによって表現されている．そこで，このエッジに「メトリック」を定義することで，時間的な距離を表現する．

このとき，メトリックを実数全体（正・負・ゼロ）に対して定義することによって，前項で述べた時間的な位置関係を表現することにする．時間的な位置関係の明示（変化させてはならないとき）は，このメトリックに制約を加えることで表す．制約の加え方は後述する．

ところで，イベントとプリミティブの時間的な位置関係には，プリミティブがイベントよりも先行するような場合も含まれることは既に述べた.「イベントよりも前」という表現をもとにして制御信号を生成する（プリミティブの出力タイミングを決める）とき，イベントが発生するよりも前にプリミティブを出力しなければならないから，イベントの発生時刻を「予測」する必要がある．この「予測」とは，イベントとイベントの時間的な距離の予測であるから，イベント間をつなぐエッジのメトリックの役割とする．

3.1.4 メトリックの表現

以上より，メトリックの役割には，(1) プリミティブとイベント，あるいはプリミティブとプリミティブの間の，点と点の時間関係（距離と位置関係）を表現すること，(2) イベントとイベントの時間的な距離を予測することの 2 つがある．このことを踏まえて，メトリックの表現を議論する．

(1) の点と点の時間的な距離は，ほとんどの場合環境や身体の状態によって変動するので，変動しない場合も含めて，環境や身体の状態の関数で表現する．(2) のイベントの発生時刻は環境や身体の状態よって決まるため†，やはりこれらの観測状態を用いて予測する．

よって，いずれの場合も，メトリックの表現は環境や身体の状態の関数として表現される．これらの観測状態は 2.4 節で述べた観測状態空間 $\mathbb{Q}$ のひとつの要素であるが，時刻の関数でもあり，ある時刻 t における観測状態は $\mathbf{Q}(t)\,(\in \mathbb{Q})$ として表される．つまり「どの時刻の観測状態によって表現するか」を定めなければならない．これに関してはいくつかの可能性があるが，エッジの基準点の生起時刻‡と，(イベントの発生時刻の）予測が行われる時刻のふたつを導入する．つまり，

$$\mathsf{Metric} = \left\{ metric \mid metric = \{ m_0, m \mid m_0 : \mathbb{R} \times \mathbb{Q} \to \mathbb{R},\ m : \mathbb{R} \times \mathbb{Q} \to \mathbb{R} \} \right\} \tag{3.1}$$

によってメトリック空間 Metric を定義する．ただし m_0 はエッジの基準点が生起されたと

† 実際には，制御出力にも依存する．このような場合の表現をどうするかは今後の課題である（6 章参照）.

‡ 「点」が「生起」されるとは，EventGraph に基づいて制御を実行する際に「点」が時間軸に配置されることを指す．このときの時刻のことを「生起時刻」と表現する.

きに，m は（イベントの発生時刻の）予測が行われるときに評価され，m は予測が行われた時刻からイベントが発生するまでの時間差を返す[§]．また，m_0 と m は基本的には排他的である（どちらか一方のみ定義される）が，イベントの発生時刻の予測に両方とも使う場合（おそらく特殊である）も考慮して，特に制約は設けない[¶]．

このとき，前述したメトリックに対する制約（時間的な位置関係を明示するとき）は，関数として正または負，あるいはゼロになるようなものを用いるか，関数に制約を加えることによって表現する．例えばふたつのプリミティブの全制御点が同期することを表現するには，各点同士をエッジで結び，メトリックをゼロとすればよい．また，ピークとピークのみの時間的な距離関係（位置関係は制約しない）を表現するには，ピーク点とピーク点をエッジで結び，メトリックとして適当な関数を与えるだけですむ．

なお,「エッジのメトリックが正である」とは，基準点よりも後に依存している点が位置する（遅延）ことを表し,「負である」とはその逆を表す．

3.2 プリミティブの表現

プリミティブは「釣鐘型の制御信号」で，その形状は後方向の分散 Δt_{b}，前方向の分散 Δt_{f}，及びピーク $peak$ によって特徴付けられるとしたが（2.3 節参照），実際にはプリミティブの制御点は，ほかのプリミティブやイベントとの関係で調整され，その結果として各制御点の時間位置が決定されて制御信号が計算される（Fig.3.3）．制御点の数はこの場合 3 点だが，将来的に拡張することも考えて，数を制限しない．このときプリミティブに要請される最低限の要素は，Δt_{b}，Δt_{f} に対応する区間幅の集合 $\{\Delta t\}$ 及び制御点が定められたときにある時刻において制御信号を出力する関数 f である．すなわち，

$$\begin{aligned} Primitive = \big\{\{\Delta t\},\, f \mid \{\Delta t\} = \{\Delta t_j \mid \Delta t_j : \mathbb{R}\times\mathbb{Q}\to\mathbb{R}\ (\Delta t_j > 0), \\ j = 1,\ldots,N_{\mathrm{C}}-1\},\, f : \mathbb{R}\to\mathbb{O}_i\big\} \end{aligned} \tag{3.2}$$

となる．メトリックと同様に，プリミティブのパラメタは状態 $\mathbf{Q}(t)\,(\in\mathbb{Q})$ によって変動するため，時間と $\mathbf{Q}(t)$ の関数として定義されている．$N_{\mathrm{C}}(\in\mathbb{N})$ はプリミティブの制御点の数を表す．

3.3 EventGraph の定義

プリミティブとプリミティブ，イベントとプリミティブ，イベントとイベントの間の，すべての二項関係をエッジで表現すると，グラフ構造ができる．このグラフ構造は複数の関節に対する制御信号のプリミティブを含んでおり，プリミティブ，イベントの半順序構造である．このグラフ構造を EventGraph と呼ぶことにする．

以下 Fig.3.4 に示した EventGraph の一例を用いて，その概要を述べる．Fig.3.4 において，楕円の中の小さな円はプリミティブの制御点に対応しており，2 重の円はイベントに対応している．EventGraph はこのように 2 層構造になっており，下のレイヤには「ポイント」（図の小さな円）とそれらをつなぐ「エッジ」が存在する．ポイントとは，プリミティブの制御点とイベント（点）をまとめて表現したものである．上のレイヤでは下のレイヤのポイントを，プリミティブの制御点に属するものか，イベントに対応するものかに分けてお

[§] 時間差を返すのは，一般的にイベントが近付くほど予測が正確になるという事実による．例えば，誰かが投げたボールをキャッチするとき，イベント（キャッチ）が近付くほど予測が正確になる．

[¶] 予測時刻の計算方法は，付録 A に示してある．

り，いくつかのポイントをまとめる「制御ノード」（図の楕円）と，ひとつのポイントから成る「イベントノード」（2 重の円の外側）が存在する．プリミティブは，制御ノードの要素として保持されている．

EventGraph は，ポイントの集合 (Point)，エッジの集合 (Edge)，イベントノードの集合 (E-Node)，制御ノードの集合 (C-Node) と,「初期ノード」及び「終了ノード」から成る.「初期ノード」はどのノードから制御の実行を開始するか明示するためのノード,「終了ノード」は EventGraph の終了条件を明示するためのノードであり，終了条件はイベント条件の一種である．よって初期ノード及び終了ノードは，ともにイベントノードの一種である（初期ノードのイベント条件は，制御の実行が開始されること）．よって形式的な定義は，

$$\begin{aligned} \mathit{EventGraph} = \big\{ & \mathsf{Point},\ \mathsf{Edge},\ \mathsf{E\text{-}Node},\ \mathsf{C\text{-}Node},\ \mathit{start\text{-}node},\ \mathit{end\text{-}node} \\ & \mid \mathsf{Point} = \{\mathit{Point}\},\ \mathsf{Edge} = \{\mathit{Edge}\},\ \mathsf{E\text{-}Node} = \{\mathit{E\text{-}Node}\}, \\ & \quad \mathsf{C\text{-}Node} = \{\mathit{C\text{-}Node}\},\ \mathit{start\text{-}node} \in \mathsf{E\text{-}Node},\ \mathit{end\text{-}node} \in \mathsf{E\text{-}Node} \big\} \end{aligned} \tag{3.3}$$

となる．以下，各要素の形式的定義を与える．

ポイント: ポイントは，プリミティブの制御点とイベント（点）をまとめて表現した要素であり，EventGraph の構造を表現するために導入されている．ポイントに対しては，エッジなどのように特に要素を定義しない（もたない）が，制御信号を生成するアルゴリズムを設計する段階で要素が追加される（4.1.4 項）．

エッジ: エッジはポイントとポイントをつないでおり，依存関係を表す向きをもつ．また，これらのポイント間の時間的な距離を表すために，関数として表現されたメトリックを要素にもつ．つまりエッジは，

$$\begin{aligned} \mathit{Edge} = \big\{ & \mathit{from\text{-}point},\ \mathit{to\text{-}point},\ \mathit{metric} \\ & \mid \mathit{from\text{-}point} \in \mathsf{Point},\ \mathit{to\text{-}point} \in \mathsf{Point},\ \mathit{metric} \in \mathsf{Metric} \big\} \end{aligned} \tag{3.4}$$

として定義される．ここで $\mathit{from\text{-}point}$ は基準点，$\mathit{to\text{-}point}$ は基準点に依存しているポイント，metric はメトリックである．

イベントノード: イベントノードは，ひとつのポイントと，イベント条件を要素にもち，

$$\mathit{E\text{-}Node} = \big\{ \mathit{point},\ \mathit{event\text{-}cond} \mid \mathit{point} \in \mathsf{Point},\ \mathit{event\text{-}cond} \in \mathsf{EventCond} \big\} \tag{3.5}$$

として定義される．EventCond は 2.4 節で定義したイベント条件の集合である．

制御ノード: 制御ノードは，複数の制御点（2.3 節で述べた「釣鐘型のプリミティブ」を用いる場合は 3 点だが，ここでは 6 章で述べたような拡張を考慮して，点の数を制限しない）に対応するポイントの集合と，プリミティブ, 及び制御信号の出力先を要素にもつ．つまり，

$$\begin{aligned} \mathit{C\text{-}Node} = \big\{ & \{\mathit{point}\},\ \mathit{Primitive},\ \mathit{Out}, \\ & \mid \{\mathit{point}\} \subset \mathsf{Point},\ \mathit{Out} \in \mathsf{OutputPortSet}.\mathit{OutputPort}_i.\{\mathit{in}\} \big\} \end{aligned} \tag{3.6}$$

として定義される．ここで $\mathit{Primitive}$ はプリミティブ，Out は出力ポート $\mathit{OutputPort}_i$ の入力接続集合 $\{\mathit{in}\}$ のひとつ，OutputPortSet は出力ポート全体の集合である．なお，$|\{\mathit{point}\}|$ は $\mathit{Primitive}.N_{\mathrm{C}}$ に等しい．

第 4 章 EventGraph に基づく制御アルゴリズム

本章では，3 章で導入した EventGraph に基づいて，2 章のイベント駆動型制御を実行するためのアルゴリズムについて述べる．

4.1 制御アルゴリズムの基本要素

EventGraph は「イベントよりも前に実行される動作（=プリミティブの出力)」という記述を含み，これを実現するために「予測」が導入されている．このため「イベントの待機→(遅延)→ 動作」という実行の流れでは制御信号を生成できず，イベントを予測し，その結果を用いてプリミティブを出力する時刻を決定して行く必要がある．この決定は，イベントからエッジを矢の向きに辿ることによって，末端のプリミティブまで伝搬される．

EventGraph に基づく制御アルゴリズムは，プリミティブの出力，イベントと制御ノードの制御点の監視，及びイベントの予測とそれに伴う制御点の生起時刻の更新から成り立っている．以下ではこれらの項目について述べる．

4.1.1 プリミティブの出力

制御ノード（プリミティブを要素としてもつ）は区間幅をもち，ある期間だけ制御信号を出力する（プリミティブの関数 f が評価され，出力ポートに送られる）．そこで，制御信号を出力中である制御ノードを，制御ノード全体の集合に対する部分集合として定義しておく．

$$\{active\ c\text{-}node\} \subset EventGraph.\text{C-Node} \tag{4.1}$$

4.1.2 ポイントの監視

制御ノードをいつ $\{active\ c\text{-}node\}$ に加え，いつ取り出すかは，その制御ノードを構成する制御点の生起時刻（出力される時刻）で決まる．よって，これらの制御点が生起されるべきか否かを監視しておく必要があるが，当然ながらすべての制御ノードの制御点を監視する必要はない．

一方，イベントの発生は，イベント条件が満たされるがどうかを常に監視することで検出される．この監視はすべてのイベントに対して行われるわけではなく，前のイベント*が既に発生した場合にのみ行われる．つまりすべてのイベントノードに対してイベントの発生を監視する必要なない．

以上から制御ノードの制御点とイベントノードを監視しておく必要があり，これらをまとめてポイントを監視することによって実現する．そのために監視される対象のポイントの集合を定義する．

$$\{waiting\ point\} \subset EventGraph.\text{Point} \tag{4.2}$$

---

* 正確には，あるイベントノードのポイントに接続されているエッジを，矢と反対方向に辿ったところに位置するイベントノード．

4.1.3 イベントの予測

予測は 3.1.3 項で述べたように，エッジのメトリックによって行われる（具体的な計算方法については付録 A を参照）．この予測は，常にすべてのエッジで行われる必要はないため，予測が行われているエッジ（「アクティブであるエッジ」と表現する）を，エッジ全体の集合に対する部分集合として定義する．

$$\{active\ edge\} \subset EventGraph.\mathsf{Edge} \tag{4.3}$$

4.1.4 ポイント生起時刻の保存と更新

制御ノードの制御点の生起時刻はイベントを中心にエッジを矢の向きに辿ることで計算されるが，イベント発生時刻の予測が変動しない場合には，一度計算するだけでよい．そこで，ポイントに，生起時刻を保存しておくための要素を加える．

$$Point = \left\{ t \mid t \in \mathbb{R} \right\} \tag{4.4}$$

イベント発生時刻の予測は常に正確であるわけではなく，身体や環境の状態によって変動する．よって，イベントの予測時刻が変更されたら，それに伴ってイベントに連結されている制御ノードの制御点の生起時刻 $Point.t$ も，エッジを辿って更新して行く必要がある（更新の伝搬）．

イベント予測時刻が変更されたかどうかは，過去の予測時刻との差がある範囲を越えたかどうかで判断する．具体的には,「過去に推定された予測時間と新たな予測時間の差が，両者の平均の $\varepsilon\,(\in \mathbb{R})$ 以上になったら更新する」ことにした．つまり，過去の推定時間 t_1 と新たな予測時間 t_2 の間に

$$|t_1 - t_2| < \frac{\varepsilon}{2}|t_1 + t_2| \tag{4.5}$$

が成立しなくなったとき更新する（この条件を更新条件と呼ぶ）．なお，ε の値は具体的には 5% とした（この値に厳密な基準はない）．

4.2 EventGraph に基づく制御アルゴリズム

前節で述べた基本的な要素をもとにして，制御信号が生成される．この流れを Algorithm 1 に示す．なお，イベント駆動型制御はディジタル計算機上で実行されることを想定しており，`while` ループは毎回のサンプリング時刻に対して実行される．

Algorithm 1: EventGraph に基づく制御アルゴリズムの流れ

```
初期化
while {waiting point} ≠ φ ∧ {active c-node} ≠ φ ∧ {active edge} ≠ φ do
  time ← 現在時刻
  for all Point in {waiting point} do
    if Point を生起させるべきか then
      Point を生起
      {waiting point}，{active c-node}，{active edge} に対する追加・削除処理
      必要に応じて制御ノードの各ポイントの t を計算
    end if
  end for
  for all C-Node in {active c-node} do
    C-Node の出力処理
  end for
  for all Edge in {active edge} do
```

```
      Edge のメトリックを再計算
      if Edge の先端のポイントが更新条件を満たすか then
        更新を伝搬
      end if
    end for
  end while
```

