複数のプロセッサを用いたロボット制御則の並列処理に関する研究

田川，聖治

(Degree)
博士（工学）

(Date of Degree)
1997-04-25

(Date of Publication)
2008-02-27

(Resource Type)
doctoral thesis

(Report Number)
乙2136

(JaLCDOI)
https://doi.org/10.11501/3129899

(URL)
https://hdl.handle.net/20.500.14094/D2002136

神戸大学博士論文

複数のプロセッサを用いた
ロボット制御則の並列処理に関する研究

平成９年３月

田川 聖治

内容梗概

現在、ロボットは生産現場だけでなく、多くの分野で使われており、その形態は様々である。また、ロボットに関連する学問や技術も、計測、制御、材料、情報処理など多岐に渡る。本論文が対象とするロボットは、幾つかの剛体リンクを、直動あるいは回転関節により相互に結合させたリンク機構から成るロボット・アームである。このようなロボット・アーム（以降、ロボット）を、高速かつ高精度に制御するためには動的制御が有効である。ところが、ロボットの動的制御を実現するためには、膨大な演算を含む制御則の計算を、実時間で行う必要がある。そこで、本論文では、複数のプロセッサを用いた、ロボットの動的制御における計算の並列処理手法を提案する。

ロボットの動的制御とは、非線形な動特性を有するロボットに対して、その動的モデルに基づく非線形な状態フィードバック補償を施すことにより、ロボットを線形かつ非干渉なシステムに変換して制御するものである。従って、ロボットの動的制御に貢献する制御則は、ロボットの動的モデルに基づく非線形な下位制御則と、線形化されたシステムに対する線形な上位制御則に分けることができる。そこで、これらの制御則の計算に対して独立に、その特性を考慮した並列処理手法を提案する。

まず、非線形な下位制御則（ロボット制御則）の計算に対して、汎用的な分散メモリ型の並列計算機モデルによる並列処理手法を提案する。一般にロボット制御則は、重複する冗長な演算を数多く含む多変数多項式となるため、そのまま計算しては効率が良くない。そこで、提案する並列処理手法では、ロボット制御則の計算を積項単位で複数のプロセッサに割り当てた後に、部分的因数分解など代数的変換により数式を簡単化して、各プロセッサごと逐次的に処理される計算式の演算回数を削減する。また、任意の個数のプロセッサを用いて、上記の並列処理に要する時間を最短とするために、ロボット制御則の簡単化と並列化を共に考慮したスケジューリング問題について考える。

次に、差分方程式として与えられる線形な上位制御則の計算に対して、複数のディジタル信号処理用プロセッサ（DSP）を用いた並列処理手法と、ディジタル制御器の構成方法を提案する。一般にDSPは、パイプライン処理の機能を有し、積和演算を効率的に実行できる。提案する並列処理手法では、各DSPが通信処理のためにパイプライン処理を中断することがない。さらに、実現する制御則の次数に関係なく、並列化するDSPの個数に比例して、ディジタル制御器のサンプリング周期を短縮できる。

ここで、ロボットの動的制御では、上記のディジタル制御器が、ロボットなどを含むフィードバック制御系に組み込まれる。このため、ロボット制御系の安定性を損なわないためには、ディジタル制御器のサンプリング周期と滞在時間を、共に短縮する必要がある。そこで、任意の個数DSPを用いたディジタル制御器において、サンプリング周期と滞在時間を共に最短とするための多目的スケジューリング問題について検討を行い、そのパレート最適解の1つを求める最適化アルゴリズムを提案する。

本論文の構成は全体を5章に分け、第1章の緒論の後、第2～3章でロボットの動的制御における非線形な下位制御則の処理について述べ、第4章で線形な上位制御則の処理について述べている。第5章は結論であり、本研究の成果を総括している。

目次

第1章

緒論

ロボットが研究の対象として登場したのは、1960年代の初頭である[1]。その後、ロボット技術は、めざましい進歩をとげ、今日、ロボットは、生産現場において広く普及すると共に、宇宙、海洋、原子力関連など、人間が耐えられない極限環境下での作業にも利用されている。さらに、将来に向けて、ロボットは、医療や福祉サービスなど、人間社会の全般に関わるより多くの分野において、その応用が期待されている。しかし、このように発展を遂げたロボット技術の多くは、計算機技術とエレクトロニクス技術の進歩に負うところが大きく、理論的な基礎研究の成果が、現在の計算機の能力不足から実用化に結びつかないもどかしさを感じる事例も少なくない。

計算機の処理能力が問題となるロボット技術の1つに、ロボット・アームの動的制御がある[2]。ロボット・アームとは、幾つかの剛体リンクを直動関節および回転関節により相互に結合させたリンク機構から成り、その先端部には、人間の手に相当する、作業に適した効果器が取り付けられている。また、ロボット・アームには、その手先をある位置から別の位置へ移動させるだけでなく、ある空間経路に沿って手先を動かすことも要求される。このような動作をロボット・アームに実行させるためには、各関節の変位だけでなく、その速度や加速度までも考慮した制御が必要となる。さらに、制御の立場から見れば、ロボット・アームは複雑な多変数の非線形システムであり、各関節には遠心力、コリオリ力、重力などによる相互干渉が作用して、高速かつ高精度な運動を実現するためには、これらの影響を補償する必要がある。

ここで、ロボット・アームの動的制御では、非線形な動特性を持つロボット・アームを、非線形状態フィードバック補償を施すことにより、線形かつ非干渉なシステムに変換する。さらに、ロボット・アームを含む線形化されたシステムに対して、サーボ補償器を設けることで、モデル化誤差や外乱の影響を低減させる。このような動的制御は、ロボット・アームの姿勢や外界に及ぼす力を、高速かつ高精度に制御できる有効な手法として期待され、ロボット制御理論において重要な位置を占めている[1]-[4]。

ところが、動的制御における非線形状態フィードバック補償は、ロボット・アームの運動方程式を、ほとんどそのまま計算するものである。従って、制御則に含まれる演算の多さが、実際のロボット・アームの制御に、動的制御を適用する場合の大きな障害となる。すなわち、ロボット・アームの動的制御に必要な計算を、実時間で行うためには、現在のところ、非常に高速な処理能力を持つ高価なコンピュータを使用せざるをえない。

そこで、処理能力および性能価格比において、さらに優れたロボット制御システムを構築するために、安価なマイクロプロセッサを用いた並列処理が注目を集めている [5]。このほか、ロボットの制御において並列処理が求められる背景には、要求の変化に応じて性能を柔軟に変更できるという利点も挙げられる。

しかしながら、これまでの並列処理の研究は、特定のロボット制御則の計算に偏り、汎用性に欠けるものである。まず、さきに述べた非線形状態フィードバック補償を実現するために、ロボット・アームの運動方程式に基づき、アームの各関節に加えるべきトルクや力を求めることを逆動力学計算と呼ぶ。さらに、この逆動力学計算は、ニュートン・オイラー（Newton-Euler）の運動方程式を用いると演算回数が少なく、その高速計算アルゴリズムがニュートン・オイラー法 [6] として知られている。そこで、ニュートン・オイラー法による逆動力学計算を、複数のプロセッサを用いて並列処理するための研究が盛んに行われた。これらの研究のうち主なものは、ニュートン・オイラー法におけるデータの流れを解析して、その特徴を考慮した専用のマルチプロセッサ・アーキテクチャを開発するアプローチ [7]、1サンプリング周期前のデータに基づく推定値を計算に用いたり、同じデータの計算を異なるプロセッサで重複して行うことによりニュートン・オイラー法における計算の流れをシストリックアレイなどによるパイプライン処理に適したものに変更するアプローチ [8],[9]、さらに、ニュートン・オイラー法の計算を適当なタスクに分割して、汎用の並列計算機による処理時間が最少となるようなスケジューリング方法を考えるアプローチ [10],[11] などが挙げられる。

ニュートン・オイラー法は、単一のプロセッサを用いて、逆動力学計算を行う場合には優れた計算アルゴリズムである。しかし、計算に伴うデータの流れが複雑であり、必ずしも並列処理に適した計算アルゴリズムであるとは言えない。また、動的制御における非線形状態フィードバック補償には、制御系の安定性をリアプノフ関数によって保証するために、ニュートン・オイラー法に代表される逆動力学計算のための高速計算アルゴリズムが、直接利用できないような制御則（線形化補償器）が採用される場合もある [2],[4]。従って、さらに多くの種類のロボット制御則に対して有効であり、かつ、並列処理に適した計算アルゴリズムの開発が嘱望されている。

また、従来の並列処理の研究では、非線形状態フィードバック補償により線形化されたシステムに対する上位の制御則の計算時間が、ほとんど考慮されていなかった。通常、伝達関数などの線形な制御則に含まれる演算回数は比較的に少ないため、線形なシステムの制御において、ディジタル制御器による制御則の計算時間が問題となることは希である。しかし、ロボット・アームの動的制御では、上位の線形な制御則を計算するディジタル制御器が、ロボット・アームや下位の非線形な制御則を含むフィードバック制御系に組み込まれる。このため、ロボット制御系の応答性や安定性を保証するためには、線形な制御則についても、これまでにない高速な処理が要求される。

この論文では、ロボット・アームの動的制御における制御則の計算の高速化に的を絞り、より効率的で汎用性の高い並列処理手法を提案する。これまで述べたように、ロボット・アームの動的制御に寄与する制御則は、多変数多項式として表される非線形な下位制御則と、伝達関数などにより表される線形な上位制御則に分けられる [3]。従って、提

案する並列処理手法としては、各々の制御則に対して独立に、その特性を考慮した複数のプロセッサによる並列処理を行うものとする。

この論文の構成は以下の通りである。第1章の緒論の後、第2～3章で非線形な下位制御則の処理について述べ、第4章で線形な上位制御則の処理について述べる。最後に、第5章として結論を示す。

まず、第2章では、ロボット・アームの動的制御における非線形な下位制御則の特徴と、数式の代数的な変換によるロボット制御則の簡単化手法について述べる。ロボット・アームの運動方程式に基づく非線形な下位制御則（ロボット制御則）は、長大な多変数多項式として表され、膨大な演算を含むと共に、重複する冗長な演算が数多く存在する。従って、このようなロボット制御則は、部分的な因数分解や共通部分式の括り出しなど、数式の簡単化によって冗長な演算を削除するとよい。そこで、数式の簡単化による演算回数の削減アルゴリズムを提案する。提案する数式の簡単化手法を用いると、あらゆるロボット制御則に対して、含まれる演算回数が最少であるという意味で最適化された計算公式を機械的に得ることができる。

次に、第3章では、ロボット・アームの動的制御において、多変数多項式により与えられる非線形な下位制御則の並列処理について述べる。提案する並列処理手法では、さきに述べた数式の簡単化手法を用いて、各プロセッサごと逐次的に処理される演算回数を減らすことにより、並列処理全体としての計算時間を最少とする。この並列処理手法は、多変数多項式として表される任意のロボット制御則の計算に有効であり、かつ、プロセッサ間の通信が少ないために、多くの並列計算機[12],[13]において適用可能である。また、第3章では、上記の並列処理おける計算時間を最短とするためのスケジューリング問題について考える。さらに、このスケジューリング問題に対して、3種類のスケジューリング・アルゴリズムを提案する。すなわち、大規模な問題にも適用可能な近似アルゴリズム、分枝限定法[14]に基づき厳密に最適解を求める最適化アルゴリズム、これら2つのアルゴリズムを組み合わせた準最適化アルゴリズムである。

第4章では、ロボット・アームの動的制御において、伝達関数などにより与えられる線形な上位制御則の並列処理について述べる。まず、複数のディジタル信号処理プロセッサDSP（Digital Signal Processor）[15]–[17]を用いたディジタル制御器の実現方法を提案する。また、このディジタル制御器が、ロボット・アームなどを含むフィードバック制御系に組み込まれることを考慮して、ディジタル制御器のサンプリング周期と滞在時間を、共に最短とするための多目的スケジューリング問題について考える。さらに、この多目的スケジューリング問題に対して、分枝限定法に基づき、パレート（Pareto）最適解を求める最適化アルゴリズムを提案する。

第2章

ロボット制御則の導出と簡単化

2.1 緒言

ロボット・アーム（以降、ロボットと呼ぶ）を高速かつ高精度に制御するためには、動的制御が有効である。ところが、実際のロボット制御において、動的制御を適用するためには、ロボットの運動方程式（動的モデル）に基づく長大な制御則の計算を実時間で行う必要がある [2]。この問題に対するソフトウェア的な解決策として、ロボット制御則に依存した高速計算アルゴリズムの開発があり、逆動力学計算のためのニュートン・オイラー（Newton-Euler）法 [6] やラグランジェ（Lagrange）法 [18] に代表される、幾つかの計算アルゴリズムが提案されている [18]–[21]。

ところが、これらの計算アルゴリズムは、個々のロボット制御則に固有のものであり、新たな理論に基づくロボット制御則を採用する場合には、その計算アルゴリズムに対しても検討が必要となる。すなわち、理論的に優れた制御方法であっても、その制御則の計算を実時間で行うことが困難であれば、実際のロボット制御に適用することは不可能である。例えば、さきに述べたニュートン・オイラー法は、ロボット制御系の安定性をリアプノフ（Lyapunov）関数によって保証するような、逆動力学計算とは異なるロボット制御則（線形化補償器）[2],[4] の計算には、使用できない。

さらに、従来の計算アルゴリズムの善し悪しは、制御の対象とするロボットの関節数（自由度）のみを基準とした計算量で評価されており、多様なロボットの幾何学的な構造の違いが、ロボット制御則の計算アルゴリズムに反映されていない。例えば、ロボットの構造を工夫することにより、そのロボットの運動方程式を単純にすることは可能である [22]。また、ある特徴的な構造を持つロボットに対しては、ニュートン・オイラー法とラグランジェ法を組み合わせることで、これらの計算アルゴリズムを単独で用いる場合よりも、高速に逆動力学計算が行えるという報告もある [23]。

この章では、初めにロボットの動的制御について、簡単に概要を紹介する。ロボットの動的制御においては、制御の対象とするロボットの運動方程式に基づく非線形な制御則を用いて、そのロボットの非線形な動特性を補償する。そこで、このようなロボット制御則の一般的な特徴を述べると共に、ロボットの構造解析や制御系の設計によく利用

されるMathematicaやREDUCEなどの数式処理システム[24],[25]を用いて、実際のロボットに対する具体的なロボット制御則を導出する方法を示す。

次に、従来のロボット制御則の計算アルゴリズムに関する問題点を補うために、あらゆるロボット制御則に対して、含まれる演算回数の総和が最少であるという意味で最適化された計算公式を、自動的に生成するためのシステマチックな手法を提案する。提案する手法は、数式処理システムを用いて、ロボット制御則をスカラの多変数多項式として導出した後に、部分的な因数分解や共通部分式の括り出しなど、数式の代数的な変換を行うことにより、そのロボット制御則の計算式を簡単化して、それに含まれる冗長な演算を削除するものである。

さらに、この章では、提案した手法を応用して開発した最適化コンパイラについて述べる。この最適化コンパイラは、数式処理システムを用いて導出したロボット制御則から、数式の簡単化を経て、C言語で記述された効率的なロボット制御則の計算プログラムを自動生成する。計算プログラムを記述するC言語は移植性に優れ、組み込み制御プログラムの作成にも広く利用されている。従って、最適化コンパイラの出力であるC言語のプログラムを、適当なCコンパイラによって再度コンパイルすることで、多くの計算機システム上で実行可能なロボット制御則のプログラムを、容易に得ることができる。また、最適化コンパイラ自体は、近年、新しいソフトウェアの設計方法として注目されているオブジェクト指向[26]に基づき開発している。

2.2 ロボットの動的制御とロボット制御則

ロボットの動的制御とは、ロボットの運動方程式（動的モデル）に基づいて制御系を設計することであり、非線形特性の影響が無視できない高速、かつ、高精度な制御に有効である。ここでは、ロボットの動的制御について概要を紹介した後に、数式処理システムを用いたロボット制御則の導出方法について説明する。

2.2.1 ロボットの動的制御

ロボット（ロボット・アーム）は、幾つかの剛体リンクを直動および回転関節により相互に結合させたリンク機構からなり、その姿勢によって各関節に加わる慣性モーメントや重力負荷などが大きく変わる非線形な動特性を有する。このようなロボットは剛体系であるために、ラグランジェ（Lagrange）の運動方程式によりモデル化できる。また、ラグランジェの運動方程式は、ロボットの各関節の変位を一般化座標系として、ハミルトンの原理から誘導されるが、詳細は文献[1],[4]などを参照されたい。

例えば、図2.1に示すような2つの回転関節を持つ2自由度のロボットのラグランジェ運動方程式は、次式のような三角関数を含む行列の形の微分方程式となる。ここで、ロボットの自由度とは、独立に駆動・制御できる関節軸の数をいう。

$$\begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} * \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} + \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{bmatrix} * \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} + \begin{bmatrix} g_1 \\ g_2 \end{bmatrix} \tag{2.1}$$

$$\begin{bmatrix}
M_{11} &:= & 2*m_2*r_2*L_1*\cos(q_2) + m_2*r_2**2 + m_2*L_1**2 + N_1 + N_2 \\
M_{12} &:= & m_2*r_2*L_1*\cos(q_2) + m_2*r_2**2 + N_2 \\
M_{21} &:= & m_2*r_2*L_1*\cos(q_2) + m_2*r_2**2 + N_2 \\
M_{22} &:= & m_2*r_2**2 + N_2 \\
H_{11} &:= & -2*m_2*r_2*L_1*\sin(q_2)*\dot{q}_2 \\
H_{12} &:= & -m_2*r_2*L_1*\sin(q_2)*\dot{q}_2 \\
H_{21} &:= & m_2*r_2*L_1*\sin(q_2)*\dot{q}_1 \\
H_{22} &:= & 0 \\
g_1 &:= & g*(m_2*L_1*\cos(q_1) + m_2*r_2*\cos(q_2)*\cos(q_1) \\
& & - m_2*r_2\sin(q_2)*\sin(q_1)) \\
g_2 &:= & g*(m_2*r_2*\cos(q_2)*\cos(q_1) - m_2*r_2*\sin(q_2)*\sin(q_1))
\end{bmatrix}$$

ただし、f_n $(n = 1, 2)$は各関節に加わるトルク、q_nは各関節の変位、gは重力加速度である。その他、式(2.1)にはロボットの運動方程式を決定するための構造パラメータや慣性パラメータが幾つか含まれているが、これらについては後で述べる。

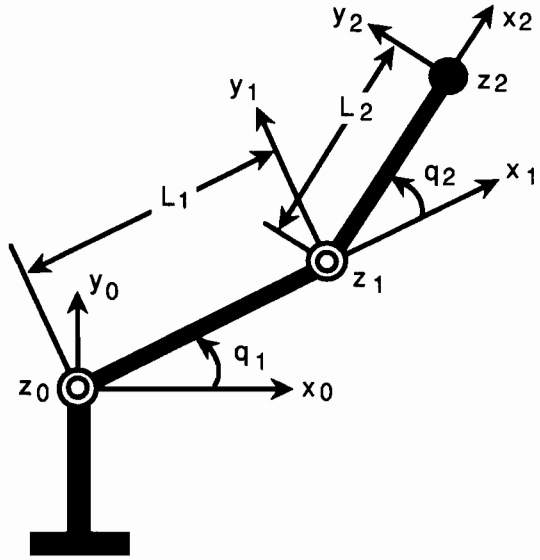


図2. 1　2自由度のロボット・アーム

一般に、N個の関節（N自由度）を持つロボットのラグランジュ運動方程式は、次式に示すような行列の形の微分方程式となる [3],[4]。

$$\mathbf{f}(t) = M(\mathbf{q}(t)) * \ddot{\mathbf{q}}(t) + H(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{q}}(t) + g(\mathbf{q}(t)) \tag{2.2}$$

ただし、$\mathbf{q}(t) \in \mathbf{R}^N$は関節変位を表すベクトル、$\mathbf{f}(t) \in \mathbf{R}^N$は関節に加わるトルクまたは力を表すベクトルである。また、式(2.2)右辺の第1項は慣性力を、第2項はコリオリ力と遠心力を、第3項は重力の影響を表す。ここで、慣性行列 $M(\mathbf{q}(t)) \in \mathbf{R}^{N\times N}$ は、有界かつ正定な対称行列となることが知られている [4]。

ロボットの動的制御における典型的な手法の1つは、式(2.3)に示すような逆動力学計算による非線形状態フィードバックを施すことにより、ロボットを時間関数として任意に与えられた目標軌道 $\mathbf{p}$ ($\mathbf{p}(t) \in \mathbf{R}^N$) と、出力である関節変位 $\mathbf{q}$ ($\mathbf{q}(t) \in \mathbf{R}^N$) との関係において、線形な動的システムに変換するものである。

$$\mathbf{f}(t) := M(\mathbf{q}(t)) * \ddot{\mathbf{p}}(t) + H(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{q}}(t) + g(\mathbf{q}(t)) \tag{2.3}$$

ただし、$\ddot{\mathbf{p}}(t) \in \mathbf{R}^N$は目標軌道 $\mathbf{p}$ の時間 t における値の加速度である。

式(2.2)でモデル化されるロボットに対して、式(2.3)に基づき算出される駆動トルク $\mathbf{f}(t)$ を加えると、次式に示すような線形かつ非干渉な2重積分系が得られる。

$$\ddot{\mathbf{q}}(t) = \ddot{\mathbf{p}}(t) \tag{2.4}$$

式(2.2)に示したロボットのモデルが完全に正確であり、また、システムに外乱が入らないとすれば、目標軌道 $\mathbf{p}$ の時間 t における加速度$\ddot{\mathbf{p}}(t)$ を、式(2.4)の線形化システムに対する入力（操作量）とすることで、初期条件 $\mathbf{p}(t_0) = \mathbf{q}(t_0)$, $\dot{\mathbf{p}}(t_0) = \dot{\mathbf{q}}(t_0)$ を満足する

目標軌道 $\mathbf{p}$ を、$\mathbf{q} = \mathbf{p}$ として完全に達成することができる。

しかし、実際には式 (2.2) のモデルが完全に正確なことはありえず、モデル化誤差や外乱の混入は避けられない。そこで、これらの影響は式 (2.4) の線形化システムに対してサーボ補償器を設けることで低減するものとする。このようなロボット制御系を、ブロック線図で表すと図2.2のようになる。図2.2のシステムでは、式 (2.3) の逆動力学計算において、$\ddot{\mathbf{p}}(t)$ の代わりに、$\mathbf{q}(t)$ や $\dot{\mathbf{q}}(t)$ の値を考慮して $\ddot{\mathbf{p}}(t)$ を適当に修正した値 $\ddot{\mathbf{p}}^*(t)$ を使用しており、逆動力学計算（下位の制御則）によるロボットの線形化と、線形化されたシステムに対するサーボ補償（上位の制御則）という2段階制御に成っている [3]。このほか、逆動力学計算を用いたロボットの線形化による制御手法には、様々なバリエーションが提案されている [27]–[31]。

ロボットの動的制御における他の手法として、式 (2.3) の逆動力学計算の代わりに、次式に示す線形化補償器によって非線形状態フィードバックを行うものがある。

$$\mathbf{f}(t) := M(\mathbf{q}(t)) * \ddot{\mathbf{p}}(t) + H(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{p}}(t) + g(\mathbf{q}(t)) \tag{2.5}$$

式 (2.5) に基づきロボットを線形化すると、ロボットの運動方程式を決定するパラメータの値に誤差が含まれる場合でも、安定性がリアプノフ関数によって保証されるロバストな制御系を構成することができる [32],[33]。このようなロボット制御系に対するリアプノフ関数と安定性の証明については、Koditschek[34] により示されたものを、付録Aにおいて紹介する。ただし、ロバストなロボット制御系であっても、外乱抑制や過渡特性を制御するために、上位のサーボ補償器は必要である。