このアルゴリズムが実行可能であるためには，EventGraph にいくつかの制約を加えておく必要がある†．この詳細を，B.1 節に示す．

また，この実行の流れからもわかるように，終了ノード ($\in$ E-Node) が生起されたとしても，制御の実行が終了するわけではない．しかし終了ノードを定めておくことで，終了ノードが生起されたときに，ほかの EventGraph に基づいた制御を開始することが可能となる（このとき両者の出力が重複しても，出力ポートによって処理され，重複による問題は生じない）．

4.3 制御アルゴリズムの効率化

EventGraph の構造が複雑になると，イベント発生予測時刻の変動による更新のコストが大きくなる．また，$\{waiting\ point\}$ のポイントは常に生起されるかどうかの判断対象になるので，できるだけ少ない方がよい．そこで，計算量を軽減するためにいくつかの制約を設けた．この詳細を B.2 節に示す．なお，エッジの矢の先端に位置するポイントを従属ポイント，その反対に位置するポイントを基準ポイントと呼んでいる．

これらの制約のもとで，$\{waiting\ point\}$ に加えるポイントを限定する．イベントノードのポイントは，予測が行われるべきか否かで $\{waiting\ point\}$ に加えられ，これに関して限定することはできないから，制御ノードを構成するポイントについて限定する‡．

例えば制御ノードの 2 番目のポイントは，1 番目のポイントが生起されるまで $\{waiting\ point\}$ に含める必要がない（1 番目のポイントと 2 番目のポイントの生起時間間隔が正だから）．同様に考えて,「あるポイント P_1 が，それ以前に生起される予定のポイント P_2 とエッジで直接つながっていれば，P_1 は P_2 が生起するまで $\{waiting\ point\}$ に加える必要がない」と言える．このとき，このルールで除外できないポイント（制御ノードを構成するポイント）を *Starting Point* と呼ぶことにする．

ポイントに接続されているエッジのメトリックが負かそうでないかによって，除外できるかできないかは変わる．しかし，Constraint 8 により，エッジのメトリックが負かそうでないかは実行中変わらない．よって，以下のように *Starting Point* を定義できる．

Starting Point 制御ノードの最初のポイントで，このポイントを従属ポイントとするエッジのメトリックはすべて負であり（従属ポイントとするエッジがなくてもよい），このポイントを基準ポイントとするエッジのメトリックはすべて負でないようなポイント

あるポイントが *Starting Point* かどうかを判定する関数を IsStartingPoint とする：

$$\mathsf{IsStartingPoint} = \mathsf{Point} \to \{true, false\} \tag{4.6}$$

以上のことを考慮した，EventGraph に基づく制御アルゴリズムを付録 C に示す．

† 例えばグラフ構造が複数のグラフに分けられる（エッジでつながっていない）なら，すべての制御ノードを出力できない．

‡ なお $\{active\ c\text{-}node\}$，$\{active\ edge\}$ に加える要素も同様に限定したいが，これらを減らすことは難しい．

第 5 章 イベント駆動型制御に基づく運動学習とその評価

5.1 目的と方針

イベント駆動型制御に基づく運動学習とは，与えられた評価関数を最大にする EventGraph を求めることである．この学習は 1.3.2 項で述べたように，(1) グラフ構造を固定した最適化（プリミティブのパラメタやエッジのメトリックを最適化）と (2) グラフ構造自体の最適化の 2 段階から成る (Fig.1.2)．

本研究の最終的な目標は，多様な動的環境下においてこれらの最適化を実現し，多自由度ロボットに運動を学習させることである．そこで本章では，人間と類似の身体性をもったロボットに，人間の定性的な運動知識に基づいて EventGraph の構造を与え，(1) のグラフ構造を固定した最適化を実現することで，イベント駆動型制御が制御則として有効に機能することを明らかにする．さらに，(2) のグラフ構造自体の最適化についての基礎的な検討として，ロボットの身体性を人間とは異なるものに変えた場合でも，EventGraph の構造を変えることによって望ましい運動が実現可能か検証する．

具体的な実験としては，トランポリン上で，多自由度ロボットに跳躍運動を学習させる．トランポリンは，ロボットの運動によって状態（位置や速度）が変化する，動的な環境であり，このため 1 章で述べたような，制御の困難さがある．

なお本実験は，Featherstone の順動力学アルゴリズム [12] をもとに，トランポリン（平面で大幅に変形しない）と多自由度ロボットの動力学シミュレーション環境を構築して行った．この詳細を付録 E に記載する．

以下，5.2 節で構造を固定したパラメタ最適化の手法について述べ，5.3 節でその評価実験を行う．5.4 節では構造最適化の基礎的な検討を行う．

5.2 EventGraph の構造を固定したパラメタ最適化の手法

5.2.1 最適化問題の定式化

「構造を固定した最適化」とは，EventGraph のイベントノード，制御ノード，エッジ，及びポイントの数を固定し，さらにそれらの連結関係，イベントノードのイベント条件と制御ノードの出力先を固定した条件下での，エッジのメトリックと，プリミティブのパラメタの最適化である．なお，イベント条件が固定であるから，イベントの予測を行うメトリックについては，予測関数を解析的に求めて与えた．つまり，このときの最適化問題は，

$$\mathbf{X}^* = \arg\max_{\mathbf{X}} \mathsf{Evaluate}(\mathit{EventGraph}) \tag{5.1}$$

where

$$\begin{aligned}\mathbf{X} = \big\{ &\{\mathit{Edge.metric} \mid \mathit{Edge} \in \mathit{EventGraph}.\mathsf{Edge},\ \mathit{Edge.to\text{-}point} \neq \mathit{E\text{-}Node.point} \\ &\qquad \text{for all } \mathit{E\text{-}Node} \in \mathit{EventGraph}.\mathsf{E\text{-}Node}\}, \\ &\{\mathit{C\text{-}Node.Primitive} \mid \mathit{C\text{-}Node} \in \mathit{EventGraph}.\mathsf{C\text{-}Node}\}\big\}\end{aligned} \tag{5.2}$$

のように定式化される（ただし $\mathbf{X}$ は最適化対象のパラメタベクトル，Evaluate は後述する評価関数である）．エッジのメトリックやプリミティブの要素については，時間と観測状態空間 $\mathbb{Q}$ の直積空間から実数への写像であるということしか定めていないので，以下で具体的に定義を与える．

メトリック: 本研究の実験では，簡単のため，最適化対象のメトリックには定数を用いた（これに合わせてタスクも固定的なものにした）．すなわち，

$$Edge.metric.m_0 \in \mathbb{R},\ Edge.metric.m = 0 \tag{5.3}$$

として，定数を最適化の対象にした．

プリミティブ: メトリックと同様に，プリミティブのパラメタにも定数を用いた．この実験で用いたプリミティブの定義は，

$$Primitive = \{\Delta t_{\mathrm{b}},\ \Delta t_{\mathrm{f}},\ peak,\ f \mid \Delta t_{\mathrm{b}} \in \mathbb{R},\ \Delta t_{\mathrm{f}} \in \mathbb{R},\ peak \in \mathbb{R}\} \tag{5.4}$$

である．時刻 $time$ に対して制御信号を出力する関数 $Primitive.f$ は，釣鐘型の形状をした，次の関数を用いた．

$$Primitive.f(time) = \begin{cases} peak \cdot \exp\left(\frac{-(time-t_2)^2}{2(t_2-t_1)^2}\right) & (time < t_2) \\ peak \cdot \exp\left(\frac{-(time-t_2)^2}{2(t_3-t_2)^2}\right) & (time \geq t_2) \end{cases} \tag{5.5}$$

なお t_1, t_2, t_3 は，プリミティブを保持している制御ノードの制御点*が生起される時刻 $(t_1 < t_2 < t_3)$ であり，プリミティブのパラメタ $\Delta t_{\mathrm{b}},\ \Delta t_{\mathrm{f}}$ に基づいて決定される（付録 C）．

5.2.2 最適化手法

最適化手法としては，パラメタと評価関数の関係が不明であるから微分情報を用いられないこと，また，構造の最適化も視野に入れていることから遺伝アルゴリズム (GA) を用いた†．GA はメタアルゴリズムであり，具体的な構成は，世代交代モデルには MGG (Minimal Generation Gap[13])，交叉オペレーションには SPX（シンプレックス交叉 [14]）を用いた．EventGraph に対する適用方法の詳細については，付録 D に記載する．

5.3 EventGraph の構造を固定した最適化の評価実験

ここでは 5.2 節で述べた，EventGraph の構造を固定した最適化を行い，解の収束性を調べることによって最適化が成功するか評価する実験を行う．

5.3.1 状況設定

達成すべき運動（タスク）を「一定の高さ‡から落下させ，トランポリン上で跳躍し，できるだけ高い位置まで跳ぶ」という設定にした．このタスクを実現するために獲得すべきパラメタは毎回の状況が同じだから，5.2.1 項で述べたようなメトリックやプリミティブ（関数が定数）でもタスクの実現には影響を与えない．

---

* $C\text{-}Node.\{point\}$ $(|\{point\}| = 3)$

† この妥当性については 6 章を参照

‡ 具体的には，トランポリンからロボットの頂点までの距離が 2.7m.

5.3.2 多自由度ロボットの構成

多自由度ロボットの構成は，Fig.5.1 に示すように，4 リンク 3 関節とした．リンクの集合を $\mathcal{L} = \{L_0, L_1, L_2, L_3\}$ とし，それらの間の関節の集合を $\mathcal{J} = \{J_1, J_2, J_3\}$，対応する関節の状態（角度）を q_1, q_2, q_3 とする．各関節には稼働範囲を制限するために，ある範囲を越えたところにバネ・ダンパを設けてある．各関節にはアクチュエータ（モータ）が取り付けられており，それぞれ u_1, u_2, u_3 $(\in \mathbb{R})$ の制御入力をもつ．多自由度ロボットの構成の詳細は E.1 節で述べる．

このロボットは人間の足をモデルに設計されており，例えば Fig.5.1(c) に示した関節の制約は人間のものと近くなっている[§]．このため，人間の定性的な運動知識に基づいて構成した EventGraph を，そのままこのロボットに対しても適用できると考えられる．

5.3.3 EventGraph の構造

EventGraph の構造は，人間の定性的な運動知識に基づいて Fig.5.2(b) のように与え，これを固定して 5.2 節で述べたパラメタ最適化する．このときの運動は Fig.5.2(a) のようになることを想定している．「定性的な運動知識」と *EventGraph* の関係は，次のようなものである．

> 運動開始（イベントノード E_S）後，着地（イベントノード E_1）の前に足を曲げ（制御ノード C_{11}, C_{21}, C_{31}），着地の直後に足をふんばり（制御ノード C_{12}, C_{22}, C_{32}），トランポリンが十分に沈みきってから足に力を加えて（制御ノード C_{13}, C_{23}, C_{33}）跳躍する．離陸（イベントノード E_2）したらバランスを取る制御（制御ノード C_{14}, C_{24}, C_{34}）をする．高さがもっとも高くなったら（イベントノード E_E）終了．

なお着地イベントの予測は，E_S-E_1 間のエッジ ($Edge.\textit{from-point} = E_\mathrm{S}.point \wedge Edge.\textit{to-point} = E_1.point$) のメトリックを，

$$Edge.metric.m_0 = 0, \quad Edge.metric.m = \frac{1}{g}\left(\sqrt{2gh + v^2} - v\right) \tag{5.6}$$

とすることによって行う（ここで g は重量加速度，h はロボットの最下部とトランポリンの距離，v はリンク L_0 の垂直方向の速度である）．ほかのイベントに関しては，それらよりも前に出力される制御ノードが接続されていないため 0 とした．

5.3.4 評価関数

評価関数は，跳躍後の高さと跳躍後のバランスで定義する．つまり，跳躍後の高さが高いほど評価値が高く，跳躍後のバランスが取れているほど評価値が高くなるようにする．バランスが取れているかどうかは，L_0 姿勢が z 軸（垂直軸）に平行かどうかと，各関節の角度が 0 に近いかを数量化する．また，L_0 の角速度，各関節の角速度が小さいほど評価が大きくなるようにした．具体的には，

$$\begin{aligned}\mathsf{Evaluate} = \frac{5}{2} \cdot \textit{after-height} + 2(1 + R_{33})^2 + \frac{4}{1 + |\boldsymbol{\omega}_0|} \\ + \sum_{i=1}^{3} \left\{ (1 + \cos q_i)^2 + \frac{2}{(1 + \dot{q_i}^2)} \right\}\end{aligned} \tag{5.7}$$

[§] ただし，胴体から上のモデルであるリンク L_0 の質量は，タスクが跳躍なので軽くしてある．また，各リンクも人間と比べるとやや長い．

とした．ここで *after-height* は跳躍後の高さを，R_{33} はエージェントの姿勢行列の (3,3) 要素を[¶]，$\boldsymbol{\omega}_0$ はリンク L_0 の角速度ベクトルをそれぞれ表す．

5.3.5 実験結果と考察

1 回の学習を「GA の世代が 400 回交代するまで」と定義して[‖]，学習を 15 回行った．各世代において個体がもつ評価値から最大のものを選び，15 回の学習での平均と分散を世代番号に対してプロットしたグラフを Fig.5.3 に示す．この結果から，初期の段階では初期値をランダムに与えているためにばらつきがあるが，最終的には収束する方向に向かっていることがわかり，学習が毎回のタスクで成功していると判断される．

学習によって得られたタスクの様子を，Fig.5.4 に示す．この図からも運動を実現できていることが確認され，イベント駆動型制御の有効性を確かめられたと言える．

5.4 EventGraph の構造最適化の基礎的検討

目的と状況設定: 前節の実験で用いた *EventGraph*（Fig.5.2(b)）は，人間の定性的な運動知識に基づくものである．つまり，人間と身体性が異なるロボットに対しては，タスクが達成できるとは限らない．本節では，このような場合でも EventGraph の構造を変化させることでタスクが達成できるようになることを，実験によって示す．

具体的には，Fig.5.1(c) に示した関節角の制約を Fig.5.5 のように変更した（関節 J_1 間の制約を $[-\frac{\pi}{2}, 0]$ から $[-\frac{\pi}{2}, \frac{\pi}{2}]$ に変更）．

同じ構造での学習: まず，構造を前節の実験と同じ Fig.5.2(b) にして，5.3.5 項と同様の実験を行った．このとき，同様の解析法でその平均と分散をプロットしたグラフを Fig.5.6 に示す．

評価関数の収束値が前節の実験結果（Fig.5.3）よりも悪くなっているが，これは関節 J_1 の制約を緩めることによって跳躍後のバランス調整が困難になっているからだと推測される．

構造を変化させた学習: 次に，跳躍後のバランス調整のための制御を増やすことによって，評価関数の改善を試みる．具体的には，Fig.5.7 のような *EventGraph* を構成し，この構造を固定して同様の実験を行った．

実験の結果を同様の解析方法でプロットしたグラフを，Fig.5.8 に示す．評価関数の収束値が，構造を変えずに学習した場合（Fig.5.6）よりも向上しており，EventGraph の構造を変えることによって，異なる身体性をもったロボットに対しても有効な制御則が獲得できることが示された．また，この結果から「構造の最適化」が有効に行える可能性が示唆される．

なお，理論的には，制御ノードの数を増加させればそれだけ制御の精度が上がるわけだが，その分最適化のコスト，及び制御信号の生成コストが大きくなる．また，消費エネルギも大きくなるだろう．よって構造の最適化の際には，制御ノードの数を抑えるように評価関数を設計すればよいと考えられる．

[¶] Z-Y-X オイラー角表現における x 軸回転を φ，y 軸回転を θ としたとき $R_{33} = \cos\theta\cos\varphi$ である．

[‖] 1 個体にひとつの EventGraph が割り当てられており，評価値はタスクを 1 回行って計算する．また，1 回の世代交代において 8 個の子孫を生成している．「400 回」という数字は，何度か予備実験を行って判断した．

第 6 章 結論

6.1 イベント駆動型制御の適性に関する議論

この節では，イベント駆動型制御に基づく運動学習が有効に行えた理由について，評価関数の複雑さ及び近接する制御信号における独立性の観点から論じ，その適性について検討する．

5 章では，ある程度簡略化された状況下であるとは言え，動的な環境下で多自由度ロボットに跳躍運動を学習させることに成功した．このような学習が可能であった理由は，満たすべき評価関数の複雑さ（独立な変数の数）が，参照軌道を評価関数にするような場合ほど大きくなかった点にある．そのためイベント駆動型制御のような，少ないパラメタでも最適化が行えたと言える．もし仮に，評価関数が「参照軌道の追従」であったとするなら，今回使用した評価関数の複雑さが，参照軌道のサンプリング点数倍になって，良好な結果は得られなかったと考えられる．

この事実は，しかし，イベント駆動型制御の価値を損なわせるものではない．例えば 5 章の実験と同じ学習を，制御信号をグリッドに区切って行ったとするなら，5 章で用いたパラメタの数よりも，はるかに多くのグリッド数に分割する必要があるものと考えられる．なぜなら，Kuniyoshi らの知見からもわかるように，近接するグリッド（制御信号）は，投入するエネルギという観点から見て独立ではないからである．

この点，イベントが身体の力学的拘束条件の変化なら，そこで微分方程式が変化し，それによってイベント前後の制御信号における独立性が強くなると考えられ（もちろん，完全に独立にはなり得ない），イベント駆動型制御でのイベントの前後という表現によってプリミティブ間の独立性が保たれているのではないかと考えられる．

以上のことから，イベント駆動型制御は「軌道の追従」と言った，精密な制御にはそれほど適していないことがわかる．もちろんプリミティブの数を増やせばそれだけ精度は上がるが，学習は困難になる．今回使用したような，満たすべき評価関数がそれほど複雑でない場合に，イベント駆動型制御に基づく学習は真価を発揮するものと考えられる．

6.2 結論と課題

本論文では，動的な環境下で多自由度ロボットに運動学習をさせるため，Kuniyoshi らの知見に基づいてイベント駆動型制御を設計し，その表現として EventGraph を提案した．また，シミュレーションによる実験により動的環境下における運動学習に対する有効性を示し，身体性（関節角の制約）が変化したときに EventGraph の構造を変化させることでタスクの達成が可能であることを示した．これらのことから，イベント駆動型制御に基づく運動学習は有効な手段となり得ると考えられる．しかし，以下の項目については，本論文では十分に扱うことができなかったため，今後の課題とする．

EventGraph の表現能力に関する課題: イベント駆動型制御の表現として提案した EventGraph は，イベントとプリミティブ，プリミティブとプリミティブ，イベントとイベントの間のすべての時間関係を表現することができるが，プリミティブとして本研究で使用した

ような単純なものを用いる場合は，制御点の中のどれか一点について前・後・同期を表現しておけば十分な場合が多く，構造の学習においてはこの表現能力の高さが却って徒となる可能性がある（過学習）．これを解決するためには，何らかの制約を設けるか，確率によってどのような接続方法を優先させるかを決めるといった仕組みが必要となる．

イベントの表現に関する課題: 本研究では，局所的な制御信号の出力タイミングがイベントによって決められるという場合を考えたが，イベントが直接制御可能な場合もある（例えば手を叩くときの，手に力を入れる制御と，手のひらが合わさるというイベント）．このような場合，どのようにメトリックを表現し，制御するのか，未考察である．

また，本研究で扱ったイベントは，運動において必ず存在することを前提にした決定性のものであった．しかし，今後複雑な運動や複合的な運動を考える上では，非決定性のイベントも扱う必要性が出て来る可能性がある．

プリミティブに関する課題: プリミティブの形状としては単純な釣鐘型のものを用いたが，より精密な制御をする上では，何通りかの種類があった方がよい．その場合，探索空間の拡大をまねくから，今回使用したような単純なものを使って大域的に学習した後，より精度を上げるために制御点を増やして行く，という段階分けが必要であると考えられる．

構造を固定した最適化手法に関する課題: 構造を固定した最適化手法についても，今回はGA を用いたが,「報酬（評価）が行動（プリミティブの出力）よりも遅れる」ことを考えると，このような場合を対象にした最適化手法である，強化学習の方が適切であったと思われる．GA を用いた理由は，構造の最適化も見据えていたからであるが，構造の最適化と，構造を固定した最適化は分けて考えた方がよい．

また，エッジのメトリックやプリミティブのパラメタを固定（定数）で最適化したが，環境の変動に対応するためにも，観測状態の関数として学習すべきである．よって構造を固定した最適化についても，検討を深めて行く必要がある．

EventGraph の構造最適化: これらを踏まえ，構造の最適化について考案していく必要がある．5 章で行った基礎的な検討から，制御ノードの追加･削除に関しては，構造に対する GA などを適用することで対処できると考えられる．しかし，大きな問題となるのは，いかにイベントの抽出を行うかである．観測状態の系列からボトムアップにイベントを抽出するのは困難であると考えられ，イベントの抽出に特化した議論が必要である．

謝 辞

松山隆司教授には本研究の核となる議論において，極めて的確な指摘を頂き，また，研究者としてのあり方についてもさまざまな助言をして頂きました．深く感謝致します．

川嶋宏彰助手 には研究の細部に渡って深い議論をして頂き，研究，論文の作成にあたって熱心かつ厳格な指導をして頂きました．深く感謝致します．

鷲見和彦客員教授，牧淳人助教授，波部斉助手には研究会及び日常での議論を通してさまざまな助言をして頂きました．深く感謝致します．

松山研究室の皆様にも，日々の研究生活の中でさまざまな助言，支援をして頂きました．深く感謝致します．

参 考 文 献

1) Yasuo Kuniyoshi, Yoshiyuki Ohmura, Koji Terada, Akihiko Nagakubo, Shin'ichiro Eitoku, and Tomoyuki Yamamoto, Embodied basis of invariant features in execution and perception of whole body dynamic actions - knacks and focuses of roll-and-rise motion, *The Second International Workshop on Man-Machine Symbiotic Systems*, (2004), pp. 289–301.

2) H.Miyamoto, M.Kawato, et al., A kendama learning robot based on bi-directional theory, *Neural Networks*, Vol. 9, No. 8, (1996), pp. 1281–1302.

3) 村田栄理, 浅井孝宣, 佐久間淳, 小林重信, 強化学習による多自由度 2 足歩行ロボットの制御, *The 19th Annual Conference of the Japanese Society for Artificial Intelligence*, (2005).

4) 吉田成徳, 水内郁夫, 稲葉雅幸, 國吉康夫, 井上博允, 脊椎構造を持つ人間型ロボットにおける GA による匍匐動作の自動獲得, 日本機械学会ロボティクス・メカトロニクス講演会'02 講演論文集, (2002), pp. 2P2–L04.

5) 美多勉, 非線形制御入門 : 劣駆動ロボットの技能制御論, (昭晃堂, Nov., 2000).

6) 鈴木高宏, 中村仁彦, 2 階非ホロノミックシステムの平均化法による解析, *the 14th Annual Conference of the Robotics Society of Japan*, (1996).

7) 比留川博久, 加賀美聡, ヒューマノイドの知能—個としての知能—, 日本ロボット学会誌, Vol. 20, No. 5, (2002), pp. 478–481.

8) M.Ishikawa, A.Neki, J.Imura, and S.Hara, Energy preserving control of a hopping robot based on hybrid port-controlled hamiltonian modeling, *IEEE Conference on Control Applications (CCA2003)*, No. CF002507, (2003).

9) 木村元, 山下透, 小林重信, 強化学習による 4 足ロボットの歩行動作獲得, 電気学会電子情報システム部門誌, Vol. 122-C, No. 3, (2002), pp. 330–337.

10) Seiichi Miyakoshi and Gordon Cheng, Ballistic walking by compass-like biped walker - exploiting physical dynamics in achieving human-like walking, *Proceedings of 5th International Conference on Climbing and Walking Robots (CLAWAR2002)*, (Professional Engineering Publishing, Sep., 2002), pp. 445–452.

11) Steve Collins, Andy Ruina, Russ Tedrake, and Martijn Wisse, Efficient bipedal robots based on passive-dynamic walkers, *Science*, Vol. 307, No. 5712, (18 Feb., 2005), pp. 1082–1085.

12) Roy Featherstone, *Robot Dynamics Algorithms*, (Kluwer Academic Publishers, 1987).

13) 佐藤浩, 小野功, 小林重信, 遺伝的アルゴリズムにおける世代交代モデルの提案と評価, 人工知能学会誌, Vol. 12, No. 5, (1997), pp. 734–744.

14) 樋口隆英, 筒井茂義, 山村雅幸, 実数値 GA におけるシンプレックス交叉の提案, 人工知能学会論文誌, Vol. 16, No. 1, (2001), pp. 147–155.

15) 梶田秀司, ヒューマノイドロボット, (オーム社, 2005).

16) 内山勝, 中村仁彦, ロボットモーション, 岩波講座, ロボット学 2, (岩波書店, 2004).

付 録A　メトリックの拡張とイベントの予測

3.1.4 項で述べたメトリックについて，A.1 節では学習のための拡張に関する考え方を，A.2 節では予測の計算方法を，それぞれ述べる．

A.1　メトリックの学習と意義

3.1.4 項ではメトリックの要素 m_0，m を時間（実数）空間と観測状態空間の直積から実数への写像 $(\mathbb{R}\times\mathbb{Q}\to\mathbb{R})$ として定義したが，この写像を学習によって獲得することは難しい．なぜなら，メトリックはエッジの数だけ存在し，観測状態空間もロボットが多自由度であるためにかなり大きいからである．

本研究では，写像を獲得するまでは至らなかったが（定数として学習した），将来的には写像を学習することを目標としている．そのためには，もっとも単純には，観測状態空間の各要素の線形和として表現し，そのときの係数ベクトルを学習することが考えられる．あるいは，ニューラルネットワークを構成するという手段も使えるかもしれない．

いずれにせよ,「観測した状態を入力として」学習する．つまり，環境のモデルパラメタや身体モデルのパラメタを直接使わない．もちろん，これらのモデルとパラメタが既知であるなら利用すればよいかもしれないが，環境のパラメタは，多自由度ロボットを設計する段階ではわからない．よって，このような学習をすることで，環境や身体モデルのパラメタが不要になり，さらに，それらを推定して与えるよりも頑強になる可能性がある．

ここではメトリックについて，写像を学習するための基本的な考え方と意義について述べたが，同様の議論はプリミティブのパラメタに対しても成立する．

A.2　メトリックの表現と予測

本節では，メトリックの要素 m_0 と m がともに与えられた場合に，イベントの生起時刻を予測する計算方法について述べる．もちろん，この計算方法は，どちらかが独立に与えられた場合に対しても適用できる．なお，3.1.4 項で述べた通り，m は「予測が行われた時刻からイベントが発生するまでの時間差を返す」ことに注意されたい．

以下では，m_0 や m を時刻 t において評価することを，$m_0.\mathsf{eval}(t)$ や $m.\mathsf{eval}(t)$ のように表記する．m_0 や m は時刻の関数として定義されているから，その意味では $m_0(t)$ のような表記が適切だが，前節で述べたようにこの定義は将来的に拡張されるので，$m_0.\mathsf{eval}(t)$ のように表記する．

時刻 t_1 においてエッジの基準ポイントが生起されたとする．このとき時刻 $t, (>t_1)$ において，従属ポイントの生起時刻 t_2 を予測することを考える．時刻 t における残り時間（予測）を $rest$ とする．時刻 t_1 の時点で m_0 が評価され，そのときの値を r_0 とする．また，時刻 t における m の評価を r とおく：

$$r_0 = m_0.\mathsf{eval}(t_1) \tag{A.1}$$

$$r = m.\mathsf{eval}(t) \tag{A.2}$$

まず，m がゼロである場合を考える．このとき $t_2 = t_1 + (r_0 + r) = t_1 + r_0$ が常に成立するから，

$$rest = \frac{r_0}{t_2 - t_1}(t_2 - t) \tag{A.3}$$

となる．m がゼロでない場合，

$$rest = \frac{r_0}{t_2 - t_1}(t_2 - t) + r \tag{A.4}$$

$$t_2 = t + rest \tag{A.5}$$

が成立し，これらを t_2 について解いて，

$$t_2 = \frac{1}{2}\left\{(t + t_1 + r_0 + r) \pm \sqrt{D}\right\} \tag{A.6}$$

where

$$D = (t - t_1 - r_0 + r)^2 + 4r_0 r \tag{A.7}$$

を得る．$t = t_1$ の場合について計算すると，

$$t_2 = \frac{1}{2}\left\{(t + t_1 + r_0 + r) \pm |r_0 + r|\right\} \tag{A.8}$$

となるが，メトリック $(r_0 + r)$ は付録 B の Constraint 3 より正，また式 (A.8) が $t_2 = t_1 + (r_0 + r)$ と恒等的に等しくなる必要があるので，複合は $+$ である．よって，

$$t_2 = \frac{1}{2}\left\{(t + t_1 + r_0 + r) + \sqrt{D}\right\} \tag{A.9}$$

を得る．$r_0 = 0$，つまり $m = 0$ の場合は $t_2 = t + r$ が成立する．

付 録B　EventGraphの制約

B.1　生成可能にするための制約

4.2 節で述べたアルゴリズムを実行可能にするための制約を示す．なお,「EE エッジ」とは，イベントノードのポイントとイベントノードのポイントをつなぐエッジのことである．

Constraint 1. **初期ノードからエッジを順方向に辿って，すべてのノードに辿り着けなければならない**

この条件が満たされないと，時間軸上にマッピングできないため，実行できない．

Constraint 2. **初期ノードからエッジとイベントノードのみを辿って終了ノードまで辿り着けなければならない**

Constraint 3. EE **エッジのメトリックは正である**

イベントは直接制御不可能なので，時間軸上での調整ができない（調整するためには何らかの制御を行う必要があるので間接的になる）．このため，イベントとイベントの関係は遅延のみという制約を設ける．

Constraint 4. **負のメトリックの大きさは** EventGraph **の実行開始の時点で決定される**

3.1.4 項で述べたように，エッジのメトリックは観測状態 $\mathbf{Q}(t)$ の関数として表されるが，メトリックが負の場合，基準ポイントが生起されるのは従属ポイントが生起された後なので，基準ポイントが生起されたときの観測状態に依存していると，計算が実行できない．

Constraint 5. **イベントノードのポイントがあるエッジの従属ポイントであるとき，そのエッジは** EE **エッジでなければならない**

イベントノードのポイントの生起時刻は，制御ノードのポイントに依存してはならない，という意味である．制御信号を，イベントを基準にして時間軸にマッピングするわけだから，ある意味では当然である．

Constraint 6. **あるポイントを従属ポイントとするエッジはひとつまでしか存在できない**

つまり，ひとつのポイントに 2 つ以上のエッジが終端点として（矢の先として）接続されない．これは，仮にこのようなことを許した場合，どちらのエッジの基準ポイントを基準にして生起時刻を定めるかがわからないからである（逆にそれを定義すればこの制約はなくなる．今回は生成アルゴリズムを簡単にするため，このような制約を設けておく）．

B.2　計算量軽減のための制約

4.3 節で述べた，計算量の爆発を抑えるための制約の詳細を示す．

Constraint 7. **あるイベントノードを根とし，制御ノードのみから構成されるすべてのグラフに含まれるポイントは，そのイベントの前のイベント（**EE **エッジを逆に辿ったイベント）よりも以前の時刻にマッピングされてはならない**

仮にこのようなマッピングが起こるなら，そのノード（制御ノード）はそのイベントの前のイベントとの関係で記述されたほうがよいと考えられる．このような制約をおくこと

で $\{waiting\ point\}$ に加えるポイントの数が大幅に減る.