さらに、ロボットの動的制御と呼ばれるものには、目標軌道を各関節変位の代わりに手先の位置で与える加速度分解制御 [27]、ロボットの手先の位置と手先が外界に及ぼす力を同時に制御するハイブリッド制御 [28]、ロボット全体の動特性を目的とする作業に適した機械インピーダンスに一致させるインピーダンス制御 [29]、ロボットのモデルに含まれる未知パラメータの値をリアルタイムに推定する適応制御 [30],[32] などがある。これらの制御手法に共通する特徴は、式 (2.3) の逆動力学計算、あるいは、式 (2.5) の線形化補償器に基づく駆動トルク $\mathbf{f}(t)$ の実時間計算を必要とすることである。

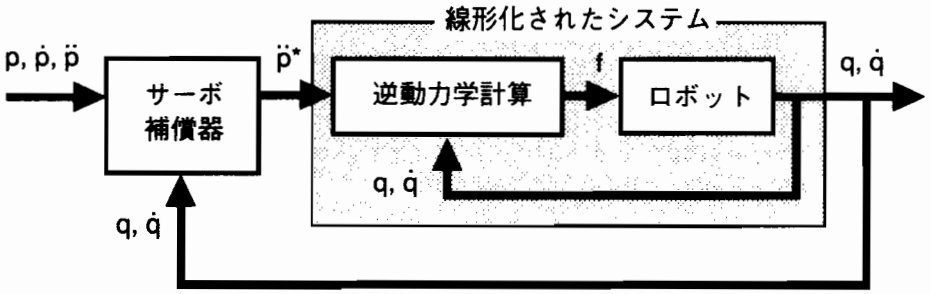


図2．2　ロボットの制御系

2.2.2 数式処理システムによるロボット制御則の導出

数式処理システム [24],[25] とは、数式を数値ではなく記号として処理する計算機システムであり、汎用の数式処理システムとしてはREDUCEやMathematicaなどがよく知られている。数式処理システムのロボティクスへの応用としてはロボット制御系のCADなどがあり、行列の式として与えられたロボットの運動方程式などを数式処理システムを用いてスカラの式に展開して、機構の解析や制御系の設計に利用する試みが成されている [35]–[39]。ここでは、数式処理システムを用いたロボットの運動方程式と、それに基づくロボット制御則の導出方法について説明する。

ロボットは一端が台座に固定され、もう一端が空間中を自由に動くようなリンクの直鎖機構であるため、その運動方程式を求める際には、リンクごとに座標系を設定すると便利である。そこで、ロボットの各リンク間の相対的な位置関係を明確に記述するために、Denavit と Hartenberg によって提案された D-H 記法がよく用いられる [40]。ここで、N個の関節からなるロボットを考える。各リンクには、台座（ベース）に固定されたリンクを 0 として、ベース側からアームの手先に向かって 1, 2, ..., Nと昇順に番号が付けられているものとする。D-H 記法では、初めに、ロボットの台座に右手系でベース座標系を任意に設定する。次に、0 番目のリンクに固定されるリンク座標系 $[x_0,\ y_0,\ z_0]$ を、このベース座標系に一致させる。さらに、n $(1 \leq n \leq N)$ 番目のリンクに固定されたリンク座標系 $[x_n,\ y_n,\ z_n]$ を、以下の手順に従って設定する。

【リンク座標系の設定方法】

(1) $n+1$ 番目の関節の回転または直動軸を z_n軸とする。z_n軸の方向は回転または直動の正方向とする。ただし、z_N軸については z_{N-1}軸と同方向にとる。

(2) z_{n-1}軸と z_n軸の共通法線と、z_n軸の交点を、リンク座標系の原点 O_nとする。z_{n-1}軸と z_n軸が交わるときは、その交点を原点 O_nとする。

(3) z_{n-1}軸と z_n軸の共通法線を、z_{n-1}軸から z_n軸方向に延長した軸を、x_n軸とする。z_{n-1}軸と z_n軸が交わるときは、$z_{n-1} \times z_n$に平行に x_n軸をとる。向きは手先に向かう方向とするが、それが明確でない場合には任意でよい。

(4) x_n軸と z_n軸に対して、右手系となるように y_n軸を決める。 ■

上記のように各リンク座標系を決定すると、隣り合う座標系の関係が、次の式(2.6)に示すような 4×4 の変換行列 A_n^{n-1}によって表される。

$$A_n^{n-1} := \begin{bmatrix} c_{\theta n} & -s_{\theta n} * c_{\alpha n} & s_{\theta n} * s_{\alpha n} & a_n * c_{\theta n} \\ s_{\theta n} & c_{\theta n} * c_{\alpha n} & -c_{\theta n} * s_{\alpha n} & a_n * s_{\theta n} \\ 0 & s_{\alpha n} & c_{\alpha n} & d_n \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

ただし、$s_{\theta n} = \sin(\theta_n)$, $c_{\theta n} = \cos(\theta_n)$, $s_{\alpha n} = \sin(\alpha_n)$, $c_{\alpha n} = \cos(\alpha_n)$ である。

式 (2.6) の行列 A_n^{n-1}は、第 n 座標系から第 $n-1$ 座標系への変換行列であり、A_n^{n-1}に含まれる 4 つのパラメータ $(\theta_n, d_n, a_n, \alpha_n)$ は、それぞれ以下のような意味を持つ。ここで、関節 n が回転関節の場合はθ_nが、直動関節の場合は d_nが、ロボットの各関節変数 q_n となり、残りの 3 つのパラメータは定数値となる。

【D-H パラメータ】

(1)　d_nは z_{n-1}の方向に測った第 $n-1$ 座標系原点と、z_{n-1}軸と x_n軸の交点間の距離。
(2)　a_nは x_nの逆方向に測った z_{n-1}軸と x_n軸の交点と、第 n 座標系の原点間の距離。
(3)　θ_nは x_{n-1}軸から x_n軸までの z_{n-1}軸まわりの回転角。
(4)　α_nは z_{n-1}軸から z_n軸までの x_n軸まわりの回転角。 ■

さらに、上記の変換行列 A_n^{n-1}を用いて、第 n 座標系から第 0 座標系（ベース座標系）への同次変換行列 T_nを、次のように定義する。

$$T_n := A_1^0 * A_2^1 * \cdots * A_n^{n-1} \tag{2.7}$$

ここで、重力加速度ベクトル $\mathbf{g}$、および、ロボットの各リンク n に関する疑似慣性行列 J_n、質量中心位置ベクトル $\mathbf{r}_n$が与えれると、式 (2.2) に示したロボットのラグランジュ運動方程式が、次式のように具体的に導出できる。

$$f_n = \sum_{j=1}^{N} M_{nj}(\mathbf{q}) * \ddot{q}_j + \sum_{j=1}^{N} H_{nj}(\dot{\mathbf{q}}, \mathbf{q}) * \dot{q}_j + g_n(\mathbf{q}), \quad (n = 1, 2, \ldots, N) \tag{2.8}$$

$$\left[\begin{array}{rcl} M_{nj}(\mathbf{q}) & := & \displaystyle\sum_{i=max\{n,j\}}^{N} Trace\left(\frac{\partial T_i}{\partial q_j} * J_i * \frac{\partial T_i^T}{\partial q_n}\right) \\ H_{nj}(\dot{\mathbf{q}}, \mathbf{q}) & := & \displaystyle\sum_{k=1}^{N} \left(\sum_{i=max\{n,j,k\}}^{N} Trace\left(\frac{\partial^2 T_i}{\partial q_j \partial q_k} * J_i * \frac{\partial T_i^T}{\partial q_n}\right) \right) * \dot{q}_k \\ g_n(\mathbf{q}) & := & \displaystyle\sum_{i=n}^{N} \left(-m_i * \mathbf{g}^T * \frac{\partial T_i}{\partial q_n} * \mathbf{r}_n \right) \end{array}\right.$$

ロボットの運動方程式の導出方法としては、ここで紹介したラグランジュ法のほか、ニュートン・オイラー法や漸化式表現によるラグランジュ法などが知られている [18]。当然のことながら、これらの方法により最終的に得られる運動方程式はすべて同じである。また、式 (2.3) に示した逆動力学計算や、式 (2.5) に示した線形化補償器の制御則は、式 (2.8) の関節変数を適当に置き換えることで同様に導出できる。

例えば、図 2.1 に示した 2 自由度のロボットにおいて、図 2.1 のようにリンク座標系を設定すると、その D-H パラメータは表 2.1 のようになる。また、重力加速度ベクトルを $\mathbf{g} := [0 \quad -g \quad 0]^T$とする。さらに、この 2 自由度のロボットの各リンク n $(n = 1, 2)$ が、

表2. 1　2自由度のロボットのD-Hパラメータ

座標系	α_n	θ_n	d_n	a_n
1	0	θ_1	0	0
2	0	θ_2	0	L_1

その座標系のx_n軸に対して左右対称的な形状であるとすると、各リンクnの疑似慣性行列J_nと質量中心位置ベクトル$\mathbf{r}_n$は、それぞれ式(2.9)と式(2.10)により与えられる。

$$J_n := \begin{bmatrix} 0 & 0 & 0 \\ 0 & N_n + m_n * r_n * *2 & 0 \\ 0 & 0 & N_n + m_n * r_n * *2 \end{bmatrix}, \quad (n = 1,\ 2) \tag{2.9}$$

$$\mathbf{r}_n := [r_n \ \ 0 \ \ 0]^T, \qquad (n = 1,\ 2) \tag{2.10}$$

ただし、リンクn $(n = 1,\ 2)$について、その長さをL_n、質量をm_n、重心の位置をr_n、重心まわりの慣性モーメントをN_nで表す。

ここで、数式処理システムREDUCEを用いて、式(2.8)のような行列の式を展開すると、図2.2に示した2自由度のロボットの運動方程式が、式(2.11)のように積和式の形で得られる。式(2.11)の積和式は、さきに式(2.1)で示した運動方程式と同じものである。ただし、数式処理システムでは、すべての変数を記号（文字列）として扱うために、関節変位q_n $(n = 1,\ 2)$の速度$\dot{q}_n$と加速度$\ddot{q}_n$は、それぞれdq_n，ddq_nと表し、三角関数$\cos(q_n)$と$\sin(q_n)$は、それぞれc_n，s_nと表している。

$$\begin{aligned} f_1 \ = \ & 2 * m_2 * r_2 * L_1 * c_2 * ddq_1 + m_1 * r_1 * *2 * ddq_1 \\ & + m_2 * r_2 * *2 * ddq_1 + m_2 * L_1 * *2 * ddq_1 \\ & + N_1 * ddq_1 + N_2 * ddq_1 + m_2 * r_2 * L_1 * c_2 * ddq_2 \\ & + m_2 * r_2 * *2 * ddq_2 + N_2 * ddq_2 - m_2 * r_2 * L_1 * s_2 * dq_2 * *2 \\ & - 2 * m_2 * r_2 * L_1 * s_2 * dq_2 * dq_1 + g * m_1 * r_1 * c_1 \\ & + g * m_2 * L_1 * c_1 + g * m_2 * r_2 * c_2 * c_1 - g * m_2 * r_2 * s_2 * s_1 \\ f_2 \ = \ & m_2 * r_2 * L_1 * c_2 * ddq_1 + m_2 * r_2 * *2 * ddq_1 + N_2 * ddq_1 \\ & + m_2 * r_2 * *2 * ddq_2 + N_2 * ddq_2 + m_2 * r_2 * L_1 * s_2 * dq_1 * *2 \\ & + g * m_2 * r_2 * c_2 * c_1 - g * m_2 * r_2 * s_2 * s_1 \end{aligned} \tag{2.11}$$

式(2.11)からも分かるように、数式処理システムを用いてスカラの式に展開したロボット制御則は、三角関数の値が予め計算してあるものとして、それらを1つの入力変数と考えると、次式に示すような多変数の積和式の形で表される。

$$f_n := \sum_{j \in J_n} \left(\pm \prod_{i \in I_j} b_i * \prod_{m \in M_j} u_m * *h_{m,j} \right), \qquad (n = 1,\ 2,\ \ldots,\ N) \tag{2.12}$$

ただし、b_iは定数パラメータ、u_mは次数$h_{m,j}$の入力変数であり、添字集合I_jとM_jは、それぞれb_i，u_mに対する添字集合I，Mの部分集合である。

式 (2.12) における定数パラメータと入力変数の分類は、ロボット制御則に依存して決まり、例えば、式 (2.3) の逆動力学計算では、各関節変数、それらの時間微分値や三角関数が入力変数となり、リンクの長さなどの構造パラメータや、リンクの質量などの慣性パラメータ、さらに重力加速度が定数パラメータに指定される。ただし、適応制御 [30],[32] では、未知の慣性パラメータも入力変数となる。

ロボットの幾何学的な形状は、D-H 記法を用いた座標変換行列により表されるため、通常、ロボット制御則も可読性に優れた行列の式として記述される。数式処理システムを用いて、このようなロボット制御則を行列の式からスカラの式に展開すると、行列の零要素に関する無意味な計算を除くことができる。さらに、加法定理など幾つかの三角公式を利用して、三角関数に関する簡単化を行うことも可能である [38]。従って、行列の式として与えられたロボット制御則を、そのまま行列の計算式としてプログラミングする代わりに、数式処理システムによってスカラの式に展開してから、その計算プログラムを作成することが提案されている [37]。

ところが、REDUCE や Mathematica など汎用の数式処理システムにより可能な数式の簡単化は、項の書き換え規則に基づく変換のみであり、停止性や完全性が保証されておらず、常に数式が簡単化されるとは限らない [25]。すなわち、項の書き換えにより逆に数式が複雑になる場合もあり、適切な書き換え規則を見出すことが難しい。さらに、数式の部分的な因数分解ができないなど、汎用の数式処理システムの能力には限界がある。従って、一般に数式処理システムを用いて導出したロボット制御則は、式 (2.11) からも推察できるように、長大で冗長な演算を多く含んだものとなる。

2.3 ロボット制御則の簡単化手法

前節で述べたように、数式処理システムを用いて導出したロボット制御則は、長大で冗長な演算を多く含んでいる。そこで、このようなロボット制御則に対して有効な数式の簡単化手法を提案する [41],[42]。提案する数式の簡単化手法は、従来の数式処理システムの能力を凌駕するものであり、任意のロボット制御則を、含まれる演算回数が最少であるという意味で最適化された形に変換する。

数式処理システムを用いて導出した式 (2.12) のロボット制御則は、定数パラメータ b_i 同士の無意味な乗算を除いて 1 つの定数にまとめ、これにより同じ値となる定数を新たな 1 つの記号 a_k によって表すと、次式のような積和式の形に変換できる。

$$f_n := \sum_{j \in J_n} \left(\pm a_k * \prod_{m \in M_j} u_m * *h_{mj} \right), \qquad (n = 1,\ 2,\ \ldots,\ N) \tag{2.13}$$

提案する数式の簡単化手法では、式 (2.13) に示したような積和式を、複数の積和式に繰り返し分解する。この過程において、簡単化の対象となる数式は、次の式 (2.14) に示すような積和式の形に一般化して捉えることができる。式 (2.14) において、a_k は定数を、v_i と v_m は変数を表す。また、これらの変数は、式 (2.13) における駆動トルク f_n のように数式の左辺のみに表れる出力変数、u_m のように右辺のみに表れる入力変数、数式の両辺に表れる中間変数に大別される。ただし、中間変数とは、以下に述べる数式の簡単化の過程で、自動的に導入されるものである。

$$v_i := \sum_{r=1}^{R} \left(\pm a_k * \prod_{m \in M_r} v_m \right) \tag{2.14}$$

ただし、M_r は出力変数を除く変数の添字集合の部分集合である。

式 (2.14) において、式 (2.13) に含まれる h 次 $(h \geq 1)$ のべき乗演算は、$(h-1)$ 回の乗算に展開されている。従って、提案する簡単化手法では、数式に含まれる乗算と加減算の削減を目的として、これらの演算回数により、簡単化の効果を評価する。

2.3.1 局所的な簡単化手法

提案する数式の簡単化では、5 つの局所的な簡単化手法、すなわち、「部分的因数分解」、「共通部分式の括り出し」、「共通因子の括り出し」、「定数の畳み込み」および「代入式の削除」を、式 (2.14) に示したような積和式に対して繰り返し適用する。最初の 3 つの局所的な簡単化手法は、Breuer[43] によって提案された 1 次の多変数多項式に対する簡単化手法を、高次の多項式にも適用できるよう改良したものである。また、残り 2 つの「定数の畳み込み」と「代入式の削除」は、一般的なコンパイラでも採用されている基本的な最適化手法である [44]。

▼ 部分的因数分解

式(2.14)において、1つ以上の要素（定数および変数）の積を因子と呼ぶ。この簡単化手法は、1つの積和式を構成する複数の積項から、それらに含まれる共通な因子を括り出すことで、冗長な乗算を削減する。

例えば、次の式(2.15)において、第1項と第2項の積項から共通因子$(a_1 * v_2 * v_2)$を括り出すと、式(2.15)は式(2.16)に変換できる。

$$v_1 := a_1 * v_2 * v_2 * v_3 - a_1 * v_2 * v_2 * v_4 + v_3 - v_4 \tag{2.15}$$

$$v_1 := a_1 * v_2 * v_2 * (v_3 - v_4) + v_3 - v_4 \tag{2.16}$$

ここで、新たな中間変数v_5を導入して、式(2.16)を式(2.17)に示すように2つの積和式に分解する。当初、式(2.15)には6回の乗算と3回の加減算が含まれていたが、式(2.17)では乗算が3回に減少している。

$$\left[\begin{array}{lcl} v_5 & := & v_3 - v_4 \\ v_1 & := & a_1 * v_2 * v_2 * v_5 + v_3 - v_4 \end{array}\right. \tag{2.17}$$

このように、P_I個$(P_I \geq 2)$の積項に含まれる要素数S_I個$(S_I \geq 1)$の共通因子を括り出すことにより削減できる乗算回数、すなわち「部分的因数分解」の効果E_Iは、次の式(2.18)によって評価できる。

$$E_I := S_I * (P_I - 1) \tag{2.18}$$

例えば、上記の式(2.15)に含まれる共通因子$(a_1 * v_2 * v_2)$に対して、式(2.18)の効果E_Iを求めると、$S_I = 3$, $P_I = 2$より$E_I = 3$である。

従って、「部分的因数分解」では、1つの積和式に含まれる幾つかの共通因子の中から、式(2.18)で定義した効果E_I $(E_I \geq 1)$が最大となる共通因子を選ぶために、以下に示すようなヒューリスティックなアルゴリズムを用いている。

【部分的因数分解における共通因子の選択方法】

(1) 最も多くの積項に含まれる要素を、1つ選択して因子とする。
(2) 現時点の因子の要素数S_Iと、それを含む積項の数P_Iから効果E_Iを評価する。
(3) 因子を含む積項に最も多く含まれる要素を、新たに因子に加える。
(4) 該当する要素が無くなるまで、上記(2), (3)の処理を繰り返す。
(5) 評価された因子の中で、簡単化の効果E_Iが最大の因子を選択する。 ■

ただし、効果E_Iの等しい因子が複数ある場合は、それらの因子が含まれる積項の要素数を考慮して選択する。その他、アルゴリズムの詳細は付録Bを参照されたい。

▼ 共通部分式の括り出し

式(2.14)において、1つ以上の積項の和を部分式と呼ぶ。この簡単化手法は、複数の積和式に含まれる共通な部分式を取り出すことで、冗長な加減算を削減する。

例えば、式(2.17)に示した2つの積和式から共通な部分式を取り出して、その部分式により新たな中間変数v_6を定義すると、式(2.17)は式(2.19)のように変換できる。式(2.17)と比べて、式(2.19)では減算が1回少なく成っている。

$$\left[\begin{array}{lcl} v_6 & := & v_3 - v_4 \\ v_5 & := & v_6 \\ v_1 & := & a_1 * v_2 * *2 * v_5 + v_6 \end{array}\right. \tag{2.19}$$

このように、P_2個$(P_2 \geq 2)$の異なる積和式に含まれるS_2個$(S_2 \geq 2)$の積項から成る共通な部分式を、1つにまとめることで削減できる加減算の回数、すなわち、「共通部分式の括り出し」の効果E_2は、次の式(2.20)によって評価できる。

$$E_2 := (S_2 - 1) * (P_2 - 1) \tag{2.20}$$

例えば、上記の式(2.17)の共通部分式$(v_3 - v_4)$に対して、式(2.20)の効果E_2を求めると、$S_2 = 2$, $P_2 = 2$より$E_2 = 1$である。

従って、「共通部分式の括り出し」では、複数の積和式に含まれる共通部分式の中から、式(2.20)で定義した効果E_2 $(E_2 \geq 1)$が最大となる共通部分式を選ぶために、以下に示すようなヒューリスティックなアルゴリズムを用いている。

【共通部分式の括り出しにおける共通部分式の選択方法】

(1) 最も多くの積和式に含まれる積項を、1つ選択して部分式とする。
(2) 現時点の部分式の積項数S_2と、それを含む積和式の数P_2から効果E_2を評価する。
(3) 部分式を含む積和式に最も多く含まれる積項を、新たに部分式に加える。
(4) 該当する積項が無くなるまで、上記(2), (3)の処理を繰り返す。
(5) 評価された部分式の中で、簡単化の効果E_2が最大の部分式を選択する。 ■

▼ 共通因子の括り出し

この簡単化手法は、異なる積和式の積項に含まれる共通な因子を括り出すことで、冗長な乗算を削減する。

例えば、次の式(2.21)は、前述の「部分的因数分解」、あるいは、「共通部分式の括り出し」を用いても、これ以上には簡単化できない。

$$\left[\begin{array}{lcl} v_7 & := & a_1 * v_2 * v_4 + v_3 * v_6 \\ v_1 & := & a_1 * v_2 * v_5 + v_3 * v_4 \end{array}\right. \tag{2.21}$$

しかし、2つの積和式に含まれる共通因子$(a_1 * v_2)$を括り出すと、次式に示すように、

乗算回数をさらに1回削減することができる。

$$\left[\begin{array}{lcl} v_8 & := & a_1 * v_2 \\ v_7 & := & v_4 * v_8 + v_3 * v_6 \\ v_1 & := & v_5 * v_8 + v_3 * v_4 \end{array}\right. \tag{2.22}$$

このように、異なる積和式に属する P_3 ($P_3 \geq 2$) 個の積項に含まれる、要素数 S_3 個 ($S_3 \geq 2$) の共通因子を1つにまとめることで削減できる乗算回数、すなわち、「共通因子の括り出し」の効果 E_3 は、次の式 (2.23) によって評価できる。

$$E_3 := (S_3 - 1) * (P_3 - 1) \tag{2.23}$$

例えば、上記の式 (2.21) における共通因子 $(a_1 * v_2)$ に対して、式 (2.23) の効果 E_3 を求めると、$S_3 = 2$, $P_3 = 2$ より $E_3 = 1$ である。

従って、「共通因子の括り出し」では、異なる積和式に含まれる幾つかの共通因子の中から、式 (2.23) で定義した効果 E_3 ($E_3 \geq 1$) が最大となる共通因子を選ぶために、以下に示すようなヒューリスティックなアルゴリズムを用いている。

【共通因子の括り出しにおける共通因子の選択方法】

(1) 最も多くの積項に含まれる要素を、1つ選択して因子とする。
(2) 現時点の因子の要素数 S_3 と、それを含む積項の数 P_3 から効果 E_3 を評価する。
(3) 因子を含む積項に最も多く含まれる要素を、新たに因子に加える。
(4) 該当する要素が無くなるまで、上記 (2), (3) の処理を繰り返す。
(5) 評価された因子の中で、簡単化の効果 E_3 が最大の因子を選択する。 ■

▼ 定数の畳み込み

この簡単化手法は、定数同士の演算を予め計算することで、オンラインで実行される演算回数を削減する。「定数の畳み込み」には、乗算と加減算に関するものがある。

例えば、次の式 (2.24) は式 (2.25) のように変換される。式 (2.25) において、定数 a_5 と a_6 の値は、定数 $a_1 \sim a_4$ の値が既知であれば、前もって求めることができる。

$$v_1 := a_1 * a_2 * v_2 + a_3 + a_4 \tag{2.24}$$

$$v_1 := a_5 * v_2 + a_6 \tag{2.25}$$

$$\left[\begin{array}{lcl} a_6 & := & a_3 + a_4 \\ a_5 & := & a_1 * a_2 \end{array}\right.$$