Constraint 8. **あるエッジのメトリックが負かそうでないかは実行開始時にわかり，実行中変わらない**

これは Constraint 4 が満たされれば自動的に満たされる制約であるが，仮に Constraint 4 の制約を取り除いたとしても，計算量の軽減のために加えた方が望ましいので，独立に定めておく.

付 録C　アルゴリズムの詳細

4 章で述べた，EventGraph に基づく運動生成アルゴリズムの詳細について記載する．まず，C.1 節で生成に必要な追加要素について述べ，C.2 節でいくつかの部分アルゴリズムを，C.3 節で中心となるアルゴリズムを，若干の説明とともに述べる．

C.1　メトリックとプリミティブパラメタの推定値

Fig.C.1 に示すような EventGraph の部分を考えたとき（エッジ e_1 のメトリック $metric$ は正），制御ノード C_1 の，先頭のポイント p_2 が生起する時間 t は，

$$p_2.t \leftarrow E_1.p_1.t + e_1.metric - C_1.primitive.\Delta t_1 \tag{C.1}$$

によって決められるが（E_1 はイベントノード，p_1 はそのポイント），$e_1.metric$ はポイント p_1 が生起されるまで値が決まらない．しかし，p_2 は p_1 よりも前に位置する可能性もあるため，p_1 が生起される前に，$e_1.metric$ のおおよその値（推定値）を知る必要がある．そこで，過去の実行からこの推定値を見積り，保持しておくことにする．具体的には，$est\text{-}metric$ ($\in \mathbb{R}$) にメトリックの推定値が代入されているとする．推定方法は，過去の平均とした．

また，プリミティブの分散 $\Delta t_1, \Delta t_2$ についても同様で，過去の平均値を推定値として保持させている*.

C.2　サブアルゴリズム

C.2.1　導入

この節では，いくつかの補助的なアルゴリズムを定義する．これらはポイント，エッジ，イベントノード，制御ノードごとに定義されるが，共通の目的や役割をもつものは，同じ名前にしてある：

Call	生起する
Update	生起時刻の更新（エッジを辿って伝搬させる）
Check	生起するか（Call を実行するか）を判断する
Open	ポイントを展開するときに実行される

ポイントを展開するとは，EE エッジ†がアクティブになったときに，生起の判定対象ポイントを $\{waiting\ point\}$ に追加することを意味する．

これらのアルゴリズムはポイントなどの各要素に対して定義され，$Point$.Call($args$) などのようにして使用される．ここで，$args$ は引数 (Input) である．返り値は Output として示してある．また，これらのアルゴリズム（関数）は，相互参照されることもある．

---

* なお，5 章の実験では，イベントと制御ノードや制御ノードとほかの制御ノードを結ぶメトリック，あるいはプリミティブのパラメタは定数しか扱っていない．

† イベントノードのポイントとイベントノードのポイントをつなぐエッジのことである．

同一のアルゴリズムが異なった状況で使われることがある．それらを区別する必要があるとき，フラグを引数に含める．フラグ集合は，以下によって定義される．

$$\mathsf{Flag} = \{\text{fkNone, fkEnforced, fkInit}\} \tag{C.2}$$

C.2.2 ポイント

ポイント $Point$ ($\in$ Point) に対して，Algorithm 2, 3, 4, 5 を定義する．なお，例えば $Point$ の t 要素を参照するとき $Point.t$ と書くべきであるが，これらのアルゴリズムは $Point$ に対して定義されているので，$Point.$ を省略して t と書く．ほかの $Point$ の要素についても同様であり，さらに $Point$ に対して定義されたほかのアルゴリズムを呼び出す場合も同じように省略する（例えば $Point$.UpdateF($time$, fkNone) $\rightarrow$ UpdateF($time$, fkNone)）．これ以降のエッジ，イベント/制御ノードの項でも同じように省略する．

Algorithm 2: $Point$.Check

```
Input: time ∈ ℝ
Output: {true, false}
    if Container(Point) ∈ C-Node then
      return time ≥ t
    else /* Container(Point) ∈ E-Node */
      return Container(Point).Check(time)
    end if
```

ここで，$/*\cdots*/$ はコメントを表す．Container($Point$) は $Point$ が含まれているイベントノードまたは制御ノードを返す関数である．また，$\simeq$ 演算子は，4.1.4 項で定義した更新条件を満たさないとき真を返す．すなわち，

$$[t_1 \simeq t_2] \equiv \left[|t_1 - t_2| < \frac{\varepsilon}{2}|t_1 + t_2|\right]. \tag{C.3}$$

これらの定義は以下同様である．

Algorithm 3: $Point$.Call

```
Input: time ∈ ℝ, flag ∈ {fkNone, fkInit}
Output: φ
    if ¬UpdateF(time, fkNone) then
      t ← time
    end if
    if flag ≠ fkInit then
      erase Point from {waiting point}
    end if
    if Container(Point) ∈ C-Node then
      Container(Point).Call(Point)
    else /* Container(Point) ∈ E-Node */
      Container(Point).Call()
    end if
    for all Edge in GetToEdge(Point) do
      if Edge.metric is not negative delay ∧ Edge is not ee-edge then
        Edge.Call(time, fkNone)
      end if
    end for
    for all Edge in GetFromEdge(Point) do
      if Edge.metric is negative delay ∧ Container(Edge.from-point) ∉ E-Node
          ∧¬IsCtrlTailPoint(Edge.from-point) then
```

```
      {waiting point} ← {waiting point} ∪ {Edge.from-point}
    end if
  end for
```

ここで，IsCtrlTailPoint($Point$) は $Point$ が制御ノードを構成するポイントで，かつ制御点の最後に位置するとき真，そうでないときに偽を返す関数である．

GetToEdge($Point$) は $Point$ を *from-point* としてもつすべてのエッジを集合として返す関数，GetFromEdge($Point$) は $Point$ を *to-point* としてもつすべてのエッジを集合として返す関数である．なお，付録 B の Constraint 6 から，GetFromEdge($Point$) が返す集合の要素数 (|GetFromEdge($Point$)|) は 0 または 1 である．

Algorithm 4: $Point$.Update

```
Input: time ∈ ℝ, flag ∈ {fkNone, fkEnforced}
Output: {true, false}
  if Container(Point) ∈ E-Node then
    return Point.UpdateF(time, flag)
  else /* Container(Point) ∈ C-Node */
    if Point.UpdateF(time, flag) then
      Container(Point).Update(flag)
      return true
    else
      return false
    end if
  end if
```

Algorithm 5: $Point$.UpdateF

```
Input: time ∈ ℝ, flag ∈ {fkNone, fkEnforced}
Output: {true, false}
  if flag = fkEnforced ∨ ¬(t ≃ time) then
    t ← time
    for all Edge in GetToEdge(Point) do
      if Container(Edge.to-point) ∉ E-Node then
        Edge.Update(t, flag)
      end if
    end for
    return true
  else
    return false
  end if
```

C.2.3 エッジ

エッジ $Edge$ (∈ Edge) に対して，Algorithm 6, 7, 8 を定義する．

Algorithm 6: $Edge$.Call

```
Input: time ∈ ℝ, flag ∈ {fkNone, fkEnforced}
Output: φ
  var r_0 ∈ ℝ
  r_0 ← metric.m_0.eval(from-point.t)
  var new-m ∈ ℝ
  new-m ← r_0 + metric.m.Evaluate(from-point.t)
  var new-t_e ∈ ℝ
  new-t_e ← from-point.t + new-m
  est-metric の更新
```

```
    if Edge is negative delay then
      return
    end if
    to-point.Update(new-t_e, flag)
    if ¬IsCtrlTailPoint(to-point) then
      {waiting point} ← {waiting point} ∪ {to-point}
    end if
    if Container(to-point) ∉ E-Node ∧ (new-t_e < from-point.t ∨ new-t_e ≃ from-point.t) then
      if to-point ∈ {waiting point} then
        to-point.Call(from-point.t, fkNone)
      end if
    end if
```

Algorithm 7: *Edge*.Update

```
Input: time ∈ ℝ, flag ∈ {fkNone, fkEnforced}
Output: {true, false}
    var new-t_e ∈ ℝ
    new-t_e ← from-point.t + est-metric
    return to-point.Update(new-t_e, flag)
```

Algorithm 8: *Edge*.CalcRest

```
Input: time ∈ ℝ
Output: φ
    if Edge is ee-edge then
      var r_0 ∈ ℝ
      r_0 ← metric.m_0.eval(from-point.t)
      var r ∈ ℝ
      r ← metric.m.eval(time)
      var D ∈ ℝ
      D ← (time − from-point.t − r_0 + r)^2 + 4r_0 r
      D が負ならエラー
      var new-t_e ∈ ℝ
      new-t_e = 1/2 ((time + from-point.t + r_0 + r) + √D)
      to-point.Update(new-t_e, fkNone)
    end if
```

C.2.4 イベントノード

イベントノード *E-Node* (∈ E-Node) に対して，Algorithm 9, 10, 11 を定義する．

Algorithm 9: *E-Node*.Check

```
Input: time
Output: {true, false}
    return event-cond(time)
```

Algorithm 10: *E-Node*.Call

```
Input: φ
Output: φ
    for all Edge in GetFromEdge(point) do
      if Edge ∈ {active edge} then
        erase Edge from {active edge}
      end if
    end for
```

```
  for all Edge in GetToEdge(point) do
    if Container(Edge.to-point) ∈ E-Node then
      {active edge} ← {active edge} ∪ {Edge}
      Edge.Call(point.t, fkEnforced)
      Container(Edge.to-point).Open()
    end if
  end for
```

Algorithm 11: *E-Node*.Open

```
Input: φ
Output: φ
  重複して E-Node.Open が実行されないかチェック
  for all Edge in GetToEdge(point) do
    if Container(Edge.to-point) ∈ C-Node then
      Container(Edge.to-point).Open()
    end if
  end for
```

C.2.5 制御ノード

制御ノード *C-Node* (∈ C-Node) に対して，Algorithm 12, 13, 14, 15 を定義する．

Algorithm 12: *C-Node*.Call

```
Input: p ∈ {point}
Output: φ
  if GetPrev(p) = NullPt then /* starting output */
    {active c-node} ← {active c-node} ∪ {C-Node}
    Primitive を初期化（パラメタの決定）
  end if
  Update(fkNone)
  if GetNext(p) ≠ NullPt then
    {waiting point} ← {waiting point} ∪ {GetNext(p)}
  else /* finishing output */
    erase C-Node from {active c-node}
  end if
```

ここで，*NullPt* は EventGraph に含まれるどのポイントでないことを表す特殊なポイントである．また，GetPrev(*Point*) は制御ノードを構成するポイント *Point* に対して，ひとつ前の制御点を返す関数で，*Point* が制御点の先頭なら *NullPt* を返す．反対に，GetNext(*Point*) はひとつ後の制御点を返す関数であり，*Point* が制御点の最後なら *NullPt* を返す．

次に述べる *C-Node*.Update はほかの要素に対して定義された Update と同様に，イベントの生起予測時刻が変動し，更新が伝搬して行く中で実行されるアルゴリズムであるが，制御点の間での計算が特殊である．ある制御ノードを構成する制御点 (Point) とそれらに接続されているエッジ（制御点を従属ポイントとするエッジ）の関係は，大きく分けて Fig.C.2 に示したような種類がある．Fig.C.2(a) や Fig.C.2(b) のように一端がエッジに接続されている場合は，そこから生起時刻が伝搬していく．Fig.C.2(c) のように端点でないポイントがエッジに接続されている場合は，そこを中心にして両方向に伝搬する．Fig.C.2(d) のように一部の制御点の両端がエッジに接続されている場合は，区間幅（この場合 Δt_1 と Δt_2）に比例して，その間の生起時刻を計算する．具体的には，

$$p_2.t \leftarrow \frac{\Delta t_1 p_2.t + \Delta t_2 p_1.t}{\Delta t_1 + \Delta t_2} \tag{C.4}$$

となる．このアルゴリズムを一般化したものが，Algorithm 13 である．

Algorithm 13: *C-Node*.Update

```
Input: flag ∈ Flag
Output: φ
    var p_1 ∈ {point} ∪ NullPt
    var p_2 ∈ {point} ∪ NullPt
    var t ∈ ℝ
    if Current(C-Node) = NullPt then
      p_2 ← point_1 /*begining point*/
    else
      p_2 ← GetNext(Current(C-Node))
    end if
    while p_2 ≠ NullPt do
      while p_2 ≠ NullPt ∧ |GetFromEdge(p_2)| ≠ 0 do
        p_2 ← GetNext(p_2)
      end while
      if p_2 = NullPt then
        return
      end if
      p_1 ← GetPrev(p_2)
      while p_2 ≠ NullPt ∧ |GetFromEdge(p_2)| = 0 do
        p_2 ← GetNext(p_2)
      end while
      if p_1 = NullPt ∧ p_2 = NullPt then
        error
      end if
      if p_2 = NullPt then
        t ← p_1.t
        while p_1 ≠ point_{|{point}|−1} do
          t ← t + GetInterval(Primitive, p_1)
          p_1 ← GetNext(p_1)
          p_1.UpdateF(t, flag)
        end while
        return
      else if p_1 = NullPt then
        t ← p_2.t
        p_1 ← GetPrev(p_2)
        while p_1 ≠ NullPt do
          t ← t − GetInterval(Primitive, p_1)
          p_1.UpdateF(t, flag)
          p_1 ← GetPrev(p_1)
        end while
      else /* p_1 ≠ NullPt ∧ p_2 ≠ NullPt */
        var sum ∈ ℝ
        var sum_r ∈ ℝ
        sum ← Σ_p GetInterval(Primitive, p)
        sum_r ← p_2.t − p_1.t
        t ← p_1.t
        while p_1 ≠ p_2 do
          t ← t + (sum_r / sum) GetInterval(Primitive, p_1)
          p_1 ← GetNext(p_1)
          p_1.UpdateF(t, flag)
        end while
      end if
      p_2 ← GetNext(p_2)
```

```
    end while
```

ここで，Current($C\text{-}Node$) は，制御ノード $C\text{-}Node$ で現在出力している区間の，両端のポイントのうち，既に生起されたポイントを返す関数である．制御ノードが出力中でない場合[‡]は，$NullPt$ を返す．また，GetInterval($Primitive, Point$) は，プリミティブ $Primitive$ の制御点 $Point$ から始まる区間の区間幅を返す関数である．制御ノードが出力中でない場合，C.1 節で述べた推定値を返す．

Algorithm 14: $C\text{-}Node$.Open

```
Input: φ
Output: φ
    重複して C-Node.Open が実行されないかチェック
    if IsStartingPoint(point_1) then
      {waiting point} ← {waiting point} ∪ {point_1}
    end if
    for i ← 0 to |{point}| − 1 do
      for all Edge in GetToEdge(point_i) do
        if Container(Edge.to-point) ∈ C-Node then
          Container(Edge.to-point).Open()
        end if
      end for
    end for
```

Algorithm 15: $C\text{-}Node$.CalcOutput

```
Input: time ∈ ℝ
Output: φ
    Out ← Primitive.f(time)
```

なお，$O_p \in$ OutputPortSet において，$O_p.out = \sum O_p.in$ でアクチュエータに対する出力が決定される．今回の実装ではこのように和を用いているが，例えば「滑らかにつなぐ」場合は，何らかのフィルタ処理を加えることが考えられる．

C.3 メインアルゴリズム

制御信号生成アルゴリズムは，Run(Algorithm 16) と StepBy(Algorithm 17) から構成される．Run は実行を開始するときに一度だけ呼び出されるアルゴリズム，StepBy($diff$) は実行中に EventGraph の内部時刻を $diff$ だけ進めるときに呼び出されるアルゴリズムである．

[‡] $C\text{-}Node \notin \{active\ c\text{-}node\}$

Algorithm 16: *EventGraph*.Run

```
Input: φ
Output: φ
    {waiting point} ← φ
    {active edge} ← φ
    {active c-node} ← φ

    time ← 0
    start-node.point.Update(time, fkEnforced)
    start-node.Open()
    start-node.point.Call(time, fkInit)
    StepBy(0)
```

Algorithm 17: *EventGraph*.StepBy

```
Input: diff ∈ ℝ
Output: {true, false}
    time ← time + diff
    for all Point in {waiting point} do
      if Point.Check(time) then
        Point.Call(time, fkNone)
      end if
    end for
    for all Edge in {active edge} do
      Edge.CalcRest(time)
    end for
    for all C-Node in {active c-node} do
      C-Node.CalcOutput(time)
    end for
    if {waiting point} = φ ∧ {active c-node} = φ ∧ {active edge} = φ then
      return false
    else
      return true
    end if
```

付 録D　EventGraphの構造を固定した最適化

ここでは，5.2節で述べた最適化手法の詳細について述べる．具体的な最適化手法はMGG*とSPX†で構成される遺伝アルゴリズム (GA) を用いる．このときGAの1個体にひとつのEventGraphを割り当てる．

以下，D.1節ではEventGraphに対する演算を定義し，D.2節でMGGとSPXのアルゴリズムについて述べる．

D.1　EventGraphに対する演算

SPXの適用において，EventGraph間の加算・減算，及びEventGraphと実数の積算が定義されている必要がある．また，ひとつのEventGraphに対する評価関数が定められなければならない．

EventGraph間の演算は，最適化対象となっているパラメタごとの演算として定義する．5.2節で述べた通り，最適化対象のパラメタはイベントの予測を行う役割ではないエッジ‡のメトリック $Edge.metric.m_0(\in \mathbb{R})$ と，プリミティブのパラメタ Δt_b, Δt_f, $peak(\in \mathbb{R})$ である（逆に，これら以外の最適化対象とならない構造や要素は，ふたつのEventGraph間で等しい）．EventGraph間の演算（加算と減算）の結果は，これらの要素それぞれについて計算したものである．このとき非最適化対象は，演算対象と同じである．

同様に，EventGraphと実数の積算の結果は，最適化対象のパラメタそれぞれに対して実数を掛けたものとする．

EventGraphに対する評価は，EventGraphを用いてタスクを行わせ，その結果に対して5.3.4項で定義した評価関数を適用することで行う．

D.2　SPXとMGG

Algorithm18, 19に，EventGraphに対するMGGとSPXのアルゴリズムを示す．なおMGGのアルゴリズムは，樋口らによる調整[14]を含んでいる．

Algorithm 18: *EventGraph*.MGG

$N_\mathrm{group}(\in \mathbb{N})$：集団サイズ
$N_\mathrm{child}(\in \mathbb{N})$：交叉が生成する子個体の数
$N_\mathrm{parent}(\in \mathbb{N})$：親個体の数
`var` $\{pool_i\} \subset$ EventGraph, $i = 1, \ldots, N_\mathrm{group} + N_\mathrm{child}$
`var` $\{evaluation_i\} \subset \mathbb{R}$, $i = 1, \ldots, N_\mathrm{group} + N_\mathrm{child}$
`var` $\{parent\} \subset \{pool_i\}$, $\{parent\} \leftarrow \phi$
`for` $i \leftarrow 1$ `to` N_group `do`
  $pool_i \leftarrow$ (ランダムに生成した *EventGraph*)
  $evaluation_i \leftarrow$ Evaluate($pool_i$)
`end for`
`while` ¬終了条件 `do`
  $\{parent\} \leftarrow$ ($\{pool_i \mid i = 1, \ldots, N_\mathrm{group}\}$ から N_parent 個ランダムに選択)

* 世代交代モデル，Minimal Generation Gap[13]
† 交叉オペレーション，シンプレックス交叉[14]
‡ $Edge \in \{e \mid e \in EventGraph.\mathsf{Edge},\ e.to\text{-}point \neq E\text{-}Node.point \text{ for all } E\text{-}Node \in EventGraph.\mathsf{E\text{-}Node}\}$

```
    for i ← 1 to N_child do
      pool_{N_group+i} ← SPX({pool_i}, {parent})
      適当な確率のもとで pool_{N_group+i} を突然変異させる
      evaluation_{N_group+i} ← Evaluate(pool_{N_group+i})
    end for
    {selection} ← ({parent} から 2 個体非復元抽出)
    {pool_i} の 2 個体非復元抽出されたインデクスのところに, {pool_i | i = N_group+1, ..., N_group+
    N_child} ∪ {selection} から最良個体及び適応度のランキングによるルーレット選択で選んだ 1
    個体を挿入
  end while
```

Algorithm 19: *EventGraph*.SimplexCrossover : SPX

```
Input: {pool} ⊂ EventGraph, {parent} ⊂ {pool}
Output: child ∈ EventGraph
  var n ∈ ℝ
  n ← |{parent}|
  α = √(n + 1)
  var g ∈ EventGraph
  g ← (1/n) Σ_{i=1}^{n} parent_i
  var c ∈ EventGraph
  c ← ZeroEG
  var p_0, p_1 ∈ EventGraph
  p_0 ← g + α(pool_1 − g)
  var r ∈ ℝ
  for k ← 2 to n do
    p_1 ← g + α(pool_k − g)
    r ← (u(0, 1))^{1/k}
    c ← r(p_0 − p_1 + c)
    p_0 ← p_1
  end for
  return p_1 + c
```

ここで *ZeroEG* は，$\{pool\}$ と同じ非最適化対象の構造と要素をもち（すべての $pool(\in \{pool\})$ についてこれらは等しい），かつ最適化対象の要素がゼロである EventGraph を表す．また，$u(0,1)$ は区間 [0,1] の一様分布乱数を返す関数である．

初期世代の生成では，人手で大雑把に調節した値（もちろん，これではタスクを達成できない）を一様乱数で分散させて生成した．また，突然変異も同様の方法で生成した．

これらの実行においては，集団サイズ $N_{\mathrm{group}} = 20$，交叉が生成する子個体の数 $N_{\mathrm{child}} = 8$，親個体の数 $N_{\mathrm{parent}} = 11$ とした．また突然変異を起こす確率は 1%とした．

付 録E　シミュレーションの詳細

5 章の実験で行ったシミュレーションの詳細を述べる．E.1 節では多自由度ロボットの構成を，E.2 節では多自由度ロボットの順動力学計算アルゴリズムを，E.3 節ではトランポリンのモデルを，それぞれ述べる．

なお，シミュレータの作成にあたって，文献 [15] を参考にした．

E.1　多自由度ロボットの構成の詳細

多自由度ロボットの構成は Fig.5.1 に示した通りである．リンクの形状はどのリンクも中身が詰まった円筒であり，リンク $L_j\,(j=0,1,2,3)$ の質量 m_j，長さ $length_j$，直径 $diameter_j$ を Table E.1 に示す．このとき及び基準姿勢における，重心まわりの慣性行列 $\bar{\mathbf{I}}_j$ は，

$$\bar{\mathbf{I}}_j = \begin{bmatrix} \frac{1}{12}m_j(\frac{3}{4}diameter_j^2 + length_j^2) & 0 & 0 \\ 0 & \frac{1}{12}m_j(\frac{3}{4}diameter_j^2 + length_j^2) & 0 \\ 0 & 0 & \frac{1}{8}m_j \cdot diameter_j^2 \end{bmatrix} \tag{E.1}$$

によって求められる．

関節 $J_j\,(j=1,2,3)$ は，角度が可動範囲にあるときは慣性力 $\mu_j\ddot{q}_j$ のみ受け，可動範囲を越えたときはそれにバネ (K_{J_j})・ダンパ (D_{J_j}) による拘束が加わるというモデルにした．各関節の設定値を Table E.2 に示す．

E.2　順動力学計算アルゴリズム

順動力学の計算には，Featherstone のアルゴリズム [12] を用いた．このアルゴリズムは，単位ベクトル法の計算量が $\mathcal{O}(N^3)$ のオーダであるのに対し，$\mathcal{O}(N)$ の計算量で，非常に高速である*．ただし，Featherstone の文献 [12] で述べられているアルゴリズムは外力や関節の慣性力を含んだ式になっていないため，以下で拡張を行う．なお記号の使い方，一部の式の導出に関して文献 [15] を参考にした．

以下で定義する物理量は，特に断らない限りワールド座標系における定義である．リンク L_j の座標を $\mathbf{p}_j(\in \mathbb{R}^{3\times1})$，その関節 J_j の単位回転軸ベクトルを $\mathbf{a}_j(\in \mathbb{R}^{3\times1})$，姿勢行列を $\mathbf{R}_j(\in \mathbb{R}^{3\times3})$ とし，空間速度ベクトルを $\boldsymbol{\xi}_j(\in \mathbb{R}^{6\times1})$（要素 1-3 が速度 $\mathbf{v}_{oj}$，要素 4-6 が角速度 $\boldsymbol{\omega}_j$ である），その微分である空間加速度ベクトルを $\dot{\boldsymbol{\xi}}_j(\in \mathbb{R}^{6\times1})$ とする．つまり，

$$\boldsymbol{\xi}_j = \begin{bmatrix} \mathbf{v}_{oj} \\ \boldsymbol{\omega}_j \end{bmatrix} \tag{E.2}$$

$$\dot{\boldsymbol{\xi}}_j = \begin{bmatrix} \dot{\mathbf{v}}_{oj} \\ \dot{\boldsymbol{\omega}}_j \end{bmatrix} \tag{E.3}$$

である．また，関節の空間加速度ベクトルを $\mathbf{s}_j(\in \mathbb{R}^{6\times1})$ とし，その微分を $\dot{\mathbf{s}}_j(\in \mathbb{R}^{6\times1})$ と

---

* Featherstone のアルゴリズムがオイラー法よりも速くなるのは，$N \geq 17$ のときであるという [16]．しかし，両者を実装し比較した結果，Featherstone のアルゴリズムの方が高速であった．

すると，これらは，

$$\mathbf{s}_j = \begin{bmatrix} \mathbf{p}_j \times \mathbf{a}_j \\ \mathbf{a}_j \end{bmatrix} \tag{E.4}$$

$$\dot{\mathbf{s}}_j = \begin{bmatrix} \widehat{\boldsymbol{\omega}}_{j-1} & \widehat{\mathbf{v}}_{oj-1} \\ \mathbf{0}^{(3)} & \widehat{\boldsymbol{\omega}}_{j-1} \end{bmatrix} \mathbf{s}_j \tag{E.5}$$

で与えられる（$\mathbf{0}^{(3)}$ は $\mathbb{R}^{3\times3}$ のゼロ行列）．ここでウェッジ $\wedge$ はベクトルの外積オペレータ†を表す．このとき，隣接する 2 つのリンク間に

$$\dot{\boldsymbol{\xi}}_{j+1} = \dot{\boldsymbol{\xi}}_j + \dot{\mathbf{s}}_{j+1}\dot{q}_{j+1} + \mathbf{s}_{j+1}\ddot{q}_{j+1} \tag{E.6}$$

が成立する．リンク L_j に加わる力の内訳は，両端のリンクのうち，ルートリンク L_0 に近い方のリンクとの関節からの力 $\mathbf{F}_j$（$\in \mathbb{R}^{6\times1}$，要素 4-6 はトルク），その反対のリンクからの力 $-\mathbf{F}_{j+1}$（$\in \mathbb{R}^{6\times1}$，要素 4-6 はトルク），及び外力 $\mathbf{F}^{\mathrm{E}}_j$（$\in \mathbb{R}^{6\times1}$，要素 4-6 はトルク）である．リンク L_j のワールド座標系における重心（$\mathbf{c}_j(\in \mathbb{R}^{3\times1})$ とする）まわりの慣性行列 $\mathbf{I}_j = \mathbf{R}_j\bar{\mathbf{I}}_j\mathbf{R}_j^\top(\in \mathbb{R}^{3\times3})$ に対して $\mathbf{I}^{\mathrm{S}}_j(\in \mathbb{R}^{6\times6})$ を

$$\mathbf{I}^{\mathrm{S}}_j = \begin{bmatrix} m_j\mathbf{1}^{(3)} & -m_j\widehat{\mathbf{c}}_j \\ m_j\widehat{\mathbf{c}}_j & \mathbf{I}_j - m_j\widehat{\mathbf{c}}_j\widehat{\mathbf{c}}_j \end{bmatrix} \tag{E.7}$$

と定義する．ただし $\mathbf{1}^{(3)}$ は $\mathbb{R}^{3\times3}$ の単位行列，$\widehat{\mathbf{c}}_j$ は $\mathbf{c}_j$ の外積オペレータである．このとき，リンク L_j の運動方程式は，

$$\mathbf{F}_j - \mathbf{F}_{j+1} + \mathbf{F}^{\mathrm{E}}_j = \mathbf{I}^{\mathrm{S}}_j\dot{\boldsymbol{\xi}}_j + \boldsymbol{\xi}_j \times \mathbf{I}^{\mathrm{S}}_j\boldsymbol{\xi}_j \tag{E.8}$$

or

$$\mathbf{F}_j = \mathbf{F}_{j+1} - \mathbf{F}^{\mathrm{E}}_j + \mathbf{I}^{\mathrm{S}}_j\dot{\boldsymbol{\xi}}_j + \boldsymbol{\xi}_j \times \mathbf{I}^{\mathrm{S}}_j\boldsymbol{\xi}_j \tag{E.9}$$

と表される．ここで，

$$\boldsymbol{\xi}_j \times \ = \begin{bmatrix} \mathbf{v}_{oj} \\ \boldsymbol{\omega}_j \end{bmatrix} \times = \begin{bmatrix} \widehat{\boldsymbol{\omega}}_j & \mathbf{0}^{(3)} \\ \widehat{\mathbf{v}}_{oj} & \widehat{\boldsymbol{\omega}}_j \end{bmatrix} \tag{E.10}$$

と定義した．また，関節トルク u_j と関節に働く力の関係式は

$$u_j - \mu_j\ddot{q}_j = \mathbf{s}_j^\top\mathbf{F}_j \tag{E.11}$$

である．

ここで

$$\mathbf{F}_j = \mathbf{I}^{\mathrm{A}}_j\dot{\boldsymbol{\xi}}_j + \mathbf{b}_j \tag{E.12}$$

---

† ある列ベクトル $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} (\in \mathbb{R}^{3\times1})$ に対して，外積オペレータは $\widehat{\mathbf{x}} = \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix}$ で与えられる．

とおいたとき，$\mathbf{I}_j^{\mathrm{A}}$ を Articulated-Body Inertia と呼ぶ．$\mathbf{b}_j$ はバイアス項 (Bias Force) である．式 (E.9) の右辺に式 (E.12)，式 (E.6) を順に代入し，式 (E.12) と比較することで，

$$\mathbf{I}_j^{\mathrm{A}} = \mathbf{I}_{j+1}^{\mathrm{A}} + \mathbf{I}_j^{\mathrm{S}} - \frac{\mathbf{I}_{j+1}^{\mathrm{A}}\mathbf{s}_{j+1}\mathbf{s}_{j+1}^{\top}\mathbf{I}_{j+1}^{\mathrm{A}}}{\mathbf{s}_{j+1}^{\top}\mathbf{I}_{j+1}^{\mathrm{A}}\mathbf{s}_{j+1} + \mu_{j+1}} \tag{E.13}$$

$$\begin{aligned}\mathbf{b}_j =& \mathbf{b}_{j+1} - \mathbf{F}_j^{\mathrm{E}} + \mathbf{I}_{j+1}^{\mathrm{A}}\dot{\mathbf{s}}_{j+1}\dot{q}_{j+1} + \boldsymbol{\xi}_j \times \mathbf{I}_j^{\mathrm{S}}\boldsymbol{\xi}_j \\ &+ \frac{\mathbf{I}_{j+1}^{\mathrm{A}}\mathbf{s}_{j+1}\{u_{j+1} - \mathbf{s}_{j+1}^{\top}(\mathbf{I}_{j+1}^{\mathrm{A}}\dot{\mathbf{s}}_{j+1}\dot{q}_{j+1} + \mathbf{b}_{j+1})\}}{\mathbf{s}_{j+1}^{\top}\mathbf{I}_{j+1}^{\mathrm{A}}\mathbf{s}_{j+1} + \mu_{j+1}}\end{aligned} \tag{E.14}$$