▼ 代入式の削除

この簡単化手法は、代入式を削除することで冗長な中間変数を消去する。数式に含まれる演算回数を減少させることはできないが、不要な中間変数の増加を抑制し、ほかの簡単化手法の効果を高める。

例えば、さきに「共通部分式の括り出し」で得られた式(2.19)は、最後に導入された中間変数v_6が消去されて、次の式(2.26)のように変換される。

$$\left[\begin{array}{lcl} v_5 & := & v_3 - v_4 \\ v_1 & := & a_1 * v_2 * *2 * v_5 + v_5 \end{array}\right. \tag{2.26}$$

2.3.2 大域的な簡単化手法

これまでに述べた5つの局所的な簡単化手法を組み合わせて、数式の簡単化の全体的な処理の流れを定めた「大域的な簡単化手法」を提案する。局所的な簡単化手法の特徴は、ある簡単化手法を用いた数式の改良が、別の簡単化手法による新たな改良を可能にすることである。従って、提案する「大域的な簡単化手法」では、すべての積和式に対して、局所的な簡単化手法を繰り返し何度も適用する。

【大域的な簡単化手法】

(1) 全積和式に対して、「定数の畳み込み」を適用する。
(2) 全積和式に対して、「共通部分式の括り出し」を適用する。
(3) 各積和式に対して、「部分的因数分解」を適用する。
(4) 数式が改良される限り、(2), (3)の処理を繰り返す。
(5) 全積和式に対して、「共通因子の括り出し」を適用する。

ただし、「定数の畳み込み」と「代入式の削除」は、ほかの局所的な簡単化手法の効果を阻害するような副作用がないため、各処理による数式の変換の後で、適用可能であれば直ちに実行する。 ■

上記の「大域的な簡単化手法」では、一般にロボット制御則は共通な部分式を多く含むことを考慮して、最初に「共通部分式の括り出し」を適用している。次に、得られる結果が局所的な最適解に陥らないように配慮して、「部分的因数分解」と「共通部分式の括り出し」を交互に適用する。また、「共通因子の括り出し」は、「部分的因数分解」に比べて簡単化の効果が小さく、「部分的因数分解」の有効な適用を妨げる恐れがあるので、最後に適用している。

提案した「大域的な簡単化手法」は、ヒューリスティックな手法を採用しているために、必ずしも大域的な最適解が得られるとは限らないが、数式処理システムを用いた項の書き換えによる数式の簡単化とは異なり、演算回数が削減可能なときのみ数式を変換するため、アルゴリズムの停止性は保証されている。

例えば、前節の式(2.11)に示した2自由度のロボットの運動方程式に対して、提案し

た「大域的な簡単化手法」を適用した結果を以下に示す。

$$\left[\begin{array}{lcl} v_6 & := & ddq_1 + ddq_2 \\ v_5 & := & c_2 * ddq_1 + s_2 * dq_1 * *2 \\ v_4 & := & c_2 * c_1 - s_1 * s_2 \\ v_3 & := & -a_2 * dq_2 - a_1 * dq_1 \\ v_2 & := & a_1 * c_1 + a_4 \\ v_1 & := & a_3 * v_4 + a_6 * v_6 \\ f_2 & := & a_2 * v_5 + v_1 \\ f_1 & := & v_2 * ddq_1 + dq_2 * s_2 * v_3 + a_5 * c_1 + a_2 * c_2 * ddq_2 + v_1 \end{array}\right. \tag{2.27}$$

ただし、「定数の畳み込み」により得られた定数は、以下の通りである。

$$\left[\begin{array}{lcl} a_1 & := & 2 * m_2 * r_2 * L_1 \\ a_2 & := & m_2 * r_2 * L_1 \\ a_3 & := & g * m_2 * r_2 \\ a_4 & := & N_1 + m_1 * r_1 * *2 + m_2 * L_1 * *2 \\ a_5 & := & g * m_1 * r_1 + g * m_2 * L_1 \\ a_6 & := & N_2 + m_2 * r_2 * *2 \end{array}\right. \tag{2.28}$$

当初、式(2.11)には三角関数の計算を除き、74回の乗算と21回の加減算が含まれていたが、それを簡単化した式(2.27)では、乗算17回、加減算11回と演算回数が大幅に減少している。ただし、式(2.28)に示した各定数の計算は、予めオフラインで実行できるため、上記の演算回数には含めていない。

2.4 最適化コンパイラの開発

前節で提案した数式の簡単化手法を応用して、任意のロボット制御則から、その効率的な計算プログラムを自動生成する最適化コンパイラを開発した[45]。この最適化コンパイラにより生成される計算プログラム（オブジェクト・プログラム）は、それに含まれる演算回数が、与えられたロボット制御則（ソース・プログラム）に含まれる演算回数よりも少ないという意味で、改良（最適化）されたものである。伝達関数など単純な制御則の計算プログラムを、自動生成するようなシステムは、既に実用化されている[46]。しかし、これらのシステムでは、ロボット制御則のように、三角関数を含む複雑で長大な制御則に対して、効率的な計算プログラムを生成できない。

2.4.1 最適化コンパイラにおける処理

開発した最適化コンパイラの処理工程は、図2.3に示すような3つの副工程（フェーズ）から成る。最適化コンパイラのソース・プログラムは、数式処理システムを用いて導出したロボット制御則である[35]–[39]。このロボット制御則は、積和式のみならず、括弧も認めた一般的なスカラの式とする。これにより、ロボット制御則に含まれる明らかに重複する計算を、意図的にまとめて導出することもできる。

最初の構文解析フェーズでは、ソース・プログラムから読み込んだロボット制御則を、数式の簡単化に適した連立の積和式に分解する。次の簡単化フェーズでは、前節で提案した数式の簡単化手法を用いて、ロボット制御則から冗長な演算を削除する。最後のコー

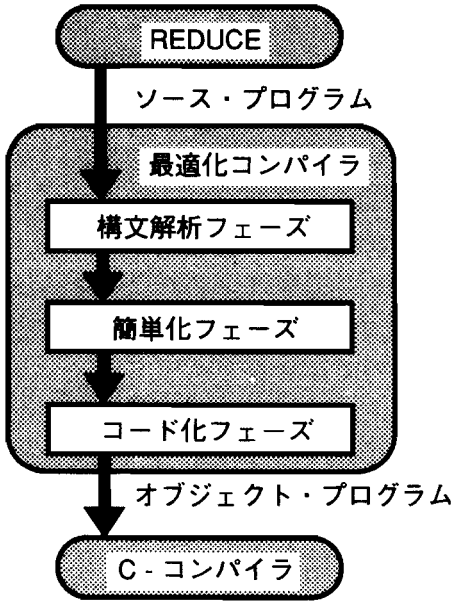


図2. 3　最適化コンパイラにおける処理工程

ド化フェーズでは、簡単化されたロボット制御則を、C言語により記述された計算プログラムに変換する。C言語は移植性に優れたプログラム言語であり、組み込み制御システムのプログラム開発にも広く利用されている[47]。従って、最適化コンパイラのオブジェクト・プログラムであるC言語の計算プログラムを、適当なCコンパイラによって再度コンパイルすることにより、多くの計算機システムで実行可能なロボット制御則の計算プログラムを、容易に得ることができる。

以下では、最適化コンパイラにおける各フェーズの具体的な処理の内容と、その実現方法について、使用するデータ構造も含めて説明する。

▼ 構文解析フェーズ

構文解析フェーズでは、ロボット制御則を連立の積和式に分解する。また、ロボット制御則に含まれる定数と変数の判別のほか、共通な三角関数などを削除する。

初めに、ソース・プログラムとして与えられるロボット制御則の構文法を、BNF記法（Backus-Naur Form）[44]によって、以下のように定義する。

```
数式  ->  変数 := 式$
式    ->  項 | - 項 | 式 + 項 | 式 − 項
項    ->  因子 | 項*因子 | 項/因子
因子  ->  要素 | 関数 | (式) | 因子**整数
関数  ->  sin(式) | cos(式) | tan(式)
要素  ->  定数 | 変数
```

ただし、$は1つの数式の終わりを表す記号である。

構文法の右辺のみに現れる「定数」、「変数」、演算子などは終端記号と呼ぶ。これらは、ソース・プログラムから文字列や記号として読み込まれ、字句解析によって認識される。一方、「項」や「因子」など構文法の両辺に現れる記号は非終端記号と呼び、その右辺の終端記号と非終端記号によって再帰的に定義される。

構文解析の手順としては、与えられたロボット制御則を左から右へ検索して、字句解析により認識された終端記号を記憶してゆく。ここで、これらの終端記号から構文法の定義に基づき非終端記号が認識されると、各非終端記号に付随する所定の処理を実行して、その認識に寄与した右辺の終端記号（および非終端記号）を、新たに認識された左辺の非終端記号で置き換えるという処理を繰り返す。上記の構文法では、ロボット制御則から、積項、積和式、および、三角関数を認識して、それらを最適化コンパイラ内部の各データ構造へ格納するために非終端記号が決められている。また、構文解析に際して、べき乗は乗算に展開され、除算は逆数の乗算に置き換えられる。

例えば、さきに図2.1で示した2自由度のロボットの運動方程式は、次の式(2.29)に示すように括弧を含めた形でも導出できる。式(2.29)では、数式処理システムを用いた項の書き換えにより、三角公式に基づく三角関数の簡単化を行っている。

$$\left[\begin{array}{lcl} f_1 & := & (b_1+b_3+2*L_1*b_4*\cos(q_2))*ddq_1+(b_3+L_1*b_4*\cos(q_2)) \\ & & *ddq_2-2*L_1*b_4*\sin(q_2)*dq_1*dq_2+L_1*b_4*\sin(q_2) \\ & & *dq_2**2+g*b_2*\cos(q_1)+g*b_4*\cos(q_1+q_2)\$ \\ f_2 & := & (b_3+L_1*b_4*\cos(q_2))*ddq_1+b_3*ddq_2 \\ & & +L_1*b_4*\sin(q_2)*dq_1**2+g*b_4*\cos(q_1+q_2)\$ \end{array}\right. \tag{2.29}$$

式 (2.29) のロボット制御則を、構文解析フェーズで処理すると、次の式 (2.30) に示すような連立の積和式に分解されて、数式中の括弧が除かれる。さらに、式 (2.30) では、共通な三角関数の計算が 1 つの数式として括り出され、積和式の中では、適当な中間変数 v_m に置き換えられていることに注意されたい。

$$\left[\begin{array}{lcl} v_1 & := & q_1+q_2\$ \\ v_2 & := & \cos(q_2)\$ \\ v_3 & := & b_1+b_3+2*L_1*b_4*v_2\$ \\ v_4 & := & L_1*b_4*v_2+b_3\$ \\ v_5 & := & \sin(q_2)\$ \\ v_6 & := & \cos(q_1)\$ \\ v_7 & := & \cos(v_1)\$ \\ v_8 & := & g*b_4*v_7\$ \\ f_1 & := & ddq_1*v_3+ddq_2*v_6-2*L_1*b_4*dq_2*dq_1*v_5 \\ & & +L_1*b_4*dq_2*dq_2*v_5+g*b_2*v_6+v_8\$ \\ f_2 & := & ddq_1*v_4+b_3*ddq_2+L_1*b_4*dq_1*dq_1*v_5+v_8\$ \end{array}\right. \tag{2.30}$$

ただし、v_m $(m=1,\ldots,8)$ は、自動的に導入される中間変数である。

最適化コンパイラに読み込まれたロボット制御則は、構文解析フェーズにおいて、幾つかの「表」の形のデータ構造へ格納される。

まず、式 (2.30) に示した数式のうち、積和式は積項単位に分割されて、表 2.2 に示す「項の表」と、表 2.3 に示す「式の表」に格納される。表 2.2 に示した「項の表」において、列は前述の構文法で定義された「項」(積項 T_j) を表し、行は各積項 T_j に含まれる要素 (定数および変数) を表す。「項の表」には、構文解析の過程で、非終端記号「項」が認識されるごとに、新たな積項が登録される。一方、表 2.3 に示した「式の表」において、列は構文法で定義された「式」(積和式) を表し、行は各積和式に含まれる積項 T_j を表す。「式の表」には、構文解析の過程で、非終端記号「式」が認識されるごとに、新たな積和式と、それにより定義される中間変数 v_m が登録される。

次に、式 (2.30) に示した数式のうち、三角関数の計算式は、表 2.4 に示す「関数の表」に格納される。「関数の表」において、列は三角関数で定義される中間変数を表し、行は三角関数の引数となる変数を表す。「関数の表」には、構文解析の過程で、非終端記号「関数」が認識されるごとに、新たな三角関数が登録される。この際、ロボット制御則に含まれる共通な三角関数の計算が除かれる。

表2. 2　項の表

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}
sign	+	+	+	+	+	+	+	+	+	−	+	+	+	+	+	+
2					1					1						
g							1					1				
L_1					1	1				1	1					1
b_1			1													
b_2												1				
b_3				1											1	
b_4					1	1	1			1	1					1
q_1	1															
q_2		1														
dq_1										1						2
dq_2										1	2					
ddq_1								1						1		
ddq_2									1						1	
v_2					1	1										
v_3								1								
v_4														1		
v_5										1	1					1
v_6									1			1				
v_7							1									
v_8													1			

表2．3 式の表

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}
v_1	1	1														
v_3			1	1	1											
v_4				1		1										
v_8							1									
f_1								1	1	1	1	1	1			
f_2													1	1	1	1

表2．4 関数の表

	q_1	q_2	v_1
v_2		cos	
v_5		sin	
v_6	cos		
v_7			cos

▼ 簡単化フェーズ

簡単化フェーズでは、構文解析フェーズで作成された3種類の表のうち、積和式を格納した2つの表、すなわち、「項の表」と「式の表」を書き換えることで、前節で述べた5つの局所的な簡単化手法、すなわち、「部分的因数分解」、「共通部分式の括り出し」、「共通因子の括り出し」、「定数の畳み込み」、「代入式の削除」を実現する。また、簡単化フェーズでは、「定数の畳み込み」によって定義される定数の計算式を格納するために、新たに「定数項の表」と「定数式の表」を作成する。

以下では、これらの局所的な簡単化手法を実現するための、「関数の表」を除く各「表」に対する具体的な操作手順について説明する。

【部分的因数分解】

(1) 「項の表」で、1つの積和式に含まれる積項から、括り出す共通因子を選択する。
(2) 上記の各積項を構成する要素から共通因子を除いて、新たな積項 T_j $(j \in J)$ を定義し、これらを「項の表」へ登録する。
(3) 新たな積和式 $v_m := \sum T_j$ $(j \in J)$ を定義して「式の表」へ登録する。
(4) 共通因子と中間変数 v_m の積より、新たな積項 T_k を定義して「項の表」へ登録する。
(5) 「式の表」で、手順(1)の積和式から、共通因子を含む積項を消去すると共に、この積和式に新たな積項 T_k を加える。 ■

【共通部分式の括り出し】
(1)　「式の表」で、括り出す共通部分式を構成する積項T_j $(j \in J)$を選択する。
(2)　新たな積和式$v_m := \sum T_j$ $(j \in J)$を定義して「式の表」へ登録する。
(3)　新たな積項$T_k := v_m$を定義して「項の表」へ登録する。
(4)　「式の表」で、共通部分式を含む全ての積和式から、それを構成する積項T_j $(j \in J)$を消去すると共に、これらの積和式に新たな積項T_kを加える。 ■

【共通因子の括り出し】
(1)　「項の表」で、括り出す共通因子を選択する。
(2)　選択した共通因子より、新たな積項T_kを定義して「項の表」へ登録する。
(3)　新たな積和式$v_m := T_k$を定義して「式の表」へ登録する。
(4)　「項の表」で、共通因子を含む全ての積項から、共通因子を消去すると共に、これらの積項に中間変数v_mを乗ずる。 ■

「定数の畳み込み」は、前節で述べたように、「乗算に対する定数の畳み込み」と「加減算に対する定数の畳み込み」に大別される。また、「定数の畳み込み」により定義される定数の計算式は、「定数項の表」と「定数式の表」に格納される。

【乗算に対する定数の畳み込み】
(1)　「項の表」で、2個以上の定数から成る定数因子を選択する。
(2)　選択した定数因子より、新たな定数項T_kを定義して「定数項の表」へ登録する。
(3)　新たな定数の計算式$a_k := T_k$を定義して「定数式の表」へ登録する。
(4)　「項の表」で、定数因子を含む全ての積項から、定数因子を消去すると共に、これらの積項に定数a_kを乗ずる。 ■

【加減算に対する定数の畳み込み】
(1)　「項の表」で、1つの積和式に含まれる2個以上の定数項T_j $(j \in J)$を選択する。
(2)　新たな定数の計算式$a_k := \sum T_j$ $(j \in J)$を定義して「定数式の表」へ登録する。
(3)　新たな定数項$T_k := a_k$を定義して「定数項の表」へ登録する。
(4)　「式の表」において、手順(1)の積和式から、定数項T_j $(j \in J)$を消去すると共に、この積和式に新たな定数項T_kを加える。 ■

【代入式の削除】
(1)　「式の表」と「項の表」より、代入式$v_k := v_m$、または、$v_m := a_k$を検索する。
(2)　代入式の右辺が中間変数v_mである場合、「式の表」と「項の表」で、右辺の中間変数v_mを、全て左辺の中間変数v_kによって置き換える。
(3)　代入式の右辺が定数a_kである場合、「項の表」で、左辺の中間変数v_mを、全て右辺の定数a_kによって置き換える。
(4)　「式の表」と「項の表」より、代入式と中間変数v_mを削除する。 ■

▼ コード化フェーズ

コード化フェーズは、簡単化フェーズにおいて簡単化されたロボット制御則から、その効率的な計算プログラムを生成する。

まず、「項の表」、「式の表」、「関数の表」から数式（変数の計算式）のデータ依存関係を調べて、それらを実行可能な順序に並び換える。このような解析は、出力変数を定義している計算式から、その右辺に含まれる中間変数を上記の「表」において順番に辿ることで行える。同様に、「定数項の表」と「定数式の表」から、数式（定数の計算式）のデータ依存関係を調べて、それらを実行可能な順序に並び換える。

次に、実行可能な順序に並べられた計算式を、C言語の計算プログラム（オブジェクト・プログラム）に変換する。このC言語のプログラムは、オフラインで1度だけ実行される定数の計算式から成る部分と、オンラインで繰り返し実行される変数の計算式から成る部分より構成される。

ところで、将来における最適化コンパイラのアルゴリズムの改良や、機能の拡張に伴うプログラムの変更に際して、懸念される最も致命的なバグは、最適化コンパイラによる数式の変換により、ソース・プログラムの意味がオブジェクト・プログラムにおいて失われることである。そこで、最適化コンパイラは、自分自身に対するディバッグ機能として、数式処理言語（REDUCE）によって記述された計算プログラムを、オブジェクト・プログラムとして生成することも可能である。そこで、数式処理システムREDUCEを用いて、オブジェクト・プログラムからソース・プログラム（ロボット制御則）を復元することにより、最適化コンパイラに入力された数式と、それを簡単化して得られた数式が等価であることを容易に検証できる。

式(2.29)に示した2自由度のロボットの運動方程式をソース・プログラムとして、最適化コンパイラにより得られたオブジェクト・プログラム（REDUCE形式）を、次の式(2.31)と式(2.32)に示す。式(2.31)はオフラインで実行される定数の計算式、式(2.32)はオンフラインで実行される変数の計算式である。当初、式(2.29)には、乗算30回，加減算14回，三角関数9回が含まれていたが、オンラインで計算される式(2.32)では、乗算15回，加減算11回，三角関数4回に削減されている。

$$\left[\begin{array}{lcl} a_1 & := & b_1 + b_3\$ \\ a_2 & := & 2 * b_4 * L_1\$ \\ a_3 & := & L_1 * b_4\$ \\ a_4 & := & g * b_2\$ \\ a_5 & := & g * b_4\$ \end{array}\right. \tag{2.31}$$

$$
\left[\begin{array}{lcl}
v_1 & := & q_2 + q_1\$ \\
v_2 & := & -a_2 * dq_1 + a_3 * dq_2\$ \\
v_3 & := & \cos(q_2)\$ \\
v_4 & := & a_2 * v_3 + a_1\$ \\
v_5 & := & b_3 + a_3 * v_3\$ \\
v_6 & := & \sin(q_2)\$ \\
v_7 & := & \cos(q_1)\$ \\
v_8 & := & \cos(v_1)\$ \\
v_9 & := & a_5 * v_8\$ \\
f_1 & := & ddq_1 * v_4 + ddq_2 * v_5 + a_4 * v_7 + v_9 + dq_2 * v_2 * v_6\$ \\
f_2 & := & ddq_1 * v_5 + b_3 * ddq_2 + a_3 * dq_1 * dq_1 * v_6 + v_9\$
\end{array}\right. \tag{2.32}
$$

2.4.2 オブジェクト指向による実現

最適化コンパイラは、近年、新しいソフトウェアの設計手法として注目されているオブジェクト指向に基づき実現している。オブジェクト指向とは、技術的にはデータ構造と操作（メソッド）を一体化したオブジェクトの集まりとして、ソフトウェアを組織化することである。オブジェクト指向によれば、同一の概念的なシステム・モデルのもとで、システムの分析、設計、実装を一貫して行うことができ、システムの開発のみならず変更や保守も容易となる.