の漸化式を得る．また，関節角速度については式 (E.11) に式 (E.12), (E.6) を代入することによって，漸化式

$$\ddot{q}_j = \frac{u_j - \mathbf{s}_j^{\top}(\mathbf{I}_j^{\mathrm{A}}(\dot{\boldsymbol{\xi}}_{j-1} + \dot{\mathbf{s}}_j\dot{q}_j) + \mathbf{b}_j)}{\mathbf{s}_j^{\top}\mathbf{I}_j^{\mathrm{A}}\mathbf{s}_j + \mu_j} \tag{E.15}$$

が得られる．漸化式 (E.13)，(E.14) に基づいて $\mathbf{I}_j^{\mathrm{A}}$，$\mathbf{b}_j$ を末端リンクから計算し，次いでルートリンクから式 (E.6) を用いて空間加速度を，式 (E.15) を用いて関節角速度を計算する手法が，Featherstone の順動力学計算アルゴリズムである [12]．なお末端リンクについては $\mathbf{F}_{j+1} = \mathbf{0}$ だから，

$$\mathbf{I}_j^{\mathrm{A}} = \mathbf{I}_j^{\mathrm{S}} \tag{E.16}$$

$$\mathbf{b}_j = \boldsymbol{\xi}_j \times \mathbf{I}_j^{\mathrm{S}}\boldsymbol{\xi}_j - \mathbf{F}_j^{\mathrm{E}} \tag{E.17}$$

とし，またルートリンクに関しては，$\mathbf{F}_j = \mathbf{0}$ だから，

$$\dot{\boldsymbol{\xi}}_j = -(\mathbf{I}_j^{\mathrm{A}})^{-1}\mathbf{b}_j \tag{E.18}$$

として計算する．

なお，積分計算には 4 次の Runge-Kutta 法を用いている．

実装の妥当性は，単位ベクトル法を用いた動力学シミュレータを Featherstone のアルゴリズムによるものとは独立に構成し，結果を比較することで確かめた．

E.3 トランポリンのモデル

E.3.1 トランポリンの順動力学

トランポリンのモデルとして，まず，質量 M_{f} の剛体がバネ K_{f} とダンパ D_{f} で地面に接続されているとする．このとき Fig.E.1 のように，ロボットとトランポリンが接触している状況を考える．トランポリンに加わる力のうち，重力やバネ，ダンパによるものは求められるが，ロボットから受ける力（この反作用力が $\mathbf{F}$，つまりロボットが受ける力である）は求められない．これはトランポリンとロボットの間の作用・反作用の関係がモデル化できていないためであり，何らかの仮定（例えば接触中はトランポリンとロボットの接触部位は連動して動くなど）を設定しなければトランポリンの運動を記述できない．

しかし $\mathbf{F}$ が求まれば，トランポリンの運動方程式は，トランポリンの座標を $\mathbf{p}_{\mathrm{tr}}$ として，

$$D_{\mathrm{f}}\dot{\mathbf{p}}_{\mathrm{tr}} + K_{\mathrm{f}}\mathbf{p}_{\mathrm{tr}} - \mathbf{F} + M_{\mathrm{f}}\mathbf{g} = M_{\mathrm{f}}\ddot{\mathbf{p}}_{\mathrm{tr}} \tag{E.19}$$

であるから（$\mathbf{g}$ は重力加速度），トランポリンの運動が記述できる．よってロボットとトランポリンから成る系をシミュレートするには，トランポリンの反作用モデルが必要である．本節では，このモデルについて述べる．

E.3.2　トランポリンの反作用モデル

多自由度ロボットのリンクはすべて円筒であるから，ひとつの円筒とトランポリンの間で作用・反作用力が記述できればよい．個々のリンクごとに力を計算すれば前節で述べた順動力学のアルゴリズムを適用でき，トランポリンについても，外力を個々のリンクに加わる力の反作用力の総和として求めることによって，運動をシミュレートできる．

現実のトランポリンを考えてみると，トランポリンに力を加えることによって表面の形状が変化し，それがもとに戻ろうとして力が発生する．このことから，トランポリンの表面と円筒の接触状態，及びトランポリン表面の形状から作用・反作用力を計算できると考えられる．しかしこの解析は困難であり，有限要素法などで数値解析することも考えられるが計算に時間がかかる．そこでトランポリンと円筒の位置関係のみから作用・反作用力を計算するモデルを考案した．

以下では簡単のために，円筒の断面積は十分に小さいと仮定する．トランポリンと円筒の位置関係を Fig.E.2 に示す．ここで $\mathbf{p}_1(\in \mathbb{R}^{3\times1})$ は円筒の下端座標，$\mathbf{p}_2(\in \mathbb{R}^{3\times1})$ は円筒の上端座標 $(\mathbf{p}_2.z \geq \mathbf{p}_1.z)$，$\mathbf{p}_x(\in \mathbb{R}^{3\times1})$ は円筒とトランポリン表面が交わる座標である．円筒とトランポリンの位置関係は，この図に示したほかに，完全に離れている場合と，完全に沈み込んでいる場合がある．完全に離れている場合は作用が発生しないので考えない．完全に沈み込んでいる場合は，$\mathbf{p}_x = \mathbf{p}_2$ であるとする．

トランポリンと円筒との位置関係がこのような関係にあるとき，円筒には上向きに押し上げる力が働く．これは浮力に似ているが，トランポリンは円筒の上側に回り込まないため，$\mathbf{p}_2.z$ が小さくなるほど反力は大きくなる．これは円筒によって変形したトランポリンが復元しようとして発生する力であり，便宜上「浮力項」と呼ぶことにする．これはある瞬間に円筒がトランポリンの表面をどのくらい変化させたかで決まるが，簡単のため，円筒から鉛直上向きに存在する部分のみが変形させられたと考える．

また，円筒の速度（トランポリンに対する相対速度）に対してそれを妨げるような「粘性項」があると考えられる．

以下，トランポリンから受ける力を，浮力項と粘性項に分けて議論する．

浮力項

底面積 dS の微小体積がトランポリンの表面から深さ $depth$ の位置にあるとき，これに働く力（トランポリンの復元力）が鉛直上向きに $dF_{\mathrm{K}} = K_{\mathrm{d}} dS\, depth$ で与えられるとする．このとき，これをトランポリンに沈んでいる円筒について積分すれば，円筒に加わっている力の浮力項 F_{K} が計算できる．円筒の断面積は十分に小さいとしているから，$\mathbf{p}_1$ を原点として $\mathbf{p}_x$ の方向へ χ 軸を取り，$\chi = 0$ から $|\mathbf{p}_x - \mathbf{p}_1|$ の範囲で積分すればよい．このとき，

$$depth = \mathbf{p}_{\mathrm{tr}}.z - \mathbf{p}_1.z - (\mathbf{p}_x.z - \mathbf{p}_1.z)\frac{\chi}{|\mathbf{p}_x - \mathbf{p}_1|} \tag{E.20}$$

$$dS = d\frac{d_{xy}}{|\mathbf{p}_x - \mathbf{p}_1|}d\chi \tag{E.21}$$

where

$$d_{xy} = \sqrt{(\mathbf{p}_x.x - \mathbf{p}_1.x)^2 + (\mathbf{p}_x.y - \mathbf{p}_1.y)^2} \tag{E.22}$$

hence,

$$dF_{\mathrm{K}} = K_{\mathrm{d}} d \frac{d_{xy}}{|\mathbf{p}_x - \mathbf{p}_1|} \left\{ \mathbf{p}_{\mathrm{tr}}.z - \mathbf{p}_1.z - \frac{\mathbf{p}_x.z - \mathbf{p}_1.z}{|\mathbf{p}_x - \mathbf{p}_1|} \chi \right\} d\chi \tag{E.23}$$

となるので（ただし d は円柱の直径 $(diameter)$ である），dF_{K} を χ について積分して，

$$F_{\mathrm{K}} = K_{\mathrm{d}} d d_{xy} \left(\mathbf{p}_{\mathrm{tr}}.z - \frac{1}{2}\mathbf{p}_x.z - \frac{1}{2}\mathbf{p}_1.z \right) \tag{E.24}$$

を得る．なお，$d_{xy} < d$ なら円筒がトランポリンに対して垂直であると考えて，

$$F_{\mathrm{K}} = K_{\mathrm{d}} \pi \left(\frac{d}{2} \right)^2 \left(\mathbf{p}_{\mathrm{tr}}.z - \frac{1}{2}\mathbf{p}_1.z \right) \tag{E.25}$$

とする．

同様に，浮力項に対してトルク $\mathbf{T}_{\mathrm{K}}$ を計算する．微小区間 $d\chi$ に対して，

$$d\mathbf{T}_{\mathrm{K}} = \left(\mathbf{p}_1 + \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|} \chi \right) \times dF_{\mathrm{K}} \mathbf{e}_z \tag{E.26}$$

であるから（$\mathbf{e}_z$ は z 軸方向の単位ベクトル），これを χ について積分して，

$$\begin{aligned} \mathbf{T}_{\mathrm{K}} =& K_{\mathrm{d}} d d_{xy} \left(\frac{1}{2}\mathbf{p}_{\mathrm{tr}}.z - \frac{1}{3}\mathbf{p}_x.z - \frac{1}{6}\mathbf{p}_1.z \right) \mathbf{p}_x \times \mathbf{e}_z \\ &+ K_{\mathrm{d}} d d_{xy} \left(\frac{1}{2}\mathbf{p}_{\mathrm{tr}}.z - \frac{1}{6}\mathbf{p}_x.z - \frac{1}{3}\mathbf{p}_1.z \right) \mathbf{p}_1 \times \mathbf{e}_z \end{aligned} \tag{E.27}$$

を得る．$d_{xy} < d$ の場合，

$$\begin{aligned} \mathbf{T}_{\mathrm{K}} &= \mathbf{p}_1 \times F_{\mathrm{K}} \mathbf{e}_z \\ &= K_{\mathrm{d}} \pi \left(\frac{d}{2} \right)^2 \left(\mathbf{p}_{\mathrm{tr}}.z - \frac{1}{2}\mathbf{p}_1.z \right) \mathbf{p}_1 \times \mathbf{e}_z \end{aligned} \tag{E.28}$$

とする．

粘性項

位置 χ における円筒の微小区間がトランポリンに対する相対速度 $\mathbf{v}_{\mathrm{r}}(\chi)$ で移動しているとき，この部分に対して

$$d\mathbf{F}_{\mathrm{D}} = D_{\mathrm{d}} \mathbf{v}_{\mathrm{r}} dS' \tag{E.29}$$

の粘性項が働くとする．ここで dS' は微小区間を $\mathbf{v}_{\mathrm{r}}(\chi)$ に垂直な方向に投影した面積である．この微小区間において，$\mathbf{p}_x - \mathbf{p}_1$ と $\mathbf{v}_{\mathrm{r}}(\chi)$ のなす角を $\theta(\chi)$ とすると，

$$dS' = d \sin\theta(\chi) d\chi \tag{E.30}$$

となる．また，

$$\mathbf{v}_{\mathrm{r}}(\chi) = \mathbf{v}_{\mathrm{o}} + \boldsymbol{\omega} \times \mathbf{p}_1 - \mathbf{v}_{\mathrm{tr}} + \left(\boldsymbol{\omega} \times \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|} \right) \chi \tag{E.31}$$

$$= \mathbf{v}_{\mathrm{r}}(0) + \left(\boldsymbol{\omega} \times \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|} \right) \chi \tag{E.32}$$

であるから（$\mathbf{v}_{\mathrm{o}}$, $\boldsymbol{\omega}$ は円柱の空間速度），

$$\sin\theta(\chi) = \sqrt{1 - \frac{1}{|\mathbf{v}_{\mathrm{r}}(\chi)|} \left(\mathbf{v}_{\mathrm{r}}(0) \cdot \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|} \right)^2} \tag{E.33}$$

となり，これを用いて $d\mathbf{F}_{\mathrm{D}}$ を χ で積分すればよい．しかしこの計算は繁雑であるから，以下のように線形近似する．

$$\sin\theta(\chi) \sim \frac{\gamma_x - \gamma_1}{|\mathbf{p}_x - \mathbf{p}_1|}\chi + \gamma_1 \tag{E.34}$$

where

$$\gamma_1 \equiv \sin\theta(0) = \sqrt{1 - \frac{1}{|\mathbf{v}_{\mathrm{r}}(0)|} \left(\mathbf{v}_{\mathrm{r}}(0) \cdot \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|} \right)^2} \tag{E.35}$$

$$\gamma_x \equiv \sin\theta(|\mathbf{p}_x - \mathbf{p}_1|) = \sqrt{1 - \frac{1}{|\mathbf{v}_{\mathrm{r}}(|\mathbf{p}_x - \mathbf{p}_1|)|} \left(\mathbf{v}_{\mathrm{r}}(0) \cdot \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|} \right)^2} \tag{E.36}$$

このとき，

$$d\mathbf{F}_{\mathrm{D}} = D_{\mathrm{d}} d \left\{ \mathbf{v}_{\mathrm{r}}(0) + \left(\boldsymbol{\omega} \times \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|} \right) \chi \right\} \left\{ \frac{\gamma_x - \gamma_1}{|\mathbf{p}_x - \mathbf{p}_1|}\chi + \gamma_1 \right\} d\chi \tag{E.37}$$

となって，これを $\chi = 0$ から $|\mathbf{p}_x - \mathbf{p}_1|$ の範囲で積分すると，

$$\begin{aligned} \mathbf{F}_{\mathrm{D}} = D_{\mathrm{d}} d |\mathbf{p}_x - \mathbf{p}_1| \Big\{ & \frac{1}{6}\gamma_1 \big(2\mathbf{v}_{\mathrm{r}}(0) + \mathbf{v}_{\mathrm{r}}(|\mathbf{p}_x - \mathbf{p}_1|) \big) \\ & + \frac{1}{6}\gamma_x \big(\mathbf{v}_{\mathrm{r}}(0) + 2\mathbf{v}_{\mathrm{r}}(|\mathbf{p}_x - \mathbf{p}_1|) \big) \Big\} \end{aligned} \tag{E.38}$$

を得る．このときトルクは，

$$d\mathbf{T}_{\mathrm{D}} = \left(\mathbf{p}_1 + \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|}\chi \right) \times d\mathbf{F}_{\mathrm{D}} \tag{E.39}$$

であるから，これを χ について積分して，

$$\begin{aligned} \mathbf{T}_{\mathrm{D}} = \frac{1}{12} D_{\mathrm{d}} d |\mathbf{p}_x - \mathbf{p}_1| \big\{ & (3\gamma_1 + \gamma_x)\mathbf{p}_1 \times \mathbf{v}_{\mathrm{r}}(0) + (\gamma_1 + \gamma_x)\mathbf{p}_1 \times \mathbf{v}_{\mathrm{r}}(|\mathbf{p}_x - \mathbf{p}_1|) \\ & + (\gamma_1 + \gamma_x)\mathbf{p}_x \times \mathbf{v}_{\mathrm{r}}(0) + (\gamma_1 + 3\gamma_x)\mathbf{p}_x \times \mathbf{v}_{\mathrm{r}}(|\mathbf{p}_x - \mathbf{p}_1|) \big\} \end{aligned} \tag{E.40}$$

となる．

さて，浮力項については，トランポリンの表面の性質から鉛直方向は沈み込んでいる（つまり水のように回り込まない）として計算したが，粘性項については考慮せずに計算した．すなわち，本来なら $\mathbf{v}_{\mathrm{r}}(\chi).z > 0$ に対しては，$d\mathbf{F}_{\mathrm{D}}.z = 0$ であるとした方が，モデルとして適切であると考えられる．この場合は，以下のように場合分けして計算する．

1. $\mathbf{v}_\mathrm{r}(0).z < 0 \wedge \mathbf{v}_\mathrm{r}(|\mathbf{p}_x - \mathbf{p}_1|).z < 0$ のとき
   式 (E.38), (E.40) をそのまま使う.
2. $\mathbf{v}_\mathrm{r}(0).z > 0 \wedge \mathbf{v}_\mathrm{r}(|\mathbf{p}_x - \mathbf{p}_1|).z > 0$ のとき
   $\mathbf{v}_\mathrm{r}(0).z = 0$, $\mathbf{v}_\mathrm{r}(|\mathbf{p}_x - \mathbf{p}_1|).z = 0$ とおいて，式 (E.38), (E.40) を使う.
3. $\mathbf{v}_\mathrm{r}(0).z > 0 \wedge \mathbf{v}_\mathrm{r}(|\mathbf{p}_x - \mathbf{p}_1|).z < 0$ のとき
   a. $\mathbf{v}_\mathrm{r}(0).z = 0$, $\mathbf{v}_\mathrm{r}(|\mathbf{p}_x - \mathbf{p}_1|).z = 0$ とおいて $\chi = [0, \chi_0]$ で式 (E.37), (E.39) を積分（ただし γ_1, γ_x はそのまま使う）.
   b. もとの $\mathbf{v}_\mathrm{r}(0)$, $\mathbf{v}_\mathrm{r}(|\mathbf{p}_x - \mathbf{p}_1|)$ で $\chi = [\chi_0, |\mathbf{p}_x - \mathbf{p}_1|]$ で式 (E.37), (E.39) を積分.
   c. a, b の結果を足して，$\mathbf{F}_\mathrm{D}$, $\mathbf{T}_\mathrm{D}$ とする.
4. $\mathbf{v}_\mathrm{r}(0).z < 0 \wedge \mathbf{v}_\mathrm{r}(|\mathbf{p}_x - \mathbf{p}_1|).z > 0$ のとき
   3 と逆の操作.