現在、オブジェクト指向による設計手法は、数多く提案されているが[51]、これらの中でも、図表による記述能力に優れたOMT（Object Modeling Technique）手法[52]を参考とする。OMT手法ではシステムを記述するために、幾つかのモデルを提案している。すなわち、システムを構成するオブジェクト間の静的な関係を表すオブジェクトモデル、時間とともに変化するシステムの状態と制御を表す動的モデル、システム内におけるデータの変換を表す機能モデルである。ここで、最適化コンパイラは、バッチ処理によるデータ変換を中心としたシステムであるため、機能モデルを利用する。

最適化コンパイラの機能モデルを図2.4に示す. 機能モデルにおいて、矢印はデータの流れを、楕円はデータを変換するプロセスを、長方形はデータを格納するデータ・ストアを表している。図2.4の各プロセスは、最適化コンパイラのフェーズに対応しており、それらの実行順序は、さきに図2.3に示した通りである。まず、ソース・プログラムのロボット制御則は、構文解析により「項の表」、「式の表」、「関数の表」に分割されて格納される。次に、数式の簡単化により「項の表」と「式の表」が書き換えられると共に、新たに「定数項の表」と「定数式の表」が作られる。最後に、コード化によって、上記の全ての「表」のデータからオブジェクト・プログラムが合成される。

機能モデルに基づいてシステムを実装する場合、プロセスはオブジェクトのメソッドに、データ・ストアはオブジェクトのデータ構造に対応づけられる。図2.4の機能モデルには、幾つもの「表」が存在するが、これらは汎用的なクラスのオブジェクトとして実現されている。クラスとは、オブジェクトを生成するための雛形であり、オブジェクト

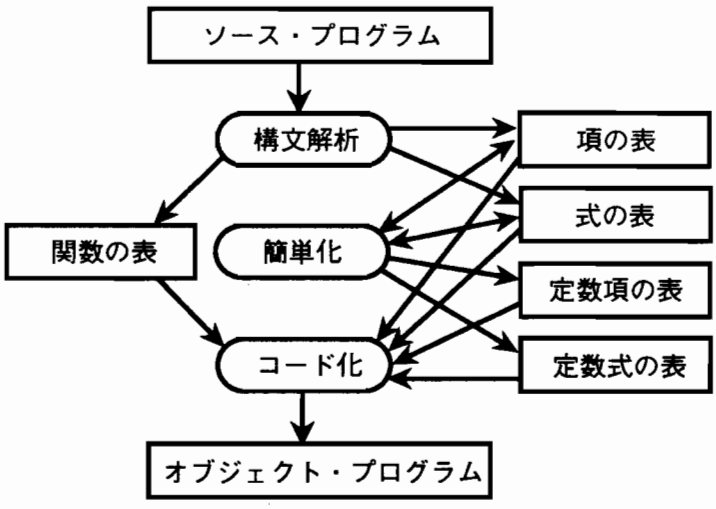


図2. 4 最適化コンパイラの機能モデル

のデータ構造とメソッドを定義している。従って、1つのクラスからは、同じデータ構造とメソッドを持つオブジェクトが無数に生成できる。このように、汎用的なクラスを再利用して、プログラムが効率的に作成できることも、オブジェクト指向によるシステム開発の実践的な利点の1つである [50]。

ところで、表2.2と表2.3に示した「項の表」と「式の表」からも推察できるように、ロボット制御則を記述する数式は膨大であるため、これらの数式を格納する各種の「表」はスパースで空欄が多くなる。従って、「表」のデータ構造には、計算機のメモリを有効に活用すると共に、プログラムの実行効率を高めるための工夫が必要である。そこで、上記のクラスが提供する「表」のデータ構造は、2次元の多重リスト構造 [54] によって実現している。このクラスを「マジック・テーブル」と呼び、その仕様と実装の詳細は付録Cに示す。

最適化コンパイラの実装には、計算機にSun SPARCワークステーションを、プログラミングにはオブジェクト指向言語の1つであるGNU C++[53] を採用した。最適化コンパイラのプログラムの主要部分を、付録Dとして添付する。

2.5 適用例による評価

開発した最適化コンパイラを用いて、幾つかのロボット制御則の計算プログラムを作成することにより、提案した数式の簡単化手法の有効性を評価する。

ロボットは、溶接や組み立て作業など産業用として、最も一般的な機構を有する多関節ロボット・アームと、極座標ロボット・アームとする [1]。また、ロボット制御則には、2．2節で紹介した逆動力学計算と線形化補償器を取り上げる。さらに、最適化コンパイラの評価方法としては、数式処理システム REDUCE を用いて導出したソース・プログラム（ロボット制御則）に含まれる演算回数と、オブジェクト・プログラム（計算プログラム）に含まれる演算回数を比較する。

▼ 例題1：多関節ロボット・アームの逆動力学計算

図2.5に示すような3つの回転関節を持つ多関節ロボット・アーム [1] を考える。このロボットのD-Hパラメータを表2.5に示す。図2.5に示した各関節変数q_nは、表2.5において$q_1 = \theta_1$，$q_2 = \theta_2$，$q_3 = \theta_3$ のように対応する。

図2.5の多関節ロボット・アームに対して、2．2節の式(2.3)に示したロボット制御則（逆動力学計算）を、数式処理システム REDUCE を用いて積和式の形で導出した。さらに、このロボット制御則をソース・プログラムとして、最適化コンパイラによってオブジェクト・プログラムを作成した。これらのソース・プログラムとオブジェクト・プログラムに含まれる演算回数を表2.6に示す。また、実際のプログラムを、付録Eに示す。

表2.6において、2つのプログラムに含まれる演算回数を比較すると、オブジェクト・プログラムの演算回数は、ソース・プログラムに比べて大幅に削減されており、提案した数式の簡単化手法の有効性が確認できる。

ところで、逆動力学計算については、幾つかの高速計算アルゴリズムが提案されている。ロボットの自由度Nに対して、代表的な逆動力学計算の高速計算アルゴリズムに含

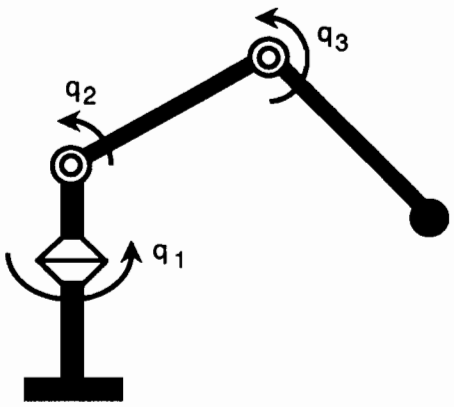


図2．5　多関節ロボット・アーム

まれる演算回数を表2.7に示す[18]。例えば、ニュートン・オイラー法[6]では、N自由度のロボットに対して、$(150N-48)$回の乗算と$(131N-48)$回の加減算を要し、$N=3$の場合には乗算402回、加減算345回となる。一方、表2.6に示した3自由度のロボット・アームに対する逆動力学計算のための計算プログラム（オブジェクト・プログラム）に含まれる演算回数は、乗算55回、加減算40回であり、明らかに、従来の高速計算アルゴリズムよりも優れている。

表2.5 多関節ロボット・アームのD-Hパラメータ

座標系	α_n	θ_n	d_n	a_n
1	$-\pi/2$	θ_1	0	0
2	0	θ_2	0	a_2
3	$\pi/2$	θ_3	d_3	a_3

表2.6 各プログラムに含まれる演算回数

program	$*$	$\div$	$\pm$	cos	sin
source-1	253	0	54	39	59
object-1	55	0	40	2	2

表2.7 高速計算アルゴリズムに含まれる演算回数

algorithm	$*$	$\pm$
Lagrange	$16N^4+215N^3/6$ $+171N^2/4+53N/3-128$	$25N^4+22N^3$ $+129N^2/2+14N-93$
Hollerbach	$412N-277$	$320N-201$
Newton-Euler	$150N-48$	$131N-48$

▼ 例題2：極座標ロボット・アームの逆動力学計算

図2.6に示すような2つの回転関節と1つの直動関節を持つ極座標ロボット・アーム[1]を考える。このロボットのD-Hパラメータを表2.8に示す。図2.6に示した各関節変数q_nは、表2.8において$q_1 = \theta_1$，$q_2 = \theta_2$，$q_3 = d_3$のように対応する。

図2.6の極座標ロボット・アームに対して、2．2節の式(2.3)に示したロボット制御則（逆動力学計算）を、数式処理システムREDUCEを用いて積和式の形で導出した。さらに、このロボット制御則をソース・プログラムとして、最適化コンパイラによってオブジェクト・プログラムを作成した。これらのソース・プログラムとオブジェクト・プログラムに含まれる演算回数を表2.9に示す。また、実際のプログラムを、付録Eに示す。

表2.9において、2つのプログラムに含まれる演算回数を比較すると、オブジェクト・プログラムでは演算回数が大幅に削減されている。

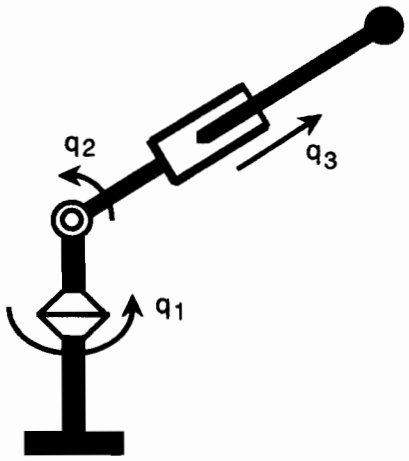


図2．6　極座標ロボット・アーム

表2．8　極座標ロボット・アームのD-Hパラメータ

座標系	α_n	θ_n	d_n	a_n
1	$-\pi/2$	θ_1	0	0
2	$\pi/2$	θ_2	d_2	0
3	0	0	d_3	0

表2．9　各プログラムに含まれる演算回数

program	*	÷	±	cos	sin
source-2	209	1	50	9	34
object-2	49	1	31	2	3

▼ 例題３：多関節ロボット・アームの線形化補償器

例題１の多関節ロボット・アームに対して、２．２節の式(2.5)に示したロボット制御則（線形化補償器）を、数式処理システムREDUCEを用いて積和式の形で導出した。さらに、このロボット制御則をソース・プログラムとして、最適化コンパイラによってオブジェクト・プログラムを作成した。これらソース・プログラムとオブジェクト・プログラムに含まれる演算回数を表2.10に示す。また、実際のプログラムを付録Eに示す。

表2.10において、２つのプログラムに含まれる演算回数を比較すると、オブジェクト・プログラムでは演算回数が大幅に削減されている。

表2. 10　各プログラムに含まれる演算回数

program	*	÷	±	cos	sin
source-3	329	0	68	52	81
object-3	70	0	55	2	2

▼ 例題４：極座標ロボット・アームの線形化補償器

例題３の極座標ロボット・アームに対して、２．２節の式(2.5)に示したロボット制御則（線形化補償器）を、数式処理システムREDUCEを用いて積和式の形で導出した。さらに、このロボット制御則をソース・プログラムとして、最適化コンパイラによってオブジェクト・プログラムを作成した。これらソース・プログラムとオブジェクト・プログラムに含まれる演算回数を表2.11に示す。また、実際のプログラムを付録Eに示す。

表2.11において、２つのプログラムに含まれる演算回数を比較すると、オブジェクト・プログラムでは演算回数が大幅に削減されている。さらに、例題１～３の結果についても考慮すると、提案した数式の簡単化手法が、対象とするロボットの幾何学的な構造の違いや、ロボット制御則の種類に関わらず有効であることが確認できる。

表2. 11　各プログラムに含まれる演算回数

program	*	÷	±	cos	sin
source-4	283	2	61	10	42
object-4	53	1	38	2	3

2.6 結言

この章では、多変数の積和式として与えられるロボット制御則から、数式の簡単化によって冗長な演算を除くことにより、ロボット制御則の効率的な計算公式をシステマチックに生成する手法を提案した。さらに、提案した数式の簡単化手法を応用して、数式処理システムを用いて導出したロボット制御則から、その効率的な計算プログラムを自動生成することができる最適化コンパイラを開発した。

従来のロボット制御則に固有の高速計算アルゴリズムと比較して、提案した手法を用いてロボット制御則の計算プログラムを自動生成することの利点としては、任意のロボット制御則に対して適用可能であること、ロボット制御則を記述する数式の特性に応じて適切な中間変数が機械的に決定されるため、制御則の種類のみならず、ロボットの幾何学的な構造の違いなども反映された、いわばオーダーメードの効率的な計算プログラムが得られることなどが挙げられる。さらに、自動生成されたロボット制御則の計算プログラムは、高級言語を用いて人手により作成された計算プログラムよりも、実行効率が良く、信頼性も高いことが期待できる。

最後に、開発した最適化コンパイラを用いて、幾つかのロボット制御則の計算プログラムを作成することにより、提案した手法の有効性と実用性を確認した。

第3章

ロボット制御則の並列処理

3.1 緒言

ロボット・アームに対して、その非線形な動力学特性を補償するような動的制御を行うためには、ロボット・アームの運動方程式に基づく非線形な下位制御則の計算を、実時間で行うことが必要となる [2]。この問題に対するハードウェア的な解決策として、複数のプロセッサを用いた並列処理がある [5]。この分野における従来の代表的な研究は、シストリックアレイを用いた逆動力学計算の並列処理など、特定のロボット制御則の計算を対象とした専用システムの開発と [7]–[9]、汎用の並列計算機を利用して、適当にスケジューリングされたロボット制御則の並列処理に大別される [10],[11]。ところが、前者は異なるロボット制御則へ応用する場合の汎用性や、ほかの処理系との融合性において問題がある。一方、後者はスケジューリング問題において、対象とするロボット制御則と並列計算機の特性を共に考慮した適切なタスクを決定することが難しい [55]。

ロボットの動的制御における非線形な下位制御則は、多変数の積和式として表される。多項式の計算の並列処理については、結合則と交換則を用いた計算木の高さの削減アルゴリズム（ツリーハイトリダクション）[56]–[58] がよく知られている。ところが、2分木状の演算を効率よく行うためには、プロセッサ間の通信が自由に遅延なく行える必要があるが、実際の並列計算機では、プロセッサ間の結合網はハードウェア量などの問題から制限されることが多い [59]。また、一般にロボット制御則の計算に必要な演算量は膨大であるため、従来の計算木に基づく並列計算手法が有効に適用できるほど、十分な数のプロセッサを準備することも現実的ではない。さらに、プロセッサの数が多くなりすぎると、プロセッサ間における頻繁な通信処理や同期処理が隘路となって、各プロセッサにおける計算効率を低下させる恐れがある。

このようにプロセッサの数やプロセッサ間の通信が制限される場合、計算木の高さの削減による演算の並列化のみならず、因数分解などを用いた数式の簡単化による冗長な演算の削除についても考慮すべきである。例えば、図 3.1 に示す変換は、プロセッサ数が2個に制限された場合において、従来の演算の並列化のみを考慮した計算木の構成に対して、ここで意図する数式の簡単化を施したものである。特に、積和式の形で与えら

れるロボット制御則は、各積項に共通な因子を数多く含んでいるために、前章で述べたような数式の簡単化による演算回数の削減は有効である。

この章では、1 つの積和式として与えられたロボット制御則の計算に対して、MIMD (Multiple Instruction stream Multiple Data stream) 分散メモリ型の並列計算機モデル [12],[13] を用いた並列処理手法を提案する。さらに、定められた個数のプロセッサのもとで、処理時間を最短とするためのスケジューリング問題について考える。

初めに、ロボット制御則の簡単化と並列化を共に考慮したスケジューリング問題を、組合せ最適化問題として定式化する。次に、この最適化問題が、*NP*困難なクラスに属する極めて難しい問題であることを証明する。最後に、対象とする最適化問題に対して、大規模な問題についても適用可能な近似アルゴリズムと、分枝限定法 [14] に基づき最適解を厳密に求めることができる最適化アルゴリズム、さらに、これら 2 つのアルゴリズムを組み合わせて、実用的な計算時間で精度の高い近似解が得られる準最適化アルゴリズムを、それぞれ提案する。

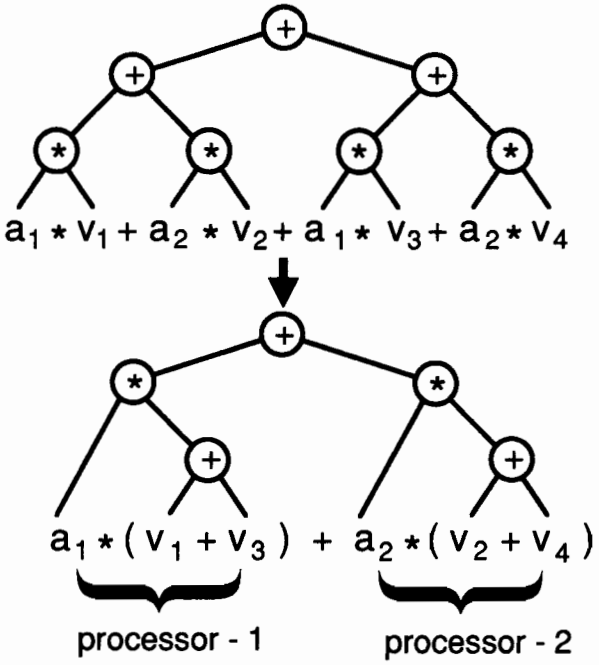


図 3. 1　計算木の簡単化と並列化

3.2 時間計算量とNP困難

この章で考えるロボット制御則のスケジューリング問題は、*NP*困難と呼ばれる極めて難しい問題である。ここでは、*NP*困難という概念について紹介する。

3.2.1 時間計算量

ある問題の難しさを評価するための尺度として、その問題を解くためのアルゴリズムが要する手間、すなわち時間計算量を定義する。ここで、問題とは与えられる入力例に対して、それに依存する解（答え）の出力を要求するものである。また、アルゴリズムとは、任意の入力例から解を得るための手順である。

初めに、問題の入力例の大きさ（サイズ）を、対象とする個々の問題ごと、あるいは、アルゴリズムごとに、ふさわしい形で定義する。例えば、実数データのソーティングであれば、与えられた実数データの総数が、その入力例のサイズに相当する。各問題に適した入力例のサイズが定義されると、個々の入力例のサイズによって、解こうとしている具体的な問題の規模を表すことができる。

次に、アルゴリズムにより問題の入力例に対して行われる特定の演算のみに着目して、それを単位演算として選び、この単位演算の総和によってアルゴリズムの時間計算量を定義する。単位演算としては、基本的な四則演算のほか、アルゴリズムを実行する各計算機モデルごとに定義される基本命令などが考えられる。例えば、実数データのソーティングであれば、実数の比較回数が単位演算として選ばれる。

さらに、アルゴリズムの時間計算量を、入力例のサイズ n、すなわち、問題の規模 n の関数 $T(n)$ として表現する。ここで重要なことは、データの転送や初期設定など、単位演算以外の処理も含めたアルゴリズム全体の時間計算量は、ある関数 $f(n)$ の定数倍以下になると言うことである。そこで、時間計算量 $T(n)$ は、次のような関数のオーダー（O–記法）を用いて表すものとする。

ある関数 $f(n)$ について、$T(n) = O(f(n))$ であるとは、任意の n $(n \geq m)$ について、次式が成り立つことである。ただし、m, c は適当に選んだ定数である。

$$T(n) \leq c * f(n) \tag{3.1}$$

上記の関数 $f(n)$ には、関数 $T(n)$ の特性をより良く表す、できるだけ単純な関数が用いられる。例えば、$T(n)$ が次数 r の多項式であれば、$f(n) = n^r$ とする。

関数のオーダーを用いると、アルゴリズムの時間計算量を、幾つかの部分に分けて考えることも容易となる。アルゴリズム中のある部分の時間計算量が $T_1(n) = O(f(n))$、別の部分の時間計算量が $T_2(n) = O(g(n))$ であるとする。この2つの処理を続けて実行する場合の全体の時間計算量 $T(n)$ は、次の「和の規則」から求められる。

$$T(n) = T_1(n) + T_2(n) = O(\max\{f(n),\ g(n)\}) \tag{3.2}$$

また、一方の処理（$T_1(n) = O(f(n))$）が、他方の処理（$T_2(n) = O(g(n))$）をサブルーチンとして繰り返し呼び出す場合などは、式(3.2)に示す「積の規則」から全体の時間計算量が求められる。これら時間計算量の評価に有用な「和の規則」と「積の規則」は、式(3.1)に示した関数のオーダーの定義から容易に導くことができる[54]。

$$T(n) = T_1(n) * T_2(n) = O(f(n) * g(n)) \tag{3.3}$$

3.2.2 NP完全問題

ある問題に対して、問題の規模nの多項式オーダー時間$O(n^r)$で解くことができるアルゴリズムが存在するならば、その問題は易しい問題であるとする。一方、考えられるすべてのアルゴリズムの時間計算量が、指数オーダー時間$O(c^n)$（cは適当な定数）等となり、多項式オーダー時間では解けないような問題は、難しい問題であるとする。ここでは、上記の意味において、明らかに難しい問題であると認知された「*NP*完全問題」について紹介する[60],[61]。

「*NP*完全」の厳密な定義をはじめ、計算量の理論における幾つかの重要な発見は、1930年代に、A. M. Turingによって提案されたTuring機械と呼ばれる単純な計算機モデルを用いて行われた[63]。この計算機モデルは、機能的には現在のコンピュータに劣らないほど強力なものであるが、構造的には実存するコンピュータと大きく異なる。また、計算機がパソコンとして家庭内にまで浸透してきた今日において、Turing機械によってアルゴリズムの議論を行う必要性は薄れている。さらに、計算量に関する理論的な研究は、歴史的に、Turing機械とは異なる計算機モデルや、数学的な概念に基づいても行われてきた。こうした事情を背景にして、A. Churchは「Churchの提唱」と呼ばれる以下のような仮説を立てている[60],[61]。

「アルゴリズムを持つ『問題』はすべて、『Turing機械』によって計算できる。」

上記の仮説において、『問題』と『Turing機械』の部分は、基礎とする数学的な定義や計算機モデルによって、適当な言葉で置き換えることができる。もちろん、「Churchの提唱」は証明されたものではないが、これを否定する人は殆んどいない。従って、以降では「Churchの提唱」に基づいて、正確にはTuring機械を構成して示すべきところを、単に文章によりアルゴリズムを記述するのみとする。

初めに、問題の難しさの議論によく用いられる判定問題について述べる。判定問題とは、与えられた入力例が、問題により指定された条件を満たすか否かを判定して、「イエス」か「ノー」の出力を求めるものである。このような判定問題の1つとして、「先行制

約条件のないスケジューリング問題」を以下に示す[62]。先行制約条件とは、与えられたタスク間に定められた、タスクの実行順序についての半順序関係である。すなわち、タスク間に先行制約条件が存在する場合には、順位の高いタスクの処理が終了するまで、順位の低いタスクの処理は開始することができない。

【先行制約条件のないスケジューリング問題（判定問題）】

入力： 処理時間 $Cost(b_j)$ を要する n 個のタスク b_j $(j = 1, 2, \ldots, n)$ と、プロセッサ数 P $(n \geq P \geq 2)$、最大処理時間 t_{max} $(t_{max} > 0)$。

目的： 式(3.4)を満たす添字集合 $S = \{1, 2, \ldots, n\}$ の P個の部分集合 S_pへの分割方法 $\{S_p \mid 1 \leq p \leq P\}$ において、式(3.5)に示す条件を満たすものが存在するとき「イエス」を、そうでないとき「ノー」を出力する。

$$\left[\begin{array}{l} S = S_1 \cup S_2 \cup \ldots \cup S_p \cup \ldots \cup S_P, \\ S_p \cap S_q = \emptyset \ \ (1 \leq p < q \leq P), \quad S_p \neq \emptyset \ \ (1 \leq p \leq P). \end{array}\right. \tag{3.4}$$

$$\max_{1 \leq p \leq P} \{ \sum_{j \in S_p} Cost(b_j) \} \leq t_{max} \tag{3.5}$$

■

さきに述べたように、易しい問題とは多項式オーダー時間のアルゴリズムが存在する問題であり、難しい問題とは多項式オーダー時間のアルゴリズムが存在しない問題である。アルゴリズムが存在することを示すことは容易であるが、存在しないことを証明するのはかなり厄介であり、かつて、このような証明に成功した人はいない。

そこで、ある種の判定問題に対しては、多項式オーダー時間のアルゴリズムが存在しないことを強く予想させるために、非決定性機械（NM: Nondeterministic Machine）と呼ばれる概念的な計算機モデルを導入する。非決定性機械は非常に強力な計算機のモデルであり、このような能力を持つ計算機は実在しない。また、将来、実現することも不可能であろう。これに対して、現在、実際に存在しているコンピューターに相当する計算機モデルは、決定性機械（DM: Deterministic Machine）と呼ばれる。

まず、決定性機械では、定められたアルゴリズムにしたがって、有限回数の基本命令を逐次的に実行することにより、判定問題の任意の入力例に対して「イエス」か「ノー」を出力して停止する。ここで、基本命令とは、通常のコンピューターにおける四則演算や比較，ジャンプ命令などである。また、決定性機械におけるアルゴリズムの時間計算量は、実行された基本命令の総数によって決められる。ただし、時間計算量のみを考えるので、決定性機械の記憶領域は無限にあるものとする。

一方、非決定性機械は、決定性機械の基本命令に加えて、非決定的な選択命令という基本命令を持っている。この非決定性機械は、K本の選択肢を持つ選択命令に達すると、記憶の内容も含めて自分自身のK個の複製を作り出して、これらK個の非決定性機械が、各々の選択肢における処理を独立に続行する。非決定性機械では、選択命令に出会うた

びに、このような自分自身の複製を際限なく作ることができる。ここで、処理の過程で生成された複製の1つでも、判定問題の入力例に対して「イエス」と答えるならば、非決定性機械による答えは「イエス」となる。ところが、非決定性機械による答えを「ノー」とするためには、すべての複製が「ノー」と答える必要がある。

このような非決定性機械におけるアルゴリズムの時間計算量は、「イエス」と答えるまでに至る基本命令の系列の中で、基本命令の数が最少のものによって定義される。また、非決定性機械が「ノー」と答える場合について、時間計算量の明確な定義はないが、「イエス」と答える場合よりも多くの時間計算量を要することは容易に推察できる。

そこで、判定問題に対するアルゴリズムを2種類に分類する。まず初めに、判定問題を計算するアルゴリズムとは、与えられた入力例が指定された条件を満たすときは「イエス」と答えて、満たさない場合には「ノー」と答えるものである。これに対して、判定問題を受理するアルゴリズムとは、与えられた入力例が指定された条件を満たすときのみ「イエス」と答える。従って、判定問題の入力例に対する答えが「ノー」となる場合には、判定問題を受理するアルゴリズムを実行した計算機は暴走してしまい停止しない恐れもある。このように、判定問題を「計算する」という概念と、「受理する」という概念は異なることに注意されたい。

以下では、易しい問題と難しい問題の分類を容易にするために、判定問題において「イエス」の答えを持つ入力例のみを対象として、これらを受理するときのアルゴリズムの時間計算量を評価する。まず、判定問題Aに対して無限個に存在する入力例のなかで、受理される入力例の集合を判定問題Aと考える。さらに、前述の決定性機械（DM）と非決定性機械（NM）を用いて、次のような判定問題Aの集合族、$D_{TIME}(f(n))$と$N_{TIME}(f(n))$を定義する。

$$\begin{cases} D_{TIME}(f(n)) & := \{\ \text{問題}\ A \mid \text{入力例がDMにより}\ O(f(n))\ \text{で受理できる}\ \} \\ N_{TIME}(f(n)) & := \{\ \text{問題}\ A \mid \text{入力例がNMにより}\ O(f(n))\ \text{で受理できる}\ \} \end{cases}$$

ここで、これら判定問題の集合族を用いて、クラスP（Polynomially solvable）とクラスNP（Nondeterministic Polynomial）を、次のように定義する。

$$\begin{cases} \text{クラス}\ P & := \displaystyle\bigcup_{r=1}^{\infty} D_{TIME}(n^r) \\ \text{クラス}\ NP & := \displaystyle\bigcup_{r=1}^{\infty} N_{TIME}(n^r) \end{cases}$$

クラスPとは、決定性機械（DM）において、「イエス」の答えを持つ入力例を多項式オーダー時間で受理できるアルゴリズムが存在する判定問題の全体であり、クラスNPとは、非決定性機械（NM）において、「イエス」の答えを持つ入力例を多項式オーダー時間で受理できるアルゴリズムが存在する判定問題の全体である。決定性機械は非決定性機械の特別な場合であるので、$P \subseteq NP$であることは明かである。しかし、この包含

関係が真の包含関係であるか否か、すなわち、$P \neq NP$であるかどうかは未解決の難問であり、「$P = NP$問題」と呼ばれている。現在のところ、多くの判定問題について、クラス Pに属するか否かは分からないが、クラス NPに属することが示されている。また、非決定性機械において多項式オーダー時間となるアルゴリズムの処理を、決定性機械によってエミュレーションすると、指数オーダー時間となることが知られている [63]。これらの事から、$P \neq NP$であるというのが、広く信じられている予測である。

例えば、さきに紹介した「先行制約条件のないスケジューリング問題」もクラス NPに属する。これを証明するためには、非決定性機械において実行可能な多項式オーダー時間のアルゴリズムを示せばよい。すなわち、与えられた n 個の各タスク b_1, b_2, ..., b_n を、プロセッサ p $(1 \leq p \leq P)$ に対して非決定的に割り当てる。このように作られた、あらゆる分割方法 $\{S_p \mid 1 \leq p \leq P\}$ に対して、同時に、最大処理時間 t_{max} に関する条件を満たすか否かを判定すればよい。このアルゴリズムの非決定性機械による時間計算量は、1 つの分割方法の受理に要する $O(n)$ である。

一方、非決定性機械における上記のアルゴリズムの計算を、決定性機械でエミュレーションしたとする。まず、n 個のタスクに関する P個のプロセッサへの割り当て方法は、全部で P^n通りである。さらに、それぞれの分割方法を評価するために $O(n)$ の時間計算量を必要とすることから、決定性機械における時間計算量は指数オーダー時間 $O(n * P^n)$ となる。また、現在のところ、「先行制約条件のないスケジューリング問題」に対して、決定性機械において時間計算量が多項式オーダー時間となるような効率の良いアルゴリズムは、発見されていない。

ところで、$P \neq NP$であると仮定して、ある判定問題 A が難しい問題であると主張するためには、$A \in NP$であるが $A \notin P$であることを示せばよい。この為には、判定問題 A がクラス NPの中で最も難しい問題であること、すなわち、その入力例の受理に要する時間計算量が、最も大きな問題であることが示せれば十分である。

【定義 3.1：多項式時間変換可能】

判定問題 A, Bにおいて、次の 2 つの条件が満たされるとき、Bは A に多項式時間変換可能であると言い、$B \prec A$ と表すものとする。

(1) Bの任意の入力例$\hat{B}$が、決定性機械 (DM) により、その問題の規模の多項式オーダー時間で、A のある入力例$\hat{A}$に変換できる。

(2) 上記の入力例$\hat{B}$が「イエス」の答を持つとき、かつ、そのときに限り、対応する A の入力例$\hat{A}$も「イエス」の答を持つ。 ■

直観的に言えば、$B \prec A$ であるとは、決定性機械において、判定問題 A を受理するアルゴリズムの時間計算量に、入力例の変換に要する多項式オーダー時間を加えた時間計算量によって、判定問題 Bが受理できることを意味する。

【定義 3.2：*NP*完全】
判定問題 A が *NP*完全であるとは、A が次の 2 つの条件を満たすことである。
(1)　A はクラス *NP*に属する。
(2)　クラス *NP*に属するすべての判定問題 Bに対して $B \prec A$ となる。 ■

ある判定問題 A が *NP*完全であることは、多項式オーダー時間の違いを無視して考えるとき、その問題 A はクラス *NP*の中で最も難しい問題であることを意味する。従って、$P \neq NP$を仮定すると $A \in NP$かつ $A \notin P$となり、*NP*完全な判定問題 A は、多項式オーダー時間のアルゴリズムが存在しない難しい問題であると言える。

*NP*完全であることが証明された最初の判定問題は、「充足可能性問題」である。「充足可能性問題」とは、積標準形によって与えられた論理式が充足可能であるか否か、すなわち、論理式を構成する Boole 変数に 1（真）か 0（偽）を割り当て、その論理式の値を 1 とすることが可能かどうかを判定する。S. A. Cook によって行われた「充足可能性問題」が *NP*完全であることの証明は、非決定性機械の具体的な表現の 1 つである非決定性 Turing 機械の計算プロセスを、「充足可能性問題」によりシュミレーションするという手法による [64]。その後、以下に示す補題に基づいて、多くの判定問題が *NP*完全となることが証明された [65]。さきに示した「先行制約条件のないスケジューリング問題」も、*NP*完全であることが知られている [62]。

【補題 3.1】
判定問題 A が *NP*完全であるための必要十分条件は、A が次の 2 つの条件を満たすことである。
(1)　A はクラス *NP*に属する。
(2)　ある *NP*完全問題 Bに対して $B \prec A$ となる。 ■

証明は自明であるので省略する。

3.2.3 最適化問題

*NP*完全の定義は、判定問題のみを対象としている。しかし、現実に直面する問題としては、判定問題のほかにも、探索問題や最適化問題などがある。

探索問題では、指定された条件を満たす解（許容解）が存在するとき、その解の 1 つを出力する。また、許容解が存在しない場合には「ノー」と答える。一方、最適化問題では、指定された条件を満たす許容解の中で、それに付随する値が最大または最小となる解（最適解）と、付随する値（最適値）を出力する。また、許容解が存在しない場合には「ノー」と答えることが求められる。

さきに示した「先行制約条件のないスケジューリング問題（判定問題）」に対応する探索問題と最適化問題を、それぞれ以下に示す。ただし、「先行制約条件のないスケジューリング問題」の最適化問題に対しては、少なくても 1 つの許容解が存在することが明らかであるので、「ノー」と答えることは求めていない。

【先行制約条件のないスケジューリング問題（探索問題）】

入力： 処理時間 $Cost(b_j)$ を要する n 個のタスク b_j $(j = 1, 2, \ldots, n)$ と、プロセッサ数 P $(n \geq P \geq 2)$、最大処理時間 t_{max} $(t_{max} > 0)$。

目的： 式(3.6)を満たす添字集合 $S = \{1, 2, \ldots, n\}$ の P個の部分集合 S_pへの分割方法 $\{S_p \mid 1 \leq p \leq P\}$ において、式(3.7)に示す条件を満たすものが存在するとき、その分割方法 $\{S_p^* \mid 1 \leq p \leq P\}$ を、そうでないとき「ノー」を出力する。

$$\left[\begin{array}{l} S = S_1 \cup S_2 \cup \ldots \cup S_p \cup \ldots \cup S_P, \\ S_p \cap S_q = \emptyset \ \ (1 \leq p < q \leq P), \quad S_p \neq \emptyset \ \ (1 \leq p \leq P). \end{array} \right. \tag{3.6}$$

$$\max_{1 \leq p \leq P} \{ \sum_{j \in S_p} Cost(b_j) \} \leq t_{max} \tag{3.7}$$

■

【先行制約条件のないスケジューリング問題（最適化問題）】

入力： 処理時間 $Cost(b_j)$ を要する n 個のタスク b_j $(j = 1, 2, \ldots, n)$ と、プロセッサ数 P $(n \geq P \geq 2)$。

目的： 式(3.8)を満たす添字集合 $S = \{1, 2, \ldots, n\}$ の P個の部分集合 S_pへの分割方法 $\{S_p \mid 1 \leq p \leq P\}$ において、式(3.9)に示す目的関数 $G(\cdot)$ の最小値と、それを与える分割方法 $\{S_p^* \mid 1 \leq p \leq P\}$ を求める。

$$\left[\begin{array}{l} S = S_1 \cup S_2 \cup \ldots \cup S_p \cup \ldots \cup S_P, \\ S_p \cap S_q = \emptyset \ \ (1 \leq p < q \leq P), \quad S_p \neq \emptyset \ \ (1 \leq p \leq P). \end{array} \right. \tag{3.8}$$

$$G(x) := \max_{1 \leq p \leq P} \{ \sum_{j \in S_p} Cost(b_j) \} \tag{3.9}$$

ただし、$x = \{S_p \mid 1 \leq p \leq P\}$ とする。 ■

さきに述べたクラス Pやクラス NP、NP完全などは、すべて判定問題について定義されたものである。そこで、判定問題に限らず、上記の探索問題や最適化問題も対象として、NP困難な問題を以下のように定義する。

【定義3.3：NP困難】

問題 Q が NP困難であるとは、その問題を解く多項式オーダー時間のアルゴリズムが存在すると仮定すると、「$P = NP$」が成立することである。 ■

ある問題 Q が NP困難である場合には、その問題が多項式オーダー時間で解けるためには「$P = NP$」であることが必要であるが十分ではない。従って、「$P = NP$」である

と仮定しても問題Qが多項式オーダー時間で解けるとは限らない。この意味において、*NP*困難な問題は、*NP*完全な問題も含めたクラス*NP*に属するどの判定問題よりも易しくはなく、同程度かそれ以上に難しい問題であると言える。

上記の*NP*困難の定義に基づき、「先行制約条件のないスケジューリング問題」の最適化問題が、*NP*困難であることを示す。

【補題3.2】

「先行制約条件のないスケジューリング問題」の最適化問題は*NP*困難である。

■

[証明]

「先行制約条件のないスケジューリング問題」の最適化問題を解くことができるアルゴリズムにより、既に*NP*完全であることが知られた「先行制約条件のないスケジューリング問題」の判定問題の答えが得られることを示す。

まず、判定問題の最大処理時間t_{max}以外の入力を、最適化問題の入力に対応させる。さらに、最適化問題を解いて最適値G^*を求めて、$G^* > t_{max}$ならば判定問題の答えとして「ノー」と出力し、$G^* \leq t_{max}$ならば「イエス」と出力すればよい。

ここで、与えられたタスクの数を問題の規模nとして、最適化問題に対するアルゴリズムの時間計算量が$O(f(n))$ならば、上記の方法により判定問題を計算するための時間計算量も$O(f(n))$である。*NP*完全な問題はクラス*NP*の中で最も難しい問題であるので、関数$f(n)$が多項式であれば、「$P = NP$」が成立する。　□

3.3 並列処理手法とスケジューリング問題

ここでは、提案するロボット制御則の並列処理手法について述べると共に、スケジューリング問題を最適化問題として定式化する。

前章でも述べたように、N関節（N自由度）を持つロボット・アームの動的制御において、各関節に加えるトルクまたは力 f_n $(n = 1, 2, \ldots, N)$ を決めるための非線形な下位制御則を、REDUCE や Mathematica などの数式処理システムを用いて導出すると、次の式 (3.10) に示すような積和式の形となる。

$$f_n := \sum_{j \in J_n} T_j, \qquad (n = 1, 2, \ldots, N) \tag{3.10}$$

$$T_j := \pm a_k * (\prod_{m \in M_j} u_m * * h_{m,j})$$

ただし、a_kは定数パラメータ、u_mは入力変数、$h_{m,j}$は u_m $(m \in M_j)$ の乗数である。ロボット制御則に含まれる三角関数の値はあらかじめ計算されているものとして、独立の入力変数 u_mと見なす。また、J_nは第 n 関節トルク f_n の計算に貢献する積項T_jの添字集合、M_jは全入力変数 u_mに対する添字集合の部分集合である。

提案する並列処理手法では、式 (3.10) に示したN自由度のロボット制御則に基づく N 個の関節トルク f_n $(n = 1, 2, \ldots, N)$ の計算において、各関節トルク f_nごと独立に並列処理を行うものとする。従って、以降においては議論を見やすくするために、各関節を区別する添字 n は、一般性を失うことなく省略する。すなわち、式 (3.10) に示したロボット制御則において、並列処理の対象とする任意の関節トルク f $(f = f_n)$ の計算式（ロボット制御則）を、次の式 (3.11) により表すものとする。

$$f := \sum_{j \in J} T_j \tag{3.11}$$

$$T_j := \pm a_k * (\prod_{m \in M_j} u_m * * h_{m,j})$$

ただし、Jは、関節トルク fの計算に貢献する積項T_jの添字集合を表す。

上記の式 (3.11) に示したロボット制御則の計算に対して、MIMD 分散メモリ型の並列計算機モデルによる並列処理を考える。この並列計算機モデルは、処理能力の等しい P 個のプロセッサから構成される。

まず、式 (3.11) のロボット制御則を、含まれる積項 T_j $(j \in J)$ を適当に P組に分けて、以下に示すように式 (3.12) と式 (3.13) に分割する。

$$f_p := \sum_{j \in J_p} T_j, \qquad (p = 1,\ 2,\ \ldots,\ P) \tag{3.12}$$

ただし、$\bigcup_{p=1}^{P} J_p = J$, $J_p \neq \emptyset\ (1 \leq p \leq P)$, $J_p \cap J_q = \emptyset\ (p \neq q)$ である。

$$f := \sum_{p=1}^{P} f_p \tag{3.13}$$

式 (3.12) に示した P個の積和式の計算処理は、並列計算機の P個のプロセッサにより同時に行われる。一方、式 (3.13) の加算については、式 (3.12) の積和式の計算がすべて終了した時点で、各プロセッサ間の結合網の状態などを考慮して、適当なプロセッサにより行うものとする [66]。ここで、プロセッサ数 Pに対して式 (3.11) のロボット制御則に含まれる積項 $T_j\ (j \in J)$ の数が十分に多いため、並列処理では式 (3.12) の計算時間が、式 (3.13) の計算時間と各プロセッサ間の通信時間に比べて支配的となる。

提案する並列処理手法においては、プロセッサ数 Pに対して、式 (3.11) に示した関節トルク fの計算式に含まれる演算量や積項の数が十分に多いことから、並列処理では各プロセッサごとに行われる式 (3.12) の計算時間が、式 (3.13) に示した各プロセッサで得られた計算結果 f_pの加算に要する時間と、プロセッサ間の通信時間に比べて支配的となる。従って、ロボット制御則のスケジューリング問題では、式 (3.13) の計算時間と通信時間は、式 (3.12) の計算時間に対して、相対的に無視できるものとする。

また、DSP[15]–[17] やトランスピュータ [67] など、ハードウェア乗算器を内蔵した最近のプロセッサでは、乗算と加減算の処理時間が、ほぼ等しい。従って、式 (3.12) に示した積和式の計算時間は、積和式に含まれる乗算と加減算の演算回数から評価できるものとする。ただし、べき乗は乗算に展開して評価する。

ところで、前章でも述べたように、式 (3.11) に示した積和式の形のロボット制御則は、各積項に多くの共通な因子を含んでいる。このことを踏まえて、式 (3.11) のロボット制御則の計算を、式 (3.12) に示したように積項単位で P個のプロセッサに割り当てた後に、各プロセッサの計算効率を高めるために、式 (3.12) の各積和式を、それぞれ簡単化して冗長な演算を削除する。また、この数式の簡単化には、前章で提案した「大域的な簡単化手法」を用いるものとする。

例えば、図 3.2 に示すような 2 自由度のロボット・アームの第 2 関節に対して、2．2 節の式 (2.5) で示した線形化補償器の制御則を数式処理システムにより導出すると、次の式 (3.14) に示すような積和式の形のロボット制御則が得られる。

$$f := \sum_{j \in J} T_j, \qquad (J = \{1,\ 2,\ \ldots,\ 7\}) \tag{3.14}$$

$$\begin{cases} T_1 := b_1 * ddp_1, & T_2 := L_1 * b_2 * c_2 * ddp_1, & T_3 := b_1 * ddp_2, \\ T_5 := D_2 * dq_2, & T_6 := -g * b_2 * s_2 * s_1, & T_7 := g * b_2 * c_2 * c_1, \\ T_4 := L_1 * b_2 * s_2 * dq_1 * dp_1. \end{cases}$$

ただし、q_n $(n = 1, 2)$ は各関節の変位、dq_nはその速度、ddp_n，dp_nは目標軌道 p_nに基づく適当な制御量、L_nは各リンクの長さ、D_nは摩擦係数、b_m $(m = 1, 2)$ は基底パラメータ [49]、gは重力加速度である。また、c_n，s_nは三角関数 $\cos(q_n)$，$\sin(q_n)$ を表す。

ここで、プロセッサの数を $P = 2$ とすると、式 (3.14) のロボット制御則については、次のような計算式の並列化と簡単化が可能である。

$$\begin{cases} f_1 := T_4 + T_3 + T_1 + T_5 \\ \quad := L_1 * b_2 * s_2 * dq_1 * dp_1 + b_1 * ddp_2 + b_1 * ddp_1 + D_2 * dq_2 \\ \quad := L_1 * b_2 * s_2 * dq_1 * dp_1 + b_1 * (ddp_2 + ddp_1) + D_2 * dq_2 \\ \\ f_2 := T_6 + T_7 + T_2 \\ \quad := -g * b_2 * s_2 * s_1 + g * b_2 * c_2 * c_1 + L_1 * b_2 * c_2 * ddp_1 \\ \quad := b_2 * (g * (c_2 * c_1 - s_2 * s_1) + L_1 * c_2 * ddp_1) \\ \\ f := f_1 + f_2 \end{cases} \tag{3.15}$$

ただし、$J_1 = \{1, 3, 4, 5\}$，$J_2 = \{2, 6, 7\}$，$J = J_1 \cup J_2$である。また、数式の簡単化に伴い自動的に導入される中間変数 v_mは省略してある。

前章で提案した「大域的な簡単化手法」により簡単化された積和式を $Fac(\cdot)$、数式に含まれる演算回数（加減算と乗算の総数）を $Op(\cdot)$ により表すと、ロボット制御則のスケジューリング問題は、以下のように定式化できる。

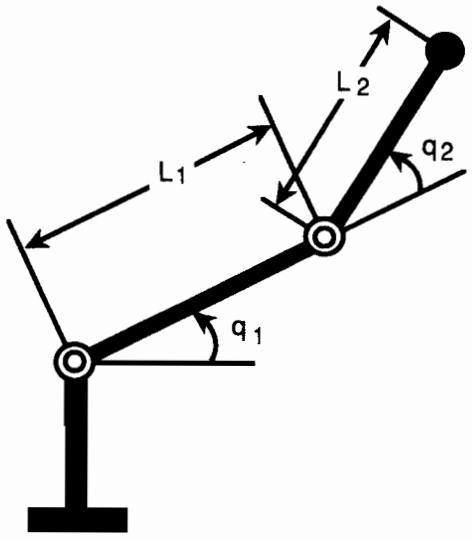


図 3. 2　2 自由度のロボット・アーム

【最適化問題 3.1】
入力： 式(3.11)に含まれる積項T_j $(j \in J)$と、プロセッサ数P $(|J| \geq P \geq 2)$。
目的： 式(3.16)を満たす添字集合$J = \{1, 2, \ldots, n\}$のP個の部分集合J_pへの分割方法$\{J_p \mid 1 \leq p \leq P\}$において、式(3.17)に示す目的関数$E(\cdot)$の最小値と、それを与える分割方法$\{J_p^* \mid 1 \leq p \leq P\}$を求める。

$$\begin{cases} J = J_1 \cup J_2 \cup \ldots \cup J_p \cup \ldots \cup J_P, \\ J_p \cap J_q = \emptyset \ \ (1 \leq p < q \leq P), \quad J_p \neq \emptyset \ \ (1 \leq p \leq P). \end{cases} \tag{3.16}$$

$$E(x) := \max_{1 \leq p \leq P} \{Op(Fac(\sum_{j \in J_p} T_j))\} \tag{3.17}$$

ただし、$x = \{J_p \mid 1 \leq p \leq P\}$とする。 ■

ここで、上記の最適化問題3.1が*NP*困難であることを示す。まず、最適化問題3.1の入力例の部分集合として、与えられた積項T_j $(j \in J)$の間に、共通する因子が存在しない場合を考える。このような入力例については、目的関数の評価において数式の簡単化が行われない。従って、このような入力例の部分集合は、最適化問題3.1の目的関数$E(\cdot)$を$\hat{E}(\cdot)$に緩和した新たな最適化問題3.2と考えることができる。

【最適化問題 3.2】
入力： 式(3.11)に含まれる積項T_j $(j \in J)$と、プロセッサ数P $(|J| \geq P \geq 2)$。
目的： 式(3.18)を満たす添字集合$J = \{1, 2, \ldots, n\}$のP個の部分集合J_pへの分割方法$\{J_p \mid 1 \leq p \leq P\}$において、式(3.19)に示す目的関数$\hat{E}(\cdot)$の最小値と、それを与える分割方法$\{J_p^* \mid 1 \leq p \leq P\}$を求める。

$$\begin{cases} J = J_1 \cup J_2 \cup \ldots \cup J_p \cup \ldots \cup J_P, \\ J_p \cap J_q = \emptyset \ \ (1 \leq p < q \leq P), \quad J_p \neq \emptyset \ \ (1 \leq p \leq P). \end{cases} \tag{3.18}$$

$$\hat{E}(x) := \max_{1 \leq p \leq P} \{Op(\sum_{j \in J_p} T_j)\} \tag{3.19}$$

ただし、$x = \{J_p \mid 1 \leq p \leq P\}$とする。 ■

【補題 3.3】
最適化問題3.2は*NP*困難である。 ■
[証明]
最適化問題3.2が、補題3.2によって*NP*困難であることが証明された「先行制約条件のないスケジューリング問題（最適化問題）」と等価な問題であることを示す。
最適化問題3.2のように数式の簡単化を行わない場合、積和式に含まれる演算回数は、

各積項T_jに含まれる乗算回数$Op(T_j)$と、積項間の加減算回数の総和となる。

$$Op(\sum_{j\in J_p} T_j) = \sum_{j\in J_p}(Op(T_j)+1) - 1, \quad (p = 1,\ 2,\ \ldots,\ P). \tag{3.20}$$