ただし，χ_0 は $\mathbf{v}_\mathrm{r}(\chi).z = 0$ を満たす χ であり，

$$\chi_0 = -\frac{\mathbf{v}_\mathrm{r}(0).z}{\left(\boldsymbol{\omega} \times \frac{\mathbf{p}_x - \mathbf{p}_1}{|\mathbf{p}_x - \mathbf{p}_1|}\right).z} \tag{E.41}$$

として計算される.

まとめ

以上，トランポリンの反作用力として浮力項と粘性項を定義した．これらを用いて，ある円筒がトランポリンから受ける力 $\mathbf{F}$ とトルク $\mathbf{T}$ は,

$$\mathbf{F} = \mathbf{F}_\mathrm{K} - \mathbf{F}_\mathrm{D} \tag{E.42}$$

$$\mathbf{T} = \mathbf{T}_\mathrm{K} - \mathbf{T}_\mathrm{D} \tag{E.43}$$

と計算される.

なお，このモデルは現実世界での実験と比較して検証を行ったわけではないため，トランポリンのモデルとして必ずしも適切ではないかもしれない．しかし実験の目的がイベント駆動型制御の有効性を示すことにあり，トランポリンを動的な環境のひとつとして捉えているため，モデルの正確さは今回は問題にしない.

実験に用いた，トランポリンのパラメタを Table E.3 に示す.

List of Figures

1.1 Relation between control signal and observed trajectory 46
1.2 Motion learning of multi-link robot in a dynamic environment 46
2.1 Control primitive 47
2.2 Multi-link robot in dynamic environment 47
2.3 Input/output in event-driven control 47
3.1 An example of a primitive with some dependencies 48
3.2 Temporal relations between two intervals with a middle point 49
3.3 Calculation process of control signal from primitive 50
3.4 An example of EventGraph 50
5.1 4-link robot structure 51
5.2 EventGraph for trampoline hopping 51
5.3 Result of learning trampoline hopping 52
5.4 A series of images in trampoline hopping motion 52
5.5 Joint constraint different from human's 52
5.6 Result of learning trampoline hopping in low constraint 53
5.7 EventGraph for the low constraint robot 53
5.8 Result of learning trampoline hopping in the different graph structure . . 53
C.1 A part of EventGraph 54
C.2 Update propagation in a control node 54
E.1 Trampoline model 55
E.2 Reaction model of a trampoline 55

List of Tables

E.1 Link parameters 55
E.2 Joint parameters 55
E.3 Trampoline parameters 55

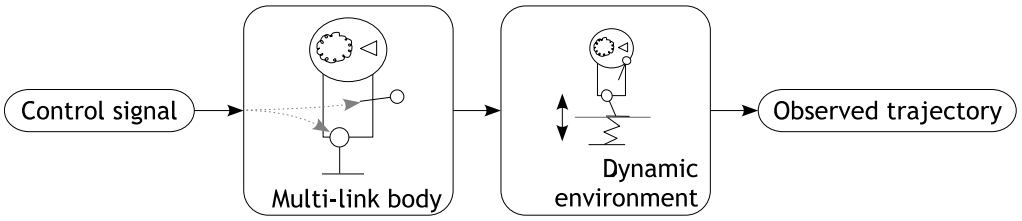


Fig. 1.1: Relation between control signal and observed trajectory

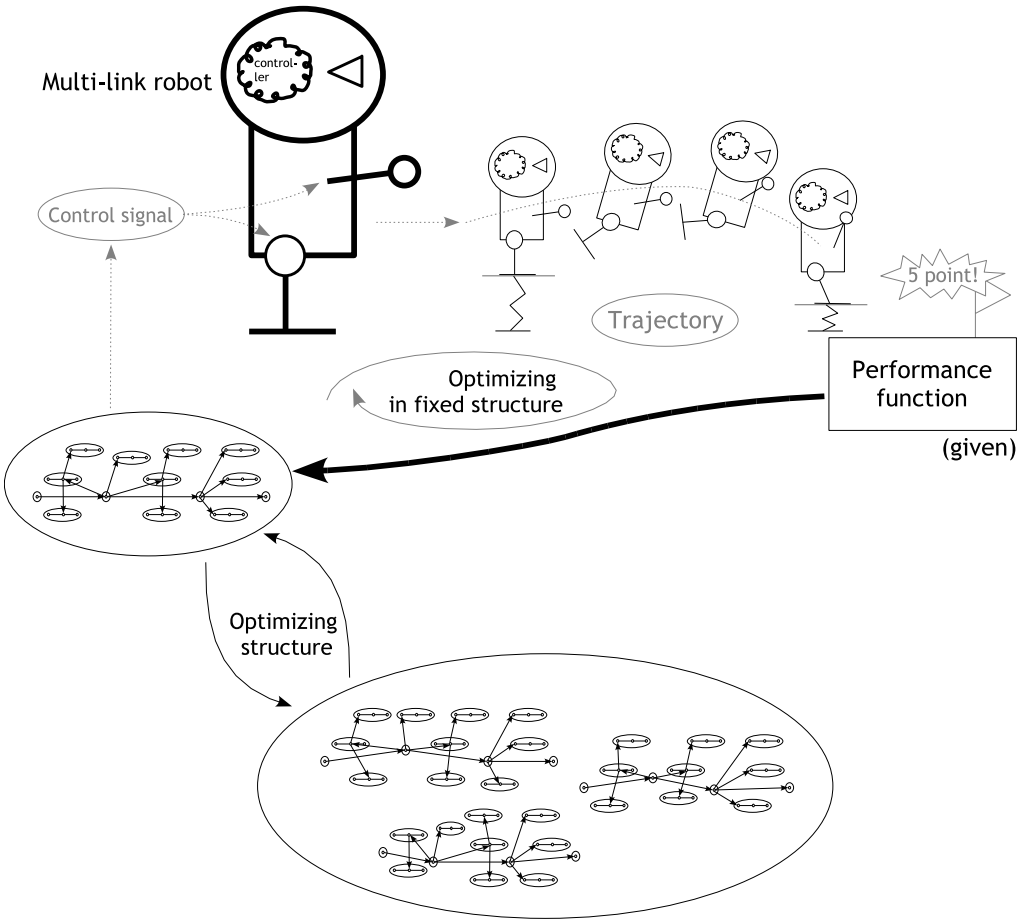


Fig. 1.2: Motion learning of multi-link robot in a dynamic environment

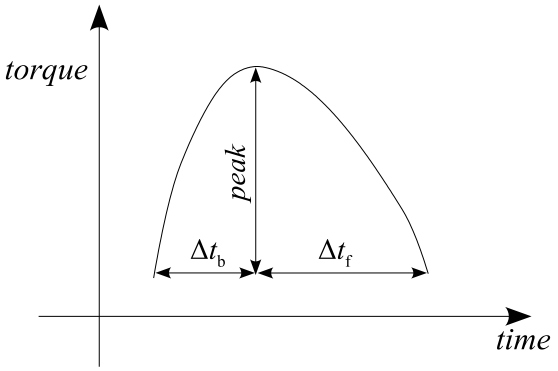


Fig. 2.1: Control primitive

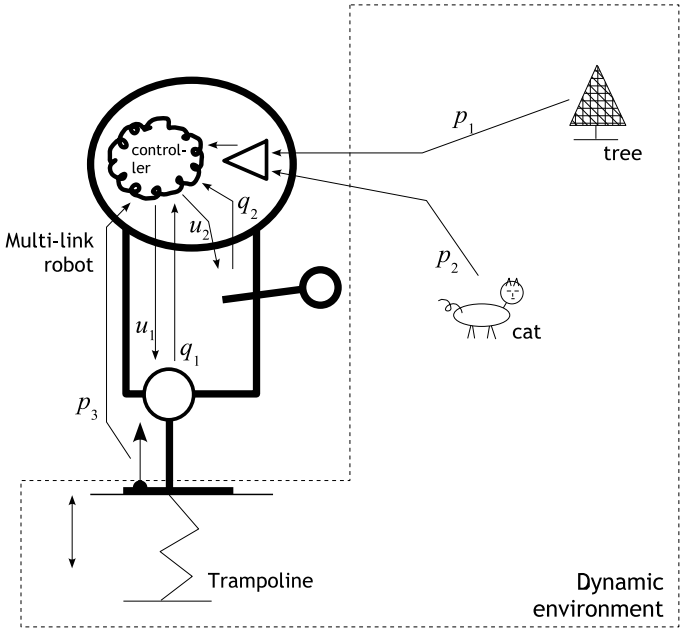


Fig. 2.2: Multi-link robot in dynamic environment

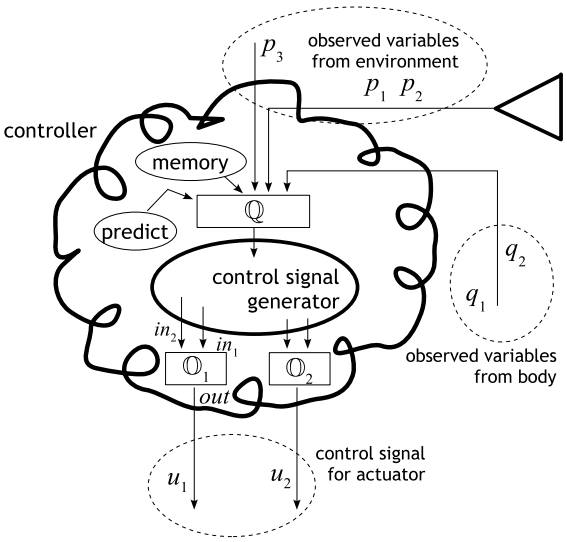


Fig. 2.3: Input/output in event-driven control

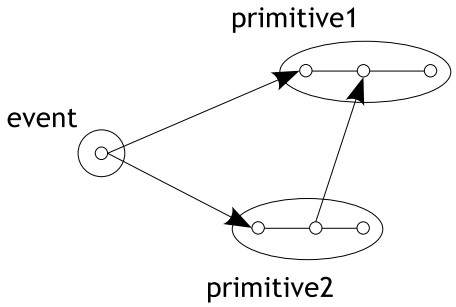
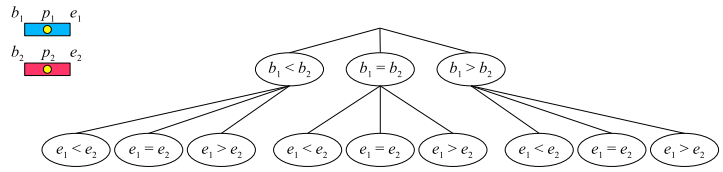
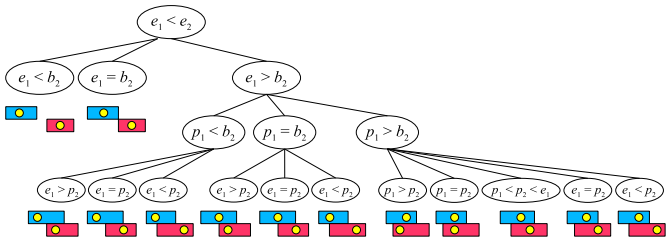


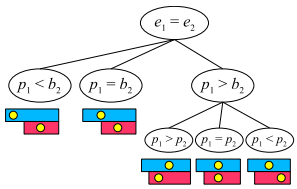
Fig. 3.1: An example of a primitive with some dependencies; ‘primitive1’ depends on ‘event’ and ‘primitive2’.



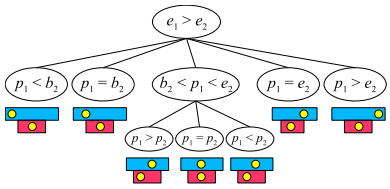
(a) Fundamental relation of each start-point and end-point



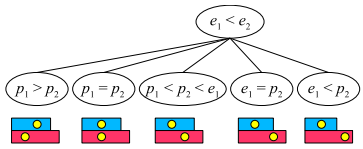
(b) $e_1 < e_2$ in $b_1 < b_2$



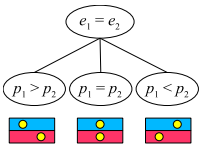
(c) $e_1 = e_2$ in $b_1 < b_2$



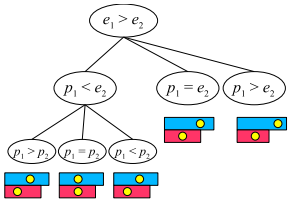
(d) $e_1 < e_2$ in $b_1 < b_2$



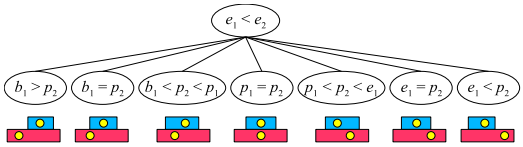
(e) $e_1 < e_2$ in $b_1 = b_2$



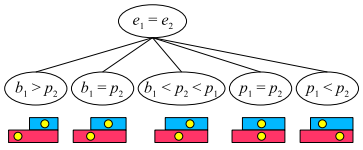
(f) $e_1 = e_2$ in $b_1 = b_2$



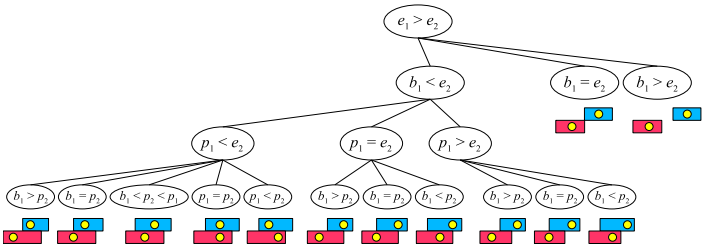
(g) $e_1 > e_2$ in $b_1 = b_2$



(h) $e_1 < e_2$ in $b_1 > b_2$



(i) $e_1 = e_2$ in $b_1 > b_2$



(j) $e_1 > e_2$ in $b_1 > b_2$

Fig. 3.2: Temporal relations between two intervals with a middle point. Fig.3.2(a) is a fundamental relation of each start-point and end-point. Fig.3.2(b)-3.2(j) are details of Fig.3.2(a).

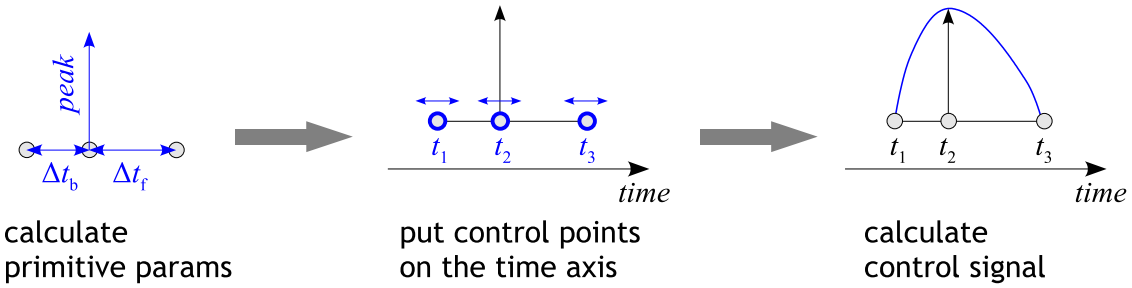


Fig. 3.3: Calculation process of control signal from primitive

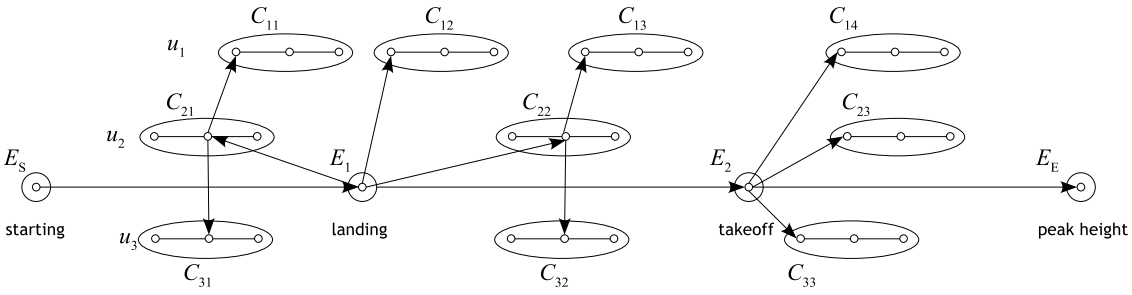
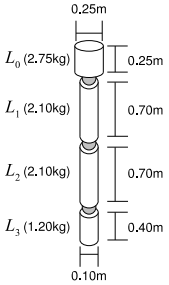
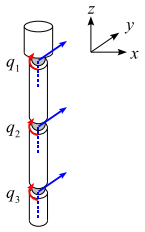


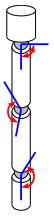
Fig. 3.4: An example of EventGraph



(a) Size and weight of links

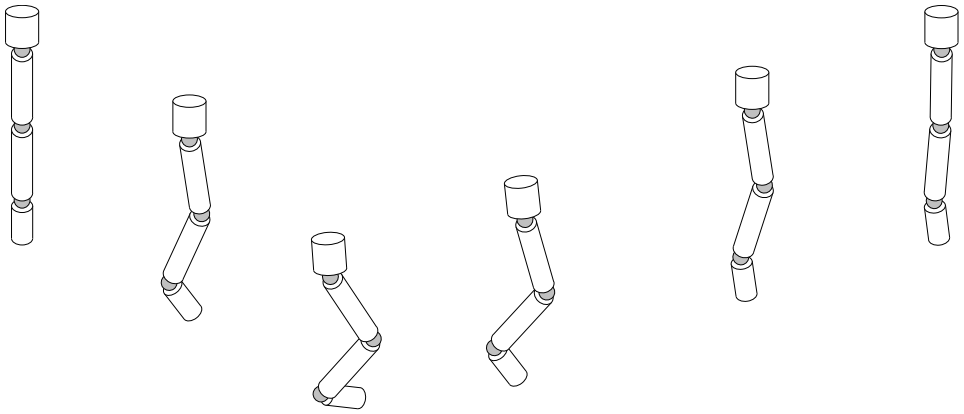


(b) Joint axis and angle

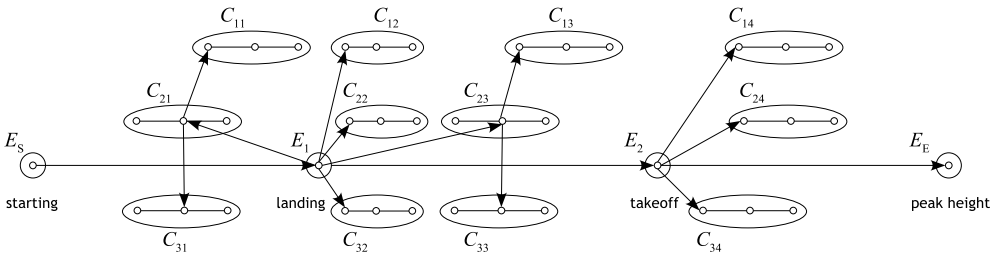


(c) Joint constraint

Fig. 5.1: 4-link robot structure.



(a) Hopping motion on a trampoline



(b) *EventGraph*

Fig. 5.2: *EventGraph* for trampoline hopping.

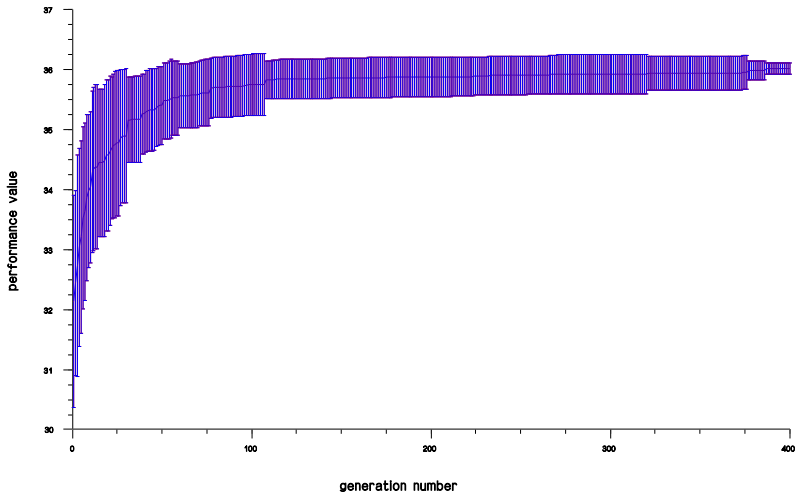


Fig. 5.3: Result of learning trampoline hopping. experiment code: exp-op5-00, directory: simulation/optimizing-eg5.

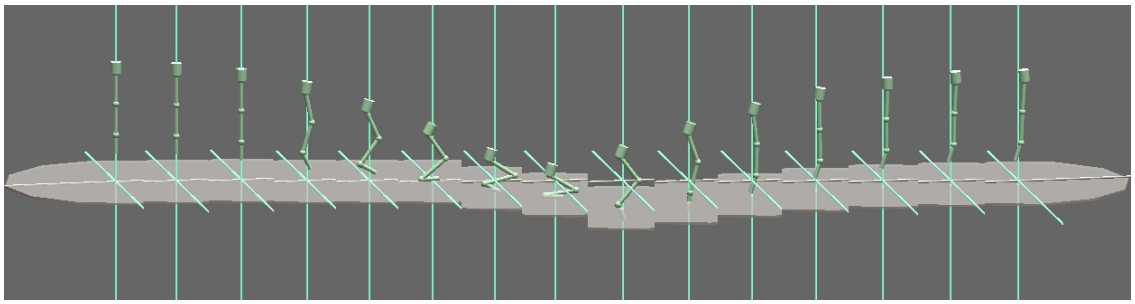


Fig. 5.4: A series of images in trampoline hopping motion (0.1s per frame).

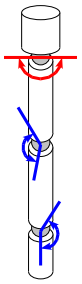


Fig. 5.5: Joint constraint different from human's; concretely, the constraint at the joint J_1 is less stiff than that of Fig. 5.1(c).

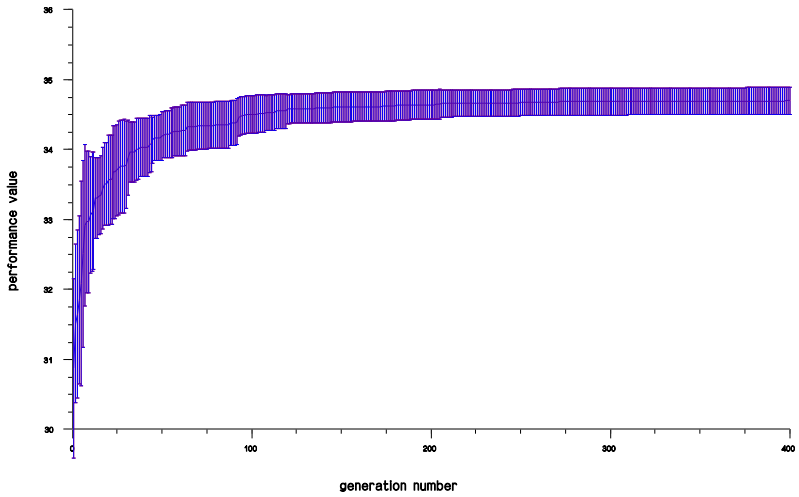


Fig. 5.6: Result of learning trampoline hopping in low constraint at the joint J_1. experiment code: exp-op42-01, directory: simulation/optimizing-eg42.

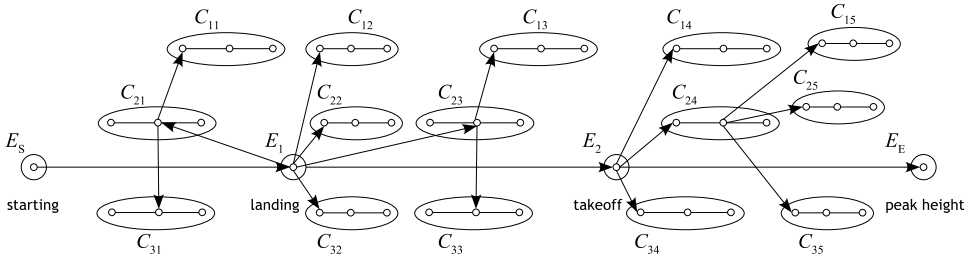


Fig. 5.7: *EventGraph* for the low constraint robot; C_{15}, C_{25}, C_{35} are added to the *EventGraph* shown in Fig. 5.2(b).

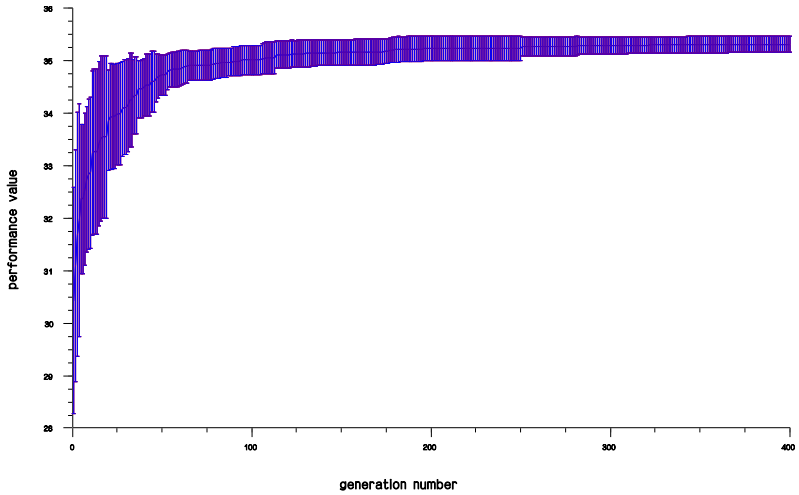


Fig. 5.8: Result of learning trampoline hopping in the different graph structure. experiment code: exp-op42-02, directory: simulation/optimizing-eg42.

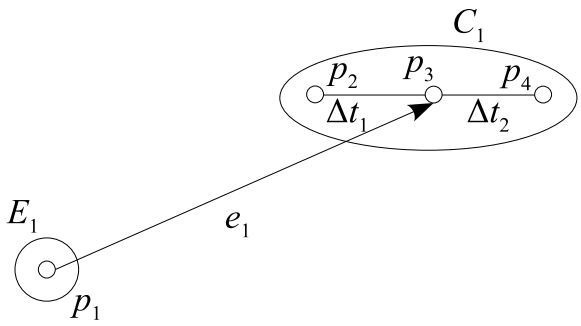


Fig. C.1: A part of EventGraph

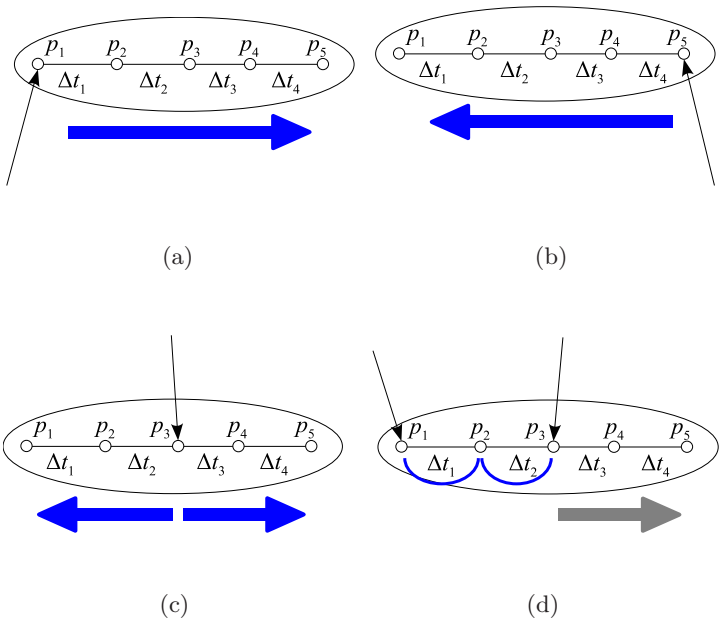


Fig. C.2: Update propagation in a control node

Table E.1: Link parameters

Link(j)	m_j [kg]	$length_j$ [m]	$diameter_j$ [m]
L_0	2.75	0.25	0.25
L_1	2.10	0.70	0.10
L_2	2.10	0.70	0.10
L_3	1.20	0.40	0.10

Table E.2: Joint parameters

Joint(j)	Movable range [rad]	K_{J_j} [Nm]	D_{J_j} [Nms]	μ_j [kgm^2]
J_1	$[-\frac{\pi}{2}, 0]$	0.02	2.50	16×10^{-4}
J_2	$[\frac{\pi}{15}, \frac{11.5\pi}{15}]$	0.02	2.50	4.41×10^{-4}
J_3	$[-\frac{5\pi}{6}, 0]$	0.01	1.25	4.41×10^{-4}

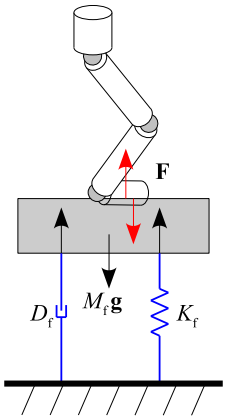


Fig. E.1: Trampoline model.

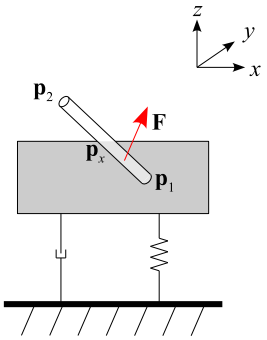


Fig. E.2: Reaction model of a trampoline.

Table E.3: Trampoline parameters

M_f [kg]	1.0
K_f [Nm^{-1}]	5.0×10^2
D_f [Nm^{-1}s]	1.0×10^2
K_d [Nm^{-3}]	5.0×10^4
D_d [Nm^{-3}s]	1.0×10^4