従って、最適化問題3.2の目的関数$\hat{E}$は、次のように変形できる。

$$\hat{E}(x) := \max_{1\le p\le P}\{\sum_{j\in J_p}(Op(T_j)+1)\} - 1 \tag{3.21}$$

十分性：最適化問題3.2で与えられたすべての積項T_j $(1 \le j \le |J|)$を、「先行制約条件のないスケジューリング問題」の各タスクb_j $(1 \le j \le n,\ n = |J|)$に対応させる。ただし、各タスク$b_j$の処理時間$Cost(b_j)$は、式(3.21)の目的関数を考慮して、対応する積項T_jに含まれる乗算回数$Op(T_j)$に1を加えた値に等しいものとする。

$$Cost(b_j) := Op(T_j) + 1, \quad (j = 1,\ 2,\ \ldots,\ n). \tag{3.22}$$

ここで、「先行制約条件のないスケジューリング問題」を解いて得られた最適解と最適値を、それぞれ$\{S_p^* \mid 1 \le p \le P\}$と$G^*$とすれば、これらが最適化問題3.2の最適解$J_p^* = S_p^*$ $(1 \le p \le P)$と最適値$\hat{E}^* = G^* - 1$を与える。

必要性：「先行制約条件のないスケジューリング問題」で与えられたタスクb_j $(1 \le j \le n)$を、最適化問題3.2の各積項T_j $(1 \le j \le |J|,\ |J| = n)$に対応させる。このとき、十分性の証明と同様にして、最適化問題3.2を解いて得られた最適解$\{J_p^* \mid 1 \le p \le P\}$と最適値$\hat{E}^*$が、「先行制約条件のないスケジューリング問題」の最適解$S_p^* = J_p^*$ $(1 \le p \le P)$と最適値$G^* = \hat{E}^* + 1$を与える。 □

【系3.3】

最適化問題3.1は*NP*困難である。 ■

[証明]

最適化問題3.2は、最適化問題3.1の入力例の部分集合である。従って、補題3.3より、最適化問題3.1が*NP*困難であることは明かである。 □

3.4 近似アルゴリズム

最適化問題 3.1 は NP困難であるため、その最適解を求めることは容易ではない。そこで、最適化問題 3.1 の最適とは言えないが、優れた許容解の 1 つである近似解を求めるための近似アルゴリズムを提案する [68],[69]。提案する近似アルゴリズム GCF/LPT 法（Greatest Common Factor/Longest Processing Time）は、さきに紹介した「先行制約条件のないスケジューリング問題（最適化問題）」に対する近似アルゴリズムとして知られる LPT 法（Longest Processing Time）[70] を改良したものである。

3.4.1 近似アルゴリズム LPT 法

初めに、「先行制約条件のないスケジューリング問題」のタスク b_j $(1 \leq j \leq n)$ を、最適化問題 3.1 の積項 T_j $(1 \leq j \leq |J|,\ n = |J|)$ に対応させることにより、上記の LPT 法を最適化問題 3.1 に対して、そのまま流用した近似アルゴリズムを以下に示す。

【近似アルゴリズム 3.1（LPT 法）】

(1) プロセッサ数 Pに等しい空の添字集合 J_p $(p = 1,\ 2,\ \ldots,\ P)$ を用意する。

(2) 各積項 T_j $(j \in J)$ に含まれる乗算回数を調べる。

(3) すべての積項 T_j $(j \in J)$ を含まれる乗算回数が多い順に並べる。

(4) 乗算回数の多い積項 T_iから順番に、次式の評価関数 $\hat{F}_p(J_p)$ の値が最小である添字集合 J_pに対して、積項 T_iの添字 i を $J_p = J_p \cup \{\ i\ \}$ のように配分する。

$$\hat{F}_p(J_p) := Op(\sum_{j \in J_p} T_j), \qquad (1 \leq p \leq P) \tag{3.23}$$

(5) 添字集合の集合 $x = \{J_p \mid 1 \leq p \leq P\}$ と、これに対して得られる最適化問題 3.1 の目的関数 $E(x)$ の値を、最適化問題 3.1 の解とする。 ■

上記の LPT 法の時間計算量を評価する。与えられる積項 T_j $(j \in J)$ が、各積項に含まれる乗算回数やプロセッサ数 Pに比べて十分に多いものとして、積項の総数 $|J|$ を問題の規模 $n = |J|$ とする。LPT 法において、手順 (2) と手順 (4) の時間計算量は $O(n)$ である。また、手順 (3) における積項の乗算回数（整数値）によるソーティングについては、整数値の比較回数が最大で $(n * \log_2 n)$ 回となるアルゴリズムが知られている [63]。そこで、全体の時間計算量は、手順 (5) の目的関数 $E(\cdot)$ の計算を除いて、「和の規則」より $O(n * \log_2 n)$ となる。

3.4.2 近似アルゴリズムGCF/LPT法

従来のLPT法では、積項T_j $(j \in J)$のプロセッサへの配分に際して、数式の簡単化の効果を考慮していないために、最適化問題3.1に適用しても優れた近似解が得られない。そこで、上記のLPT法を改良して、最適化問題3.1に対する新たな近似アルゴリズム（GCF/LPT法）を提案する。

提案するGCF/LPT法では、お互いに共通因子の少ない積項が異なるプロセッサへ割り当てられるように、与えられた積項の共通因子を前処理によって調べる。この処理の基本的なアイデアを、前節で式(3.14)に示したロボット制御則を例に挙げて説明する。式(3.14)に含まれる各積項を節点とし、積項間の共通な定数および変数を辺とするグラフを作ると図3.3のようになる。このようなグラフの各節点、すなわち、積項T_j $(j \in J)$に対して、以下の手順によりラベル付けを行う。ただし、すべての積項T_j $(j \in J)$は、あらかじめ含まれる乗算回数が少ない順に並べられているものとする。

【ラベル付けアルゴリズム】

(1) ラベルが付いていない積項の中から、含まれる乗算回数が最も少ない積項を1つ選び、その積項のラベルを$k=1$とする。
(2) ラベル1の積項に含まれる、異なる要素（変数と定数）の組みを共通因子とする。
(3) ラベルのない各積項について、共通因子に含まれる要素の数を調べる。
(4) 共通因子に含まれる要素を持つ積項が存在しなければ、手順(8)へ進む。
(5) 共通因子に含まれる要素の数が最も多い積項の中で、含まれる乗算回数が最少である積項を1つ選び、その積項のラベルを$(k+1)$とする。
(6) 共通因子とラベル$(k+1)$が付けられた積項の双方に共通して存在する要素を求め、これらを新たな共通因子とする。
(7) $k := k+1$として手順(3)に戻る。
(8) ラベルの付いた積項を思考の対象から除いて、手順(1)に戻る。 ■

上記の「ラベル付けアルゴリズム」の時間計算量を評価する。与えられる積項の総数$|J|$を、問題の規模$n=|J|$とする。このとき、手順(1)~(8)のループの実行回数は$O(n)$により抑えられる。さらに、サブルーチンとして繰り返される手順(3)~(7)のループの実行回数も$O(n)$によって抑えられる。従って、「ラベル付けアルゴリズム」全体の時間計算量は、「積の規則」より$O(n^2)$となる。

図3.3に示したグラフにおいて、同じラベルを持つ積項の間には共通因子が少ないことが分かる。すなわち、上記の「ラベル付けアルゴリズム」を用いて、与えられた積項T_j $(j \in J)$に対してラベル付けを行った後に、同じラベルを持った積項について集合を作ると、1つの集合には比較的に共通因子の少ない積項が集められる。従って、このような積項の集合ごとに最適化問題3.1を考えると、それらの問題は数式の簡単化を考慮しない最適化問題3.2に近い性質を持っていると予想される。そこで、同じラベルを持つ積項の集合ごとにLPT法を適用すれば、同じ集合に含まれる共通因子の少ない積項が、

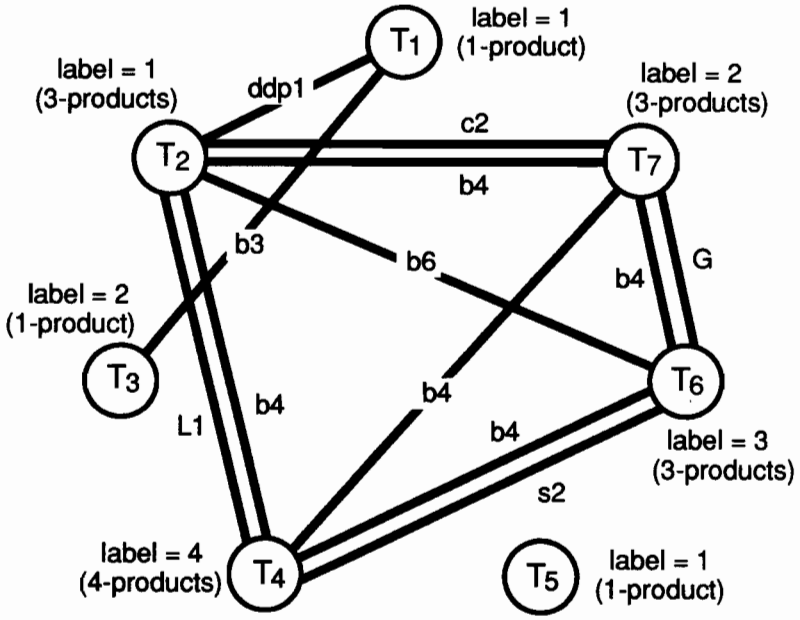


図 3. 3　積項に対するラベル付け

異なるプロセッサに分散されることが期待できる。

また、「ラベル付けアルゴリズム」では、含まれる乗算回数が少ない積項を優先してラベル付けを行っている。このため、大きなラベルを持つ積項の集合には、比較的に含まれる乗算回数の多い積項が集まる。従って、LPT 法の効果を考慮すると、大きなラベルを持つ積項の集合から順番に、各プロセッサへ割り当てればよい。

さらに、提案する GCF/LPT 法では、多くの共通因子を含む積項が、できるだけ同じプロセッサに配分されるように、数式の簡単化を行った後の演算回数を予測して、積項の割り当てを行う。すなわち、簡単化された積和式に含まれる演算回数は、明らかに簡単化を行う前の積和式に含まれる演算回数 $Op(\cdot)$ よりも少ない。また、数式の簡単化手法における「定数の畳み込み」の効果を考慮しても、簡単化された積和式に含まれる演算回数は、その積和式に含まれる異なる変数の総数 $Atom(\cdot)$ よりも多くなる。そこで、これらの平均値を取ることによって、各積項を割り当てるプロセッサを考える際に、簡単化された積和式に含まれる演算回数 $Op(Fac(\cdot))$ を推定するものとする。

【近似アルゴリズム 3.2（GCF/LPT 法）】

(1)　プロセッサ数 P に等しい空の添字集合 J_p $(p = 1, 2, \ldots, P)$ を用意する。
(2)　各積項 T_j $(j \in J)$ に含まれる乗算回数を調べる。
(3)　すべての積項 T_j $(j \in J)$ を含まれる乗算回数が少ない順に並べる。
(4)　ラベル付けアルゴリズムを用いて、すべての積項 T_j $(j \in J)$ にラベルを付ける。
(5)　同じラベルの付いた積項を 1 つの集合にまとめる。
(6)　大きなラベルを持つ積項の集合から、次の手順 (7)~(8) を繰り返す。
(7)　集合内の積項を含まれる乗算回数が多い順に並べる。
(8)　乗算回数の最も多い積項 T_i から順番に、次式の評価関数 $F_p(J_p)$ の値が最小である添

字集合 J_pに対して、積項 T_iの添字 i を $J_p = J_p \cup \{ i \}$ のように配分する。

$$F_p(J_p) := Op(\sum_{j \in J_p} T_j + T_i) + Atom(\sum_{j \in J_p} T_j + T_i), \qquad (1 \leq p \leq P) \qquad (3.24)$$

(9) 添字集合の集合 $x = \{J_p \mid 1 \leq p \leq P\}$ と、これに対して得られる最適化問題 3.1 の目的関数 $E(x)$ の値を、最適化問題 3.1 の解とする。 ■

上記の GCF/LPT 法の時間計算量を評価する。LPT 法と同様に、与えられる積項の総数 $|J|$ を問題の規模 $n = |J|$ とする。GCF/LPT 法において、手順 (2) の時間計算量は $O(n)$ である。また、手順 (3) におけるソーティングの時間計算量は $O(n * \log_2 n)$ である。さらに、手順 (4) のラベル付けの時間計算量は、前述のように $O(n^2)$ である。ここで、手順 (7) のソーティングの時間計算量は、すでに手順 (3) において積項がソートされているために、各集合に含まれる積項の数によって抑えられる。従って、手順 (6)~(8) のループ全体に要する時間計算量は $O(n)$ である。以上の考察から、全体の時間計算量は、手順 (9) の目的関数 $E(\cdot)$ の計算を除いて、「和の規則」より $O(n^2)$ となる。

3.5 最適化アルゴリズム

ここでは、*NP*困難な組合せ最適化問題の最も有力な解法の1つである分枝限定法[14]に基づき、最適化問題3.1の最適解を厳密に求めることができる最適化アルゴリズムを提案する[71],[72]。分枝限定法とは、いわば要領の良い列挙法[73]であって、直接解くのが困難であるような複雑な組合せ最適化問題を、より易しい小規模な部分問題に分割して、それらをすべて解くことにより等価的に原問題を解くものである。従って、分枝限定法は、与えられた原問題の部分問題を次々に生成していく分枝操作と、できるだけ多くの部分問題を終端させるための限定操作から構成される。

分枝限定法では、最適解の探索に伴って生成される部分問題の総数が、アルゴリズムの計算時間を左右する。そこで、提案する最適化アルゴリズムでは、等価な部分問題を排除して必要最少の部分問題を生成するための分枝操作を考案する。

また、限定操作としては、前節で提案した近似アルゴリズム（GCF/LPT法）により得られた近似解を利用して、多くの部分問題を探索の初期の段階で終端させる工夫を凝らしている。さらに、目的関数値の下界値と上界値を用いることにより、各部分問題を終端するか否かの判定の高速化も図っている。

▼ 部分問題

最適化問題3.1の部分問題では、与えられた積項$T_j\,(j \in J)$の一部の積項$T_j\,(j \in I \subseteq J)$について、$Q$個$(Q \leq P)$のプロセッサへの配分方法を考える。すなわち、添字集合Jの部分集合$I\,(I \subseteq J)$を対象として、添字集合IのQ個$(Q \leq P)$の部分集合$I_p\,(1 \leq p \leq Q)$への分割方法eを、最適化問題3.1の部分問題eとする。

$$e = \{\ I_p \mid 1 \leq p \leq Q\ \} \tag{3.25}$$

ただし、$\bigcup_{p=1}^{Q} I_p = I,\ I_p \neq \emptyset\,(1 \leq p \leq Q),\ I_p \cap I_q = \emptyset\,(p \neq q)$である。

ここで、式(3.25)に示したような部分問題（分割方法）$e = \{I_p \mid 1 \leq p \leq Q\}$に対して、次の式(3.26)に示す目的関数$E(e)$の値を求めるものとする。

$$E(e) := \max_{1 \leq p \leq Q} \{ Op(Fac(\sum_{j \in I_p} T_j)) \} \tag{3.26}$$

さらに、ある分割方法$e = \{I_p \mid 1 \leq p \leq Q\}$において、$I = J\ (I = I_1 \cup \cdots \cup I_Q)$、かつ、$Q = P$が成り立つとき、分割方法$e$は最適化問題3.1の許容解である。

▼ 分枝操作

集合の分割アルゴリズム[74]を応用した、部分問題eの生成方法（分枝操作）を提案する。この分枝操作は、与えられた積項の添字集合Jのすべての部分集合$I\ (I \subseteq J)$に

対して、それらの幾つかの部分集合 I_p ($I_p \subseteq I$) への分割方法を重複なく、見落としがないように順次発生させるものである。

次の式(3.27)に示す添字集合 I ($I \subseteq J$) に対する Q 個の部分集合への1つの分割方法（部分問題）e から、新たな部分問題を派生させる場合について説明する。

$$e = \{I_1,\ I_2,\ \ldots,\ I_p,\ \ldots,\ I_Q\} \tag{3.27}$$

ただし、$I = I_1 \cup\ I_2 \cup \cdots \cup I_Q$である。

初めに、式(3.27)の分割方法 e から、新たな添字集合$\hat{I}$ ($\hat{I} \subseteq J$) に対する $(Q+1)$ 通りの分割方法 e^r ($r = 1 \sim Q+1$) を、以下のように派生させる。

$$\left[\begin{array}{rcl} e^1 & = & \{\{I_1 \cup \{\ j\ \}\},\ I_2,\ \ldots,\ I_p,\ \ldots,\ I_Q\}, \\ e^2 & = & \{I_1,\ \{I_2 \cup \{\ j\ \}\},\ \ldots,\ I_p,\ \ldots,\ I_Q\}, \\ & \vdots & \\ e^p & = & \{I_1,\ I_2,\ \ldots,\ \{I_p \cup \{\ j\ \}\},\ \ldots,\ I_Q\}, \\ & \vdots & \\ e^Q & = & \{I_1,\ I_2,\ \ldots,\ I_p,\ \ldots,\ \{I_Q \cup \{\ j\ \}\}\}, \\ e^{Q+1} & = & \{I_1,\ I_2,\ \ldots,\ I_p,\ \ldots,\ I_Q,\ \{\ j\ \}\}. \end{array}\right. \tag{3.28}$$

ただし、$\hat{I} = I \cup \{\ j\ \}$, $j \notin I$、かつ、$j \in J$である。

ここで、式(3.28)に示した添字集合$\hat{I}$ ($\hat{I} \subseteq J$) に対する任意の分割方法 e^rを、改めて $\hat{e} = \{\hat{I}_p \mid 1 \leq p \leq \hat{Q}\}$ と表す。このとき、次の式(3.29)と式(3.30)に示す2つの条件を、共に満足する分割方法$\hat{e}$のみが、新たな部分問題として有効である。

$$\hat{Q} \leq P \tag{3.29}$$

$$P - \hat{Q} \leq |J| - |\hat{I}| \tag{3.30}$$

ただし、$\hat{I} = \hat{I}_1 \cup \cdots \cup \hat{I}_{\hat{Q}}$。また、$|\hat{I}|$, $|J|$ は、添字集合$\hat{I}$, Jの要素数を表す。

式(3.29)の条件が満たされないと、並列処理においてP個以上のプロセッサが必要となる。また、式(3.30)の条件が満たされないと、最終的に積項が全く配分されないプロセッサが生じる。従って、上記の条件を満たさない分割方法 $\{\hat{I}_p \mid 1 \leq p \leq \hat{Q}\}$ は、最適化問題3.1の許容解とは成りえない。そこで、提案する分枝操作としては、このような分割方法が生成されると、直ちに破棄するものとする。

例えば、与えられた積項の添字集合が $J = \{1,\ 2,\ 3,\ 4\}$, プロセッサ数が $P = 3$ である場合、提案した分枝操作によって生成される部分問題をすべて列挙すると、図3.4に示すような分枝図が得られる。図3.4の分枝図において、破線の枝は式(3.29), 式(3.30)の条件を満たさないために除かれる分割方法を示している。図3.4に示した分枝図には、明らかに冗長な分割方法（部分問題）は存在せず、最適化問題3.1に対して必要最少の

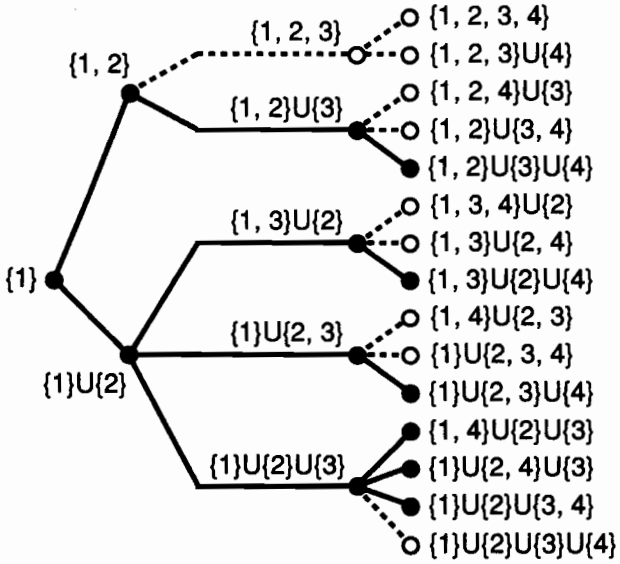


図3．4　部分問題の分枝図

部分問題が生成されることが確認できる。

▼ 限定操作

分枝操作による部分問題の生成のみを続けると、図3.4に示したような分枝図に現れるすべての部分問題を評価することになる。これでは、実質的に単なる列挙法[73]と変わらず、大規模な問題を扱うことができない。そこで、ある部分問題からは、既に得られている暫定的な解（暫定解）よりも、優れた許容解が得られないことが明らかであるならば、その部分問題を直ちに終端して、以後の考察から除くものとする。

現時点において生成されている部分問題のうち、まだ終端も目的関数$E(\cdot)$の評価もされていない部分問題を、活性な部分問題と呼ぶ。また、それまでの探索で得られた許容解のうち、最良のものを暫定解xとして保持するものとする。さらに、暫定解xに対する目的関数$E(\cdot)$の値を暫定値と呼び$Z = E(x)$と表す。

ある部分問題$e = \{I_p \mid 1 \leq p \leq Q\}$に対して、式(3.26)の目的関数$E(e)$の値を求めて、次の式(3.31)に示す関係が成り立てば、部分問題eからは、現在の暫定解xよりも優れた許容解は得られない。従って、この部分問題eは終端する。

$$Z \leq E(e) \tag{3.31}$$

一方、ある部分問題eに対する目的関数$E(e)$の値が、式(3.31)の関係を満たさなければ、部分問題eから暫定解xよりも優れた許容解が作られる可能性がある。従って、この部分問題eからは新たな部分問題e^r $(r = 1 \sim Q + 1)$を派生させる。

ところで、1つの部分問題eに対して目的関数$E(e)$の値を求めるためには、数式の簡単化を行う必要があり、生成されたすべての部分問題eに対して$E(e)$を評価することは、計算時間において不利である。従って、提案する最適化アルゴリズムでは、目的関

数値 $E(e)$ の上界値と下界値を利用して、部分問題 e の評価の高速化を図っている。

式 (3.26) に示した部分問題 $e = \{I_p \mid 1 \leq p \leq Q\}$ の目的関数値 $E(e)$ に対する上界値 $U(e)$ と下界値 $L(e)$ を、それぞれ以下のように定義する。

$$U(e) := \max_{1 \leq p \leq Q} \{Op(\sum_{j \in I_p} T_j)\} \tag{3.32}$$

$$L(e) := \max_{1 \leq p \leq Q} \{Atom(\sum_{j \in I_p} T_j)\} \tag{3.33}$$

ただし、関数 $Atom(\cdot)$ は、数式に含まれる異なる変数の総数を表す。

このとき、簡単な数式の簡単化について考えれば分かるように、次の式 (3.34) に示す関係が、任意の部分問題 $e = \{I_p \mid 1 \leq p \leq Q\}$ に対して常に成り立つ。

$$L(e) \leq E(e) \leq U(e) \tag{3.34}$$

従って、部分問題 e に対して、次の式 (3.35) に示す関係が成り立てば e を終端し、式 (3.36) に示す関係が成り立てば、e から新たな部分問題を派生させればよく、これらの関係が成り立つ場合には、実際に目的関数値 $E(e)$ を求める必要はない。

$$Z \leq L(e) \tag{3.35}$$

$$U(e) < Z \tag{3.36}$$

▼ 最適化アルゴリズムの構成

一般に分枝限定法においては、優れた暫定値 Zが早期に求まるほど、終端される部分問題の割合が増えて探索すべき領域が縮小する。従って、前節で提案した最適化問題 3.1 に対する近似アルゴリズム（GCF/LPT 法）を用いて得られた近似解 x'を、最適化アルゴリズムにおける暫定解 x の初期値とする。

また、探索方法としては、図 3.4 に示した分枝図の各節点（部分問題）を、左回りに辿る深さ優先探索法を採用する。そこで、このような深さ優先探索を実現するために、活性な部分問題 e を保存するための抽象的なデータ構造としてスタック Sを用いる。スタック Sとは、データの挿入や削除を常に一方の端から行うリストであり、以下のような付随する操作がある [54]。

- push(e,S) : 要素 e をスタック Sの先頭に入れる。
- top(S) : スタック Sの先頭の要素を返す。（スタック Sからは削除しない）
- pop(S) : スタック Sの先頭の要素を返して、スタック Sから削除する。
- empty(S) : スタック Sが空のとき真、そうでなければ偽を返す。

提案する最適化アルゴリズムを以下に示す。スタック Sが空になると、アルゴリズムの計算は終了して、その時点における暫定解 $x = \{J_p \mid 1 \leq p \leq P\}$ と暫定値 $Z = E(x)$ が、最適化問題 3.1 の最適解 x^*と最適値 E^*を与える。

【最適化アルゴリズム 3.1】

begin

1 : $Z := E(x')$;　　// GCF/LPT 法による近似解 $x' = \{J_p \mid 1 \leq p \leq P\}$
2 : $I_1 := \{1\}$;　$e := \{\ I_1\ \}$;　$push(e, S)$;
3 : $while\ (not\ empty(S))\ \{$
4 :　　$e := top(S)$;　　// 限定操作の開始
5 :　　$if\ (U(e) \geq Z)\ \{$
6 :　　$if\ (L(e) < Z)\ \{$
7 :　　$if\ (E(e) < Z)\ \{$
8 :　　$if\ (I = J)\ \{$
9 :　　$x := pop(S)$;　$Z := E(x)$;
10 :　　$\}$　　// 暫定解と暫定値の更新
11 :　　$\}\ else\ pop(S)$;　　// 目的関数による終端
12 :　　$\}\ else\ pop(S)$;　　// 下界値による終端
13 :　　$\}$　　// 部分問題：$e = \{I_p \mid 1 \leq p \leq Q\}$
14 :　　$e := pop(S)$;　　// 分枝操作の開始
15 :　　$make\ new\ e^r\ (r = 1 \sim Q + 1)\ from\ e$;
16 :　　$for\ each\ e^r\ (r = 1 \sim Q + 1)\ do\ \{$
17 :　　$if\ ((\hat{Q} \leq P)\ and\ (P - \hat{Q} \leq |J| - |\hat{I}|))\ \ push(e^r, S)$;
18 :　　$\}$　　// 部分問題：$e^r = \{\hat{I}_p \mid 1 \leq p \leq \hat{Q}\}$
19 : $\}$
20 : $output\ x = \{J_p \mid 1 \leq p \leq P\}\ and\ Z$;　　// 最適解と最適値の出力

end. ■

3.6 準最適化アルゴリズム

最適化問題3.1は*NP*困難であるため、問題の規模が大きくなると最適解の探索に要する計算時間が指数関数的に増加する。このため、いわゆる組合せ論的爆発によって、前節で提案した最適化アルゴリズム3.1のように分枝限定法を用いて計算の効率化を図った場合でも、厳密に最適解を求めることが困難となる。

このように大規模な組合せ最適化問題に対する一般的な解法としては、ランダム法や欲張り法[73]のような近似アルゴリズムがあり、さきに提案したGCF/LPT法もその1つである。しかしながら、ロボット制御則の計算のように繰り返し何度も実行される処理のスケジューリング問題では、アルゴリズムの計算時間が許す限り、少しでも精度の高い近似解、すなわち、準最適解を求めることが望まれる。

ここでは、最適化問題3.1の大規模な入力例に対しても、実用的な計算時間で優れた準最適解を得ることができる準最適化アルゴリズムを提案する[71],[72]。提案する準最適化アルゴリズムは、ある種の部分列挙法[73]に基づくものであり、与えられる積項の集合を分割して、解に大きな影響を与える積項の部分集合には最適化アルゴリズム3.1を、影響の小さい積項の部分集合には近似アルゴリズム（GCF/LPT法）を適用する。また、この準最適化アルゴリズムによる積項の集合の分割方法は、式(3.11)に示したような積和式の形で表されるロボット制御則では、各積項に含まれる乗算回数に大きなばらつきがあることに着目したものである。

【準最適化アルゴリズム3.2】

(1) プロセッサ数Pに等しい空の添字集合J_p $(p = 1,\ 2,\ \ldots,\ P)$を用意する。

(2) 与えられた積項の集合$\{T_j \mid j \in J\}$から、含まれる乗算回数が多いものから順番に、幾つかの積項を取り出して、積項の部分集合$\{T_j \mid j \in \hat{J} \subseteq J\}$を定義する。

(3) 積項の集合$\{T_j \mid j \in \hat{J}\}$に対して最適化アルゴリズム3.1を適用して、各積項T_jの添字jを、添字集合J_p $(1 \leq p \leq P)$へ配分する。

(4) 残りの積項の集合$\{T_i \mid i \in J - \hat{J}\}$に対してGCF/LPT法を適用して、各積項$T_i$の添字$i$を、上記の添字集合$J_p$ $(1 \leq p \leq P)$へ続けて配分する。

(5) 添字集合の集合$x = \{J_p \mid 1 \leq p \leq P\}$と、これに対する最適化問題3.1の目的関数値$E(x)$を、最適化問題3.1の準最適解として出力する。 ■

ところで、提案した準最適化アルゴリズム3.2のように、分枝限定法を利用して、大規模な組合せ最適化問題に対する優れた近似解を得るための手法としては、分枝限定法における計算を途中で打ち切り、その時点での暫定解を準最適解として採用するような単純な方法が考えられる[73]。しかし、計算を途中で打ち切ると、探索範囲に偏りが生じて、大域的な探索が行われない恐れがある。このため、探索方法に工夫が必要となるほか、計算を打ち切る適切な時点を判断することも難しい。

このほか、原問題（入力例）を幾つかの部分問題に分割して、各部分問題ごと独立に

分枝限定法によって最適解を求めるような、いわゆる分割法[73]が考えられる。しかし、一般に分割法では、原問題の分割方法が不適切であると、各部分問題に対して得られる最適解が局所的なものとなり、大域的に優れた解が得られない。

これら従来の手法に対して、提案した準最適化アルゴリズム3.2では、与えられたロボット制御則の特性を考慮して入力例を分割することにより、問題の解に大きな影響を与える部分に対してのみ、分枝限定法に基づく最適化アルゴリズム3.1を適用することができる。また、最適化アルゴリズム3.1を用いて得られた、部分的ではあるが最適な積項の配分方法（分割方法）$\{J_p \mid 1 \leq p \leq P\}$が、続いて実行されるGCF/LPT法においても考慮されることに注目されたい。すなわち、分割法のように、入力例を完全に2つの部分問題に分割して、それらを独立な問題として解く分けではない。

さらに、準最適化アルゴリズム3.2では、2種類のアルゴリズムの適用範囲の割合を指定して、得られる準最適解の精度と計算時間を調節することが可能である。従って、次節の適用例による評価において示すように、準最適化アルゴリズム3.2は、アルゴリズムの計算時間に比べて得られる解の精度が高く効率が良い。

3.7 適用例による評価

ここでは、幾つかのロボット制御則を入力例とした最適化問題 3.1 に対して、従来の近似アルゴリズム LPT 法、提案した近似アルゴリズム GCF/LPT 法、最適化アルゴリズム 3.1、準最適化アルゴリズム 3.2 に対して、それぞれの特性と有効性を評価する。対象とするロボット制御則は、前章の適用例としても用いた、極座標ロボット・アームと多関節ロボット・アームに対する逆動力学計算と線形化補償器である。また、使用した計算機は Sun SPARC Station 20 である。

▼ 例題 1 ：極座標ロボット・アームの逆動力学計算

図 3.5 に示すような 2 つの回転関節と 1 つの直動関節を持つ極座標ロボット・アームに対して、数式処理システム REDUCE を用いて、式 (2.3) に示した逆動力学計算のためのロボット制御則を、積和式の形で導出した。このロボット制御則による第 2 関節トルク f ($f = f_2$) の計算式（積項数：16）を、最適化問題 3.1 の入力例とする。

初めに、上記の最適化問題 3.1 において、プロセッサ数 Pを変えて、近似アルゴリズム LPT 法と GCF/LPT 法、および、最適化アルゴリズム 3.1 を、それぞれ適用して得られた解に対する目的関数値 Eと計算時間を表 3.1 に示す。

まず、近似アルゴリズムである LPT 法と GCF/LPT 法を比べると、目的関数値 Eから、提案した GCF/LPT 法によって得られた近似解は、従来の LPT 法で得られる近似解よりも優れていることが確認できる。また、2 つのアルゴリズムの計算時間には、大きな差がないことにも注目されたい。これは、双方のアルゴリズムの計算時間において、各積項のプロセッサへの配分よりも、数式の簡単化を伴う目的関数値 Eの評価に占める計算時間の割合が大きいためであると考えられる。

次に、表 3.1 において、近似アルゴリズム GCF/LPT 法と最適化アルゴリズム 3.1 を

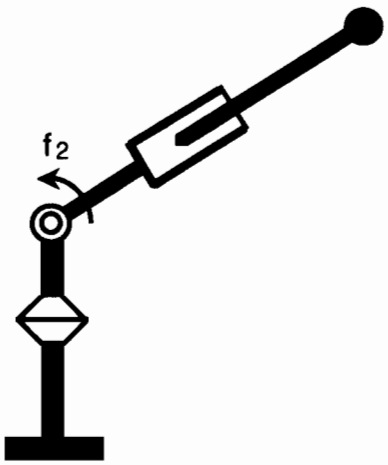


図 3．5　極座標ロボット・アーム

比較する。最適化アルゴリズム 3.1 を適用すると、明らかに、GCF/LPT 法で得られる近似解よりも優れた最適解が求まることが確認できる。また、2 つのアルゴリズムの計算時間を比べると、GCF/LPT 法の計算時間は短く、かつ、ほぼ一定であるのに対して、最適化アルゴリズム 3.1 の計算時間は、プロセッサ数 Pの異なる入力例ごとに違いが大きく、かなりの計算時間を要する場合もある。この原因としては、ヒューリスティックに積項を配分する GCF/LPT 法の計算時間が、与えられる積項の数（問題の規模）によりほぼ決まるのに対して、分枝限定法に基づく最適化アルゴリズム 3.1 では、アルゴリズムの計算時間が、探索方法に依存する部分問題の評価順序と、入力例ごとに異なる最適解の分枝図上での位置に、大きく影響されるためであると考えられる。

このような最適化アルゴリズム 3.1 の挙動について、さらに詳しく分析する。最適化アルゴリズム 3.1 における上界値 U，下界値 L，目的関数 Eの各評価回数と、暫定値 Zの更新回数を表 3.2 に示す。下界値 Lと目的関数 Eの評価回数の差が、最適化アルゴリズム 3.1 において、下界値 Lの評価により終端された部分問題の数を表す。これらの結果から、下界値 Lを用いた部分問題の終端が、数式の簡単化を伴う目的関数 Eの評価回数の削減に、大きく貢献していることが確認できる。

また、最適化アルゴリズム 3.1 では、すべての部分問題に対して上界値 U の値が評価

表 3．1　目的関数の値とアルゴリズムの計算時間

	LPT		GCF/LPT		optimal	
P	E	time [sec]	E	time [sec]	E	time [sec]
2	21	0.21	17	0.14	14	69.85
3	15	0.14	15	0.15	10	84.22
4	12	0.13	12	0.14	9	332.07
5	11	0.09	11	0.15	8	299.58
6	10	0.05	9	0.15	7	18.67
7	9	0.06	9	0.14	6	9.09
8	8	0.05	7	0.17	6	1.01

表 3．2　最適化アルゴリズムの探索効率

P	U [no.]	L [no.]	E [no.]	Z [no.]	D_r [%]
2	1223	1213	1017	3	3.732
3	3137	3133	1589	5	4.393×10^{-2}
4	14846	14842	5507	3	8.641×10^{-3}
5	17212	17207	4665	3	1.570×10^{-3}
6	1705	1704	388	2	6.234×10^{-5}
7	679	678	136	3	2.070×10^{-5}
8	74	74	23	1	3.455×10^{-6}

されることから、表3.2に示した上界値 Uの評価回数が、生成された部分問題の数である。そこで、部分問題の数が、各入力例に対して存在する許容解の総数（第2種スターリング数[74]）に占める比率 D_r [%] を求めて、同じく表3.2に示す。生成された部分問題の数が、許容解の総数に比べて非常に少ないことから、最適化アルゴリズム 3.1 による最適解の探索効率は、極めて高いと言える。

最後に、準最適化アルゴリズム 3.2 について評価する。評価の尺度として、最適化アルゴリズム 3.1 を適用する積項 T_j $(j \in \hat{J})$ の数 $|\hat{J}|$ が、与えられた積項の総数 $|J|$ に占める割合 X_r $(X_r = |\hat{J}|/|J|)$ に対して、得られた準最適解の精度 R_a $(R_a \geq 1)$ と、その求解に要した計算時間の比 R_t $(R_t \leq 1)$ を、それぞれ以下のように定義する。

$$R_a(X_r) := \frac{\text{準最適解 } x \text{ に対する目的関数値 } E(x)}{\text{最適解 } x^* \text{に対する目的関数値 } E(x^*)} \tag{3.37}$$

$$R_t(X_r) := \frac{\text{準最適化アルゴリズムの計算時間}}{\text{最適化アルゴリズムの計算時間}} \tag{3.38}$$

積項の割合が $X_r = 0$ $(\hat{J} = \emptyset)$ であるとき、準最適化アルゴリズム 3.2 は、近似アルゴリズム（GCF/LPT 法）に等しい。また、$X_r = 1$ $(\hat{J} = J)$ である場合は、最適化アルゴリズム 3.1 に等しく、次式に示す関係が成り立つ。

$$R_a(X_r) = R_t(X_r) = 1 \quad (X_r = 1) \tag{3.39}$$

準最適化アルゴリズム 3.2 を、プロセッサ数 Pの異なる入力例に対して適用した結果を図3.6に示す。図3.6では、積項の割合 X_r $(X_r = |\hat{J}|/|J|)$ に対する R_aと R_tの変化を示している。X_rが1に近づくのに伴って、R_aと R_tは共に1へ収束しており、準最適化アルゴリズム 3.2 において、最適化アルゴリズム 3.1 を適用する積項数を増やすほど、アルゴリズムの計算時間は長くなるが、得られる解の精度は向上することが確認できる。このような結果は、直観的にも予想しうるものである。

図3.6の結果において特に注目すべき点は、X_rの増加に伴い、計算時間の比 R_tよりも解の精度 R_aの方が急激に1に収束していることである。これは僅かなアルゴリズムの計算時間の増加によって、解の精度が大幅に改善される可能性があることを意味しており、提案した準最適化アルゴリズム 3.2 の妥当性を示すものである。

ところで、図3.6からも分かるように、解の精度 R_aについては単調に変化する分けではなく、X_rの増加により必ずしも解の精度が改善されるとは限らない。この原因としては、最適化アルゴリズム 3.1 による最適解が局所的なものであり、入力例によっては大域的に不適切な解となっている可能性があること、GCF/LPT 法による近似解の精度が、問題の規模に対して保証されないことなどが考えられる。

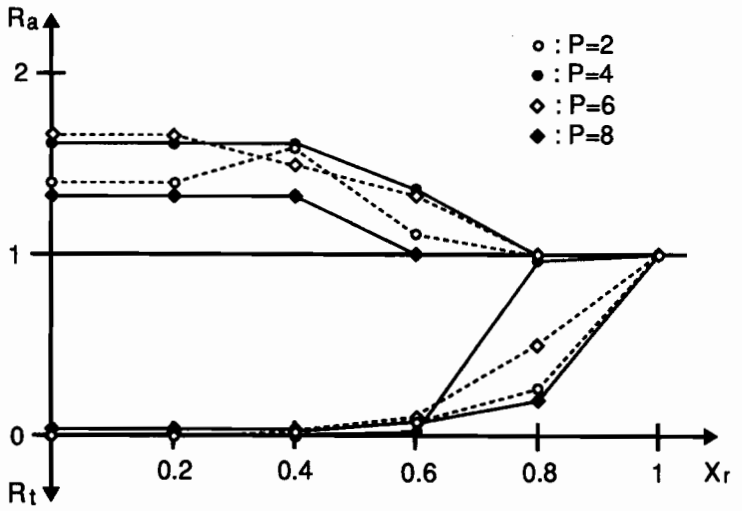


図3．6　アルゴリズムの計算時間と解の精度

▼ 例題2：多関節ロボット・アームの線形化補償器

図3.7に示すような3つの回転関節を持つ多関節ロボット・アームに対して、数式処理システムREDUCEを用いて、式(2.5)に示した線形化補償器を実現するためのロボット制御則を、積和式の形で導出した。このロボット制御則による第2関節トルクf ($f = f_2$)の計算式（積項数：19）を、最適化問題3.1の入力例とする。

初めに、上記の最適化問題3.1において、プロセッサ数Pを変えて、近似アルゴリズムLPT法とGCF/LPT法、および、最適化アルゴリズム3.1を、それぞれ適用して得られた解に対する目的関数値Eと計算時間を表3.3に示す。

また、最適化アルゴリズム3.1の挙動について、詳細に分析した結果を表3.4に示す。表3.4では、最適化アルゴリズム3.1における上界値U、下界値L、目的関数Eの評価回数と、暫定値Zの更新回数、さらに、探索の過程で生成された部分問題の数が、各入力例に対する許容解の総数に占める比率D_r [%]を示している。

最後に、プロセッサ数Pの異なる入力例に対して、準最適化アルゴリズム3.2を適用した結果を図3.8に示す。図3.8では、準最適化アルゴリズム3.2において最適化アルゴリズム3.1を適用する積項の割合X_r ($X_r = |\hat{J}|/|J|$)に対して、得られる解の精度R_aと計算時間の比R_tの変化を示している。

これら各アルゴリズムによる結果には、例題1の結果とほぼ同様の傾向が見られ、提案した3種類のアルゴリズム、すなわち、GCF/LPT法、最適化アルゴリズム3.1、準最適化アルゴリズム3.2が、対象とするロボット・アームの幾何学的な構造の違いや、ロボット制御則の種類に関わらず有効であり、かつ、実用的であることが確認できる。

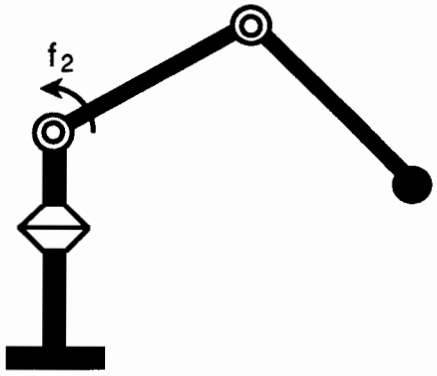


図3. 7　多関節ロボット・アーム

表3．3　目的関数の値とアルゴリズムの計算時間

	LPT		GCF/LPT		optimal	
P	E	time [sec]	E	time [sec]	E	time [sec]
2	30	0.31	29	0.40	24	3753.37
3	22	0.20	21	0.29	16	17075.50
4	19	0.19	18	0.26	13	27588.80
5	14	0.18	14	0.27	11	15210.20
6	13	0.15	12	0.28	9	96.42
7	12	0.09	10	0.26	9	1327.02
8	11	0.06	10	0.26	8	13.66

表3．4　最適化アルゴリズムの探索効率

P	U [no.]	L [no.]	E [no.]	Z [no.]	D [%]
2	14743	14705	14705	5	5.624
3	89732	89692	87767	5	4.639×10^{-2}
4	213990	213981	157001	5	1.901×10^{-3}
5	205679	205673	94154	3	1.394×10^{-4}
6	4723	4717	968	3	6.810×10^{-7}
7	58989	58982	9381	1	3.951×10^{-6}
8	1003	1000	190	2	5.866×10^{-10}

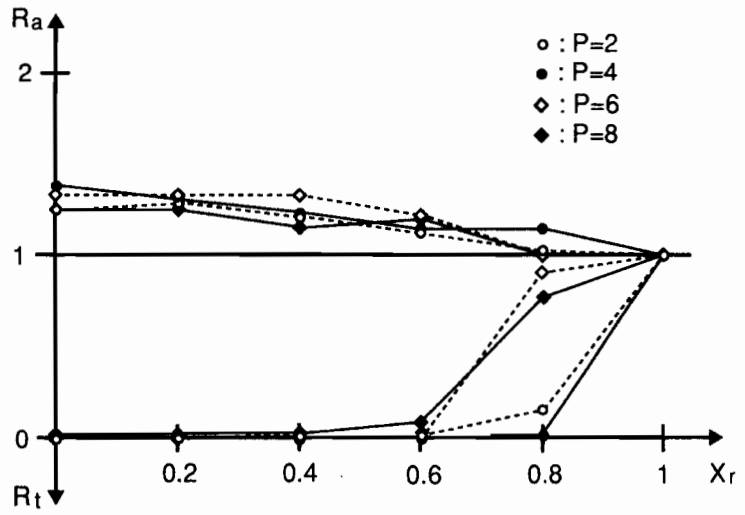


図3．8　アルゴリズムの計算時間と解の精度

3.8 結言

この章では、ロボット・アームの動的制御則における非線形な下位制御則の計算に対して、MIMD 分散メモリ型の並列計算機モデルによる並列処理手法を提案した。この並列処理手法は、積和式の形で与えられる任意のロボット制御則に対して適用することができ、ロボット・アームの幾何学的な構造の違いや、ロボット制御則の種類に関係なく有効である。また、プロセッサ間の通信量が少ないために、プロセッサの結合方法など、実際の並列計算機のハードウェア的な違いに大きく影響されることもない。

提案した並列処理手法の特徴は、ロボット制御則の計算式の並列化と、因数分解等を用いた簡単化にある。Fateman[75] の指摘にもあるように、一般に数式の並列化と簡単化は、相矛盾する目的である。しかし、プロセッサの個数が制限された場合、数式の簡単化によって各プロセッサにおける逐次処理の計算効率を高めることは、冗長な演算を多く含むロボット制御則の計算の並列処理に極めて有効である。

そこで、この章では、任意のプロセッサ数のもとで、上記の並列処理に要する時間を最短とするために、ロボット制御則の簡単化と並列化を共に考慮したスケジューリング問題（最適化問題 3.1）について考えた。さらに、このスケジューリング問題に対して、3 種類の異なる特性を持つスケジューリング・アルゴリズムを提案した。すなわち、大規模な問題に対しても適用可能な近似アルゴリズム（GCF/LPT 法）、分枝限定法に基づき最適解を厳密に求めることができる最適化アルゴリズム、さらに、これら 2 つのアルゴリズムを組み合わせて、実用的な計算時間で精度の高い近似解（準最適解）が得られる準最適化アルゴリズムである。

ところで、対象としたスケジューリング問題は、問題の規模の指数関数程度の時間計算量が予想される *NP* 困難な問題である。しかし、*NP* 困難な問題であっても、すべての入力例の求解が困難である分けではない。また、際限なく繰り返し実行されるロボット制御則の計算の並列処理では、スケジューリングのためにある程度の計算時間を要しても、最適解、あるいは、それに近い準最適解が望まれる。この意味において、ロボット制御則のスケジューリング問題に対して、ヒューリスティックな近似アルゴリズムのほかに、分枝限定法に基づく最適化アルゴリズムと準最適化アルゴリズムを提案した意義は大きい。特に、準最適化アルゴリズムについては、大規模な問題に対しても適用可能であり、僅かなアルゴリズムの計算時間の増加によって、得られる解の精度が大幅に改善される可能性があることが、幾つかの適応例において確認できた。

第4章

複数のDSPを用いたディジタル制御器の実現

4.1 緒言

この章では、複数のディジタル信号処理用プロセッサ（DSP：Digital Signal Processor）[15]-[17] を用いた並列処理によるディジタル制御器の実現方法を提案する。このディジタル制御器は、ロボット・アームの動的制御における上位制御則など、伝達関数として与えられる線形な制御則の実時間処理を目的としている。

あるサンプル信号がディジタル制御器に入力されてから、そのサンプル信号に影響された最初の信号が出力されるまでの遅延時間を、ディジタル制御器の滞在時間（Latency）と呼ぶ [76]。ロボット・アームの動的制御のように、ディジタル制御器がフィードバック制御系に組み込まれて使用される場合、ディジタル制御器における滞在時間が長くなると、制御系の応答性や安定性が損なわれる [77]。従って、フィードバック制御系に組み込まれるディジタル制御器では、そのサンプリング周期のみならず、滞在時間についても十分に短いことが要求される。

ところが、従来のディジタル・フィルタを実現するための並列処理手法は、サンプリング周期の短縮を主な目的としており、滞在時間については考慮されていない。例えば、状態方程式により特性が与えられたディジタル・フィルタを、2次元のシストリックアレイを用いて実現する典型的な並列処理手法 [78] では、状態方程式の次数によって、直列に接続されるプロセッサの数、すなわち、パイプライン処理の段数が決定される。このため、実現されるディジタル・フィルタの滞在時間は、状態方程式の次数に比例して増大する。また、滞在時間の短縮も考慮して、1次元のシストリックアレイを用いる場合 [76] でも、並列化できるプロセッサの数が、状態方程式の次数によって決まり、滞在時間をサンプリング周期よりも短くすることはできない。

提案する並列処理手法は、サンプリング周期ごとに制御則の計算を行うDSPを切り換えるものであり、与えられた伝達関数の次数などに関わらず任意の個数のDSPを並列化することができる。さらに、使用するDSPの個数の増加に伴って、ディジタル制御器のサンプリング周期を短縮することができ、その滞在時間については、サンプリング周期よりも短くすることが可能である。また、提案する並列処理手法では、DSPが計算処理

を中断してデータの交換や同期処理を行う必要がないために、DSP 特有の優れた機能である演算のパイプライン処理の効率を損なうことがない。

さらに、この章では、提案したディジタル制御器を実現する際に、その重要な性能特性であるサンプリング周期と滞在時間を、共に最短とするための多目的スケジューリング問題について考える。初めに、提案した並列処理手法において、幾つかのタイミングに関する制約条件を明らかにして、上記の多目的スケジューリング問題を、多目的の最適化問題として定式化する。次に、この多目的の最適化問題に対して、分枝限定法 [14] に基づく最適化アルゴリズムを提案する。この最適化アルゴリズムによれば、多目的の最適化問題に対するパレート（Pareto）最適解の 1 つを求めることができる。パレート最適解とは、最も優れた解であるとは言えないが、ほかの如何なる解よりも劣ることのない解の集合である [79]。

4.2 ディジタル信号処理用プロセッサの特徴

ここでは、一般的なディジタル信号処理用プロセッサ（DSP：Digital Signal Processor）の特徴について説明する。DSP とは、音声合成や音声認識などのアナログ信号をディジタル的に実時間で処理する目的で開発された専用のプロセッサである。本来はモデムなどに用途を絞ったプロセッサであったが、DSP の汎用性や経済性の向上に伴い、ロボット制御の分野においても広く使用されるようになってきている [80]。

DSP が汎用のマイクロプロセッサと比べて大きく異なる点として、乗算機能の高速化による積和演算のパイプライン処理を挙げることができる。また、DSP は以下に示すような特有のアーキテクチャを備えている [15]–[17]。

(1)　積和演算のパイプライン演算器
(2)　ハードウェア乗算器
(3)　プログラム実行部とデータ処理部の分離構造

例えば、代表的な DSP の 1 つである TMS320C25[81] について、概念的なアーキテクチャのブロック図を図 4.1 に示す。積和演算をパイプライン処理によって行うために、乗算器と加減算器（ALU）が直列に接続されていることが分かる。さらに、リアルタイム性の追求により、図 4.1 に示したアーキテクチャでは、プログラム・バスとデータ・バスが分離した構造と成っている。

DSP の優れた機能である積和演算のパイプライン処理は、演算データのレジスタへのロード、乗算、乗算結果の足し込みといった一連の処理を、パイプライン的に実行するも

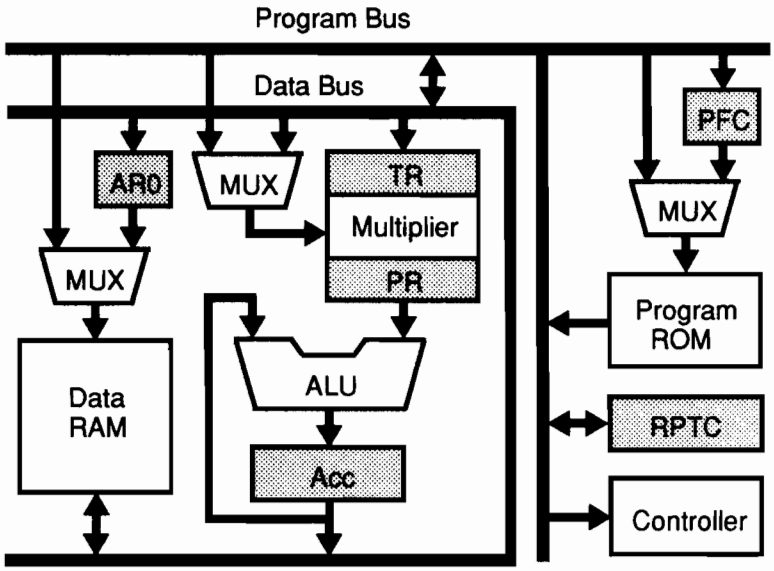


図 4. 1　DSP のアーキテクチャ

のである。ここで、DSPにより積和演算のパイプライン処理を効率的に行うためには、ある程度まとまった量のデータに対して連続的に積和演算を実行する必要がある。すなわち、連続的に処理するデータ数によって最適な計算方法が変わり、データ数が多いほど演算1回当たりに要する処理時間は短くなる。このため、与えれたプログラム内の演算順序を適切に変更して、積和演算がまとめて行われるようにプログラムを変換するDSP専用のコンパイラの研究も行われている[82]。

パイプライン処理による積和演算の特性については、実際のDSP（TMS320C25）を例に挙げて説明する。例えば、式(4.1)に示すような係数c_rとデータ$u[r]$ $(r = 1, 2, \ldots, R)$に関するR回の積和演算を実行する場合を考える。

$$c_1 * u[1] + c_2 * u[2] + \cdots + c_r * u[r] + \cdots + c_R * u[R] \tag{4.1}$$

初めに、式(4.1)の計算に必要なすべての係数とデータは、DSPのメモリ内において、図4.2に示すようなアドレスに格納されているものとする。ここで、式(4.1)の積和演算をパイプライン処理によって効率的に実行するために、それぞれ表4.1と表4.2に示すような2種類の計算方法（プログラム）が考えられる[83],[84]。

まず、表4.1に示した計算方法では、DSPの最小命令実行時間（1サイクル時間）を単位時間として、演算を開始する前に1サイクルでデータ$u[1]$のアドレス$AU1$を補助レジスタAR0に設定した後に、1サイクルでレジスタTRへ係数c_rをロードして、次の2サイクルで係数c_rとデータ$u[r]$の乗算、および、レジスタPR内の乗算結果のアキュムレータAccへの足し込みを同時に行うという処理をR回くり返す。従って、表4.1の計算方法によりR回の積和演算に要する時間は、$(3R + 1)$サイクルとなる。

一方、表4.2に示した計算方法では、演算を開始する前に4サイクルでデータ$u[1]$のアドレス$AU1$を補助レジスタAR0へ、演算回数Rをリピート・カウンタRPTCへ、係数c_1のアドレス$AC1$をプリフェッチ・カウンタPFCへそれぞれ設定した後に、続く2サイクルでレジスタTRへのデータ$u[r]$のロードと乗算、レジスタPR内の乗算結果のアキュムレータAccへの足し込み、各種カウンタの更新等をすべて同時に実行するという処理をR回くり返す。従って、表4.2に示した計算方法によりR回の積和演算に要す

address	data	address	data
$AU1$	$u[1]$	$AC1$	c_1
$AU2$	$u[2]$	$AC2$	c_2
$AU3$	$u[3]$	$AC3$	c_3
$\vdots$	$\vdots$	$\vdots$	$\vdots$
AUr	$u[r]$	ACr	c_r
$\vdots$	$\vdots$	$\vdots$	$\vdots$
AUR	$u[R]$	ACR	c_R

図4.2 メモリ内のアドレスとデータ

る時間は、$(2R+4)$ サイクルとなる。

表4.1と表4.2に示した2種類の計算方法を比較すると、R個のデータに関して連続的に積和演算を実行する場合、データ数が $R \leq 3$ ならば表4.1に示した計算方法を採用し、$R \geq 4$ ならば表4.2に示した計算方法を用いて行えば、全体の処理時間を最短にできることが分かる。すなわち、連続的に処理するデータ数によって、明らかに最適な計算方法が変わり、積和演算1回当たりに要する時間も一定ではない。ここで、連続的に処理するデータ数 Rに対して、TMS320C25 における最適な計算方法を選択して積和演算を行った場合の処理時間（サイクル数）fを図4.3に示す。関数 fのグラフの傾きが、積和演算1回当たりの処理時間を表している。

ところで、表4.1と表4.2に示した2種類の計算方法では、パイプライン処理を行うために、データ $u[r]$ $(r = 1, 2, \ldots, R)$ のアドレス AUrを、補助レジスタ AR0 を用いた間接アドレスによって指定する必要がある。このため、すべてのデータ $u[r]$ は DSP のメモリ内において、図4.2に示したように、それらのアドレスが式(4.1)の計算順序に従って連続する状態で格納されていなければならない。

さらに、DSP における積和演算の処理時間を評価する上で重要なことは、計算方法によって変わる積和演算1回当たりの処理時間から、係数のロードや各種レジスタの設定等にかかる時間を除いた、実質的な1回の積和演算（乗算）のみに要する時間は、乗算器のハードウェアによって決まり、計算方法に関わらず常に一定となることである。例えば、TMS320C25 の場合では、表4.1と表4.2に示した各計算方法（プログラム）より、乗算1回の処理時間は2サイクルであることが分かる。

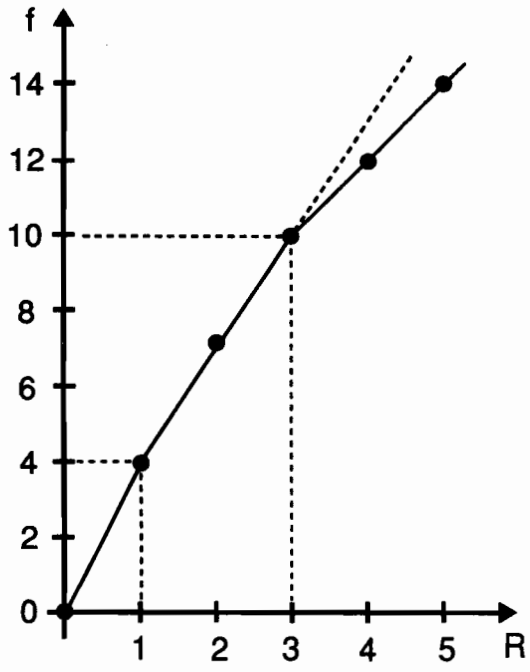


図4. 3　DSP の積和演算特性

表 4. 1 積和演算の計算方法（Type-1）

program	description	cycles
$LRAK\ 0\ AU1$	address pointer : $AR0 := AU1;$	1
$LT\ c_1$	$TR := c_1;$	1
$MPYA\ *\ +$	$PR := (AR0) * TR;\quad A_{cc} := A_{cc} + PR;$	2
	$AR0 := AR0 + 1;$	
$LT\ c_2$	$TR := c_2;$	1
$MPYA\ *\ +$	$PR := (AR0) * TR;\quad A_{cc} := A_{cc} + PR;$	2
	$AR0 := AR0 + 1;$	
⋮	⋮	⋮
$LT\ c_r$	$TR := c_r;$	1
$MPYA\ *\ +$	$PR := (AR0) * TR;\quad A_{cc} := A_{cc} + PR;$	2
	$AR0 := AR0 + 1;$	
⋮	⋮	⋮
$LT\ c_R$	$TR := c_R;$	1
$MPYA\ *\ +$	$PR := (AR0) * TR;\quad A_{cc} := A_{cc} + PR;$	2
	$AR0 := AR0 + 1;$	

表 4. 2 積和演算の計算方法（Type-2）

program	description	cycles
$LRAK\ 0\ AU1$	address pointer : $AR0 := AU1;$	1
$RPTC\ R-1$	repeat counter : $RPTC := R - 1;$	1
$MAC\ AC1\ *\ +$	$PFC := AC1;$	$2 + 2 * R$
	repeat following instructions	
	$if(RPTC \neq 0)\{$	
	$\quad TR := (AR0);\quad PR := TR * (PFC);$	
	$\quad A_{cc} := A_{cc} + PR;$	
	$\quad AR0 := AR0 + 1;\quad PFC := PEC + 1;$	
	$\quad RPTC := RPTC - 1;$	
	$\}\ else\ \{$	
	$\quad TR := (AR0);\quad PR := TR * (PFC);$	
	$\quad A_{cc} := A_{cc} + PR;$	
	$\}$	

4.3 ディジタル制御器の制御則と並列処理手法

ここでは、提案する複数のDSPを用いたディジタル制御器について説明する[83]-[86]。まず、実現するディジタル制御器は多入力1出力であり、任意の自然数kについて、ディジタル制御器の出力を$y[k]$、入力を$u_q[k]$ $(q = 1, 2, \ldots, Q)$とするとき、その入出力特性は次式に示すような差分方程式によって表される。

$$y[k] := \sum_{p=1}^{P} a_p * y[k-p] + \sum_{p=0}^{P} \sum_{q=1}^{Q} b_{pq} * u_q[k-p] \tag{4.2}$$

ロボットの動的制御では、2章でも述べたように、非線形状態フィードバックによって線形かつ非干渉なシステムに変換されたロボット・アームに対して、各関節ごと独立にサーボ系が構成される。すなわち、ロボットの動的制御における線形な上位制御則は1出力である。そこで、線形な上位制御則が1入力1出力の伝達関数によって与えられる場合は、Z変換など適当な方法で伝達関数を離散化することにより、式(4.2)で入力数$Q = 1$とした1入力1出力の差分方程式が得られる。また、上位制御則が多入力1出力の伝達関数行列として与えられる場合には、各々の伝達関数を離散化した後に分母を通分することで、やはり式(4.2)に示したような多入力1出力の差分方程式が得られる。従って、式(4.2)に示した差分方程式は、ロボットの動的制御における線形な上位制御則の一般的な離散化表現であると言える。

提案する複数のDSPを用いたディジタル制御器の構成を図4.4に示す。図4.4ではL_{DSP}

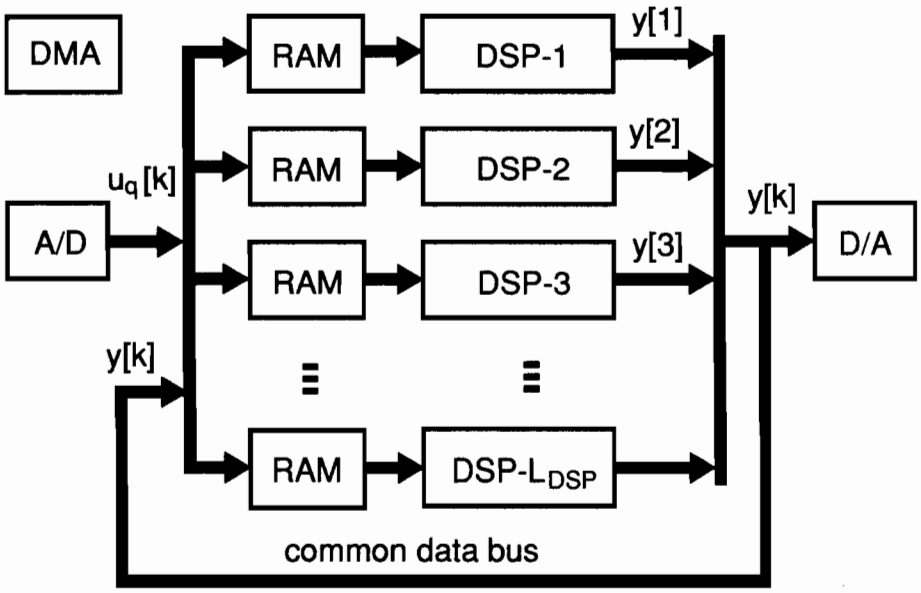


図4.4　ディジタル制御器の構成

個のDSPが並列化されており、各DSPは内部にプログラム・メモリと外部にデータ・メモリ（RAM：Random Access Memory）を持っている。また、すべてのDSP、A/D変換器、D/A変換器およびDMA（Direct Memory Access controller）は、共通データ・バスによって結合されると共に、同一の外部クロック信号により完全に同期して作動する。図4.4のブロック図に基づき、商用のDSP（TMS320C25）[81]を用いて、実際に試作したディジタル制御器のハードウェア構成を付録Fに示す[86],[89]。一方向性バッファを使用した共通データ・バスにおけるデータの流れの制御方法など、ディジタル制御器の実装方法の詳細については、付録Fを参照されたい。

図4.4のディジタル制御器において、Q個の入力$u_q[k]$ $(Q \geq q \geq 1)$は、A/D変換器により$u_Q[k]$, ..., $u_2[k]$, $u_1[k]$と降順にサンプリングされて、DMAによりすべてのDSPのRAMに対して同時に書き込まれる。提案する並列処理手法では、並列化するDSPの個数をL_{DSP}とするとき、任意のkに対して出力$y[k]$を求めるための式(4.2)全体の計算処理を、$[\{(k-1)\ mod\ L_{DSP}\}+1]$番目のDSPに割り当てる。すなわち、$y[1]$はDSP-1に、$y[2]$はDSP-2によりそれぞれ計算される。また、出力$y[k+L_{DSP}]$は、$y[k]$の計算が終了した後で、再び$[\{(k-1)\ mod\ L_{DSP}\}+1]$番目のDSPによって計算される。これらの出力$y[k]$は、算出されると直ちに、それらを計算した各DSPによってD/A変換器へ送られ出力される。また、同時に共通データ・バスを経て、すべてDSPのRAMに対しても書き込まれ、ほかのDSPによる他の出力の計算に用いられる。

このように、提案したディジタル制御器において、DMAあるいは各DSPによるRAMへの入出力データの書き込みは、常にブロードキャストによって行われる。従って、任意の時刻において、図4.4に示したすべてのRAMは同じデータを保持している。さらに、RAMとしてデュアルポート・メモリを採用することにより、各RAMに対するデータの読み出しと書き込みは、同時に行うことができる。

図4.5のガント・チャートは、制御則の入力数$Q=2$、DSPの個数$L_{DSP}=2$とした場合について、図4.4のディジタル制御器におけるデータ処理のタイミングを示している。2種類の入力$u_2[k]$と$u_1[k]$は、A/D変換器によりサンプリング周期N_Sごとにディジタル制御器へ取り込まれる。ここで、N_uは1つの入力データ$u_q[k]$のサンプリングと、RAMへの書き込みに要する時間である。

次に、各DSPは、式(4.2)に示した差分方程式（制御則）の計算を、1サンプリング周期N_Sづつの時間差を取って実行し、担当する出力$y[k]$の値を求める。さらに、これらの出力$y[k]$は、サンプリング周期N_Sごとに、共通データ・バスを経て、すべてのRAMとD/A変換器へ送られる。例えば、DSP-2によって計算された出力$y[2]$は、共通データ・バスを介してDSP-1とDSP-2のRAMに書き込まれ、これらのDSPが、それぞれ次の出力$y[3]$と$y[4]$を計算する際に用いられる。ここで、任意のkについて、最初の入力$u_Q[k]$ $(Q=2)$のサンプリングが開始されてから、出力$y[k]$が算出されるまでに要する遅延時間が、ディジタル制御器の滞在時間N_Lである。

ところで、さきに述べたように、提案したディジタル制御器において、各DSP、A/D変換器、D/A変換器およびDMAは、完全に同期して作動する。従って、図4.5に示したガント・チャートにおいて、各DSPが出力$y[k]$の計算に要する処理時間はもとより、

サンプリング周期 N_S や滞在時間 N_L、A/D 変換時間 N_u なども、DSP の最小命令実行時間（1サイクル時間）を単位時間とする適当な整数によって表される。

さらに、図4.5のガント・チャートからも推察できるように、各DSP間における通信処理、すなわち、出力データ $y[k]$ を交換する時刻とデータの整合性は、それらが算出されるタイミングと使用されるタイミングに依存している。また、入力 $u_q[k]$ $(Q \geq q \geq 1)$ と出力 $y[k]$ は、RAM への書き込み時刻の差によって、共通データ・バス上でのデータ競合を回避する必要がある。このような並列処理におけるタイミングの調整は、次節で述べるスケジューリング問題の制約条件として考慮される。

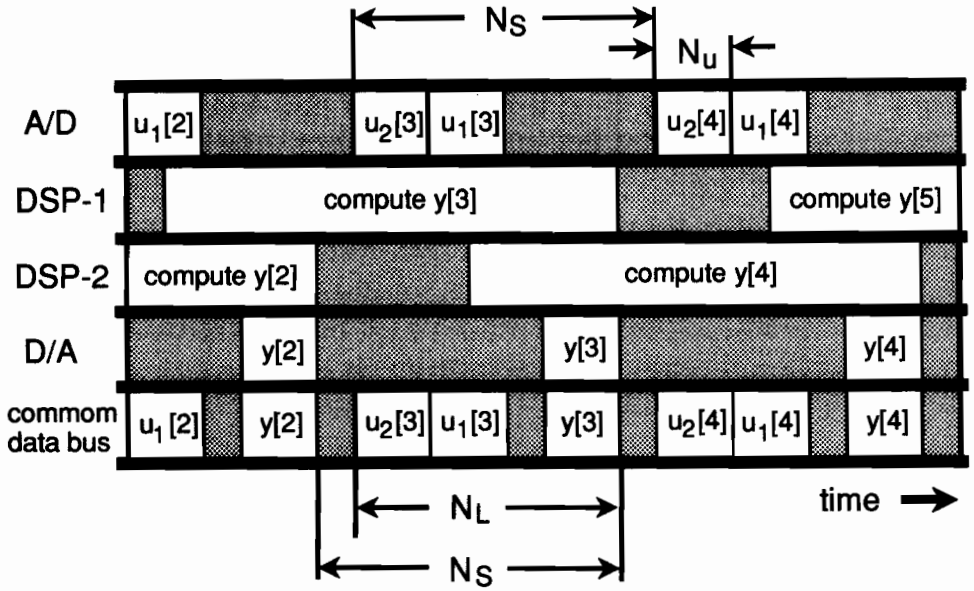


図 4. 5　ガント・チャート

4.4 多目的スケジューリング問題の定式化

ここでは、与えられた個数のDSPを用いたディジタル制御器により、実現可能なサンプリング周期N_Sと滞在時間N_Lを、共に最短とするための多目的スケジューリング問題について考える。初めに、各DSPのプログラムに相当する計算処理の記述方法を定義する。次に、複数のDSPによる並列処理における3つの制約条件、すなわち、因果性条件、滞在時間条件、および、DSP個数条件を明らかにする。最後に、上記の多目的スケジューリング問題を、多目的の最適化問題として定式化する[87],[88]。

4.4.1 各DSPにおける計算処理

ディジタル制御器において、並列化された各DSPは、1サンプリング周期ごとの時間差を取って、すべて同じ計算順序に従い、式(4.2)に示した制御則（差分方程式）の計算を実行する。このとき、図4.5に示したガント・チャートからも推察できるように、実現可能なサンプリング周期N_Sと滞在時間N_Lは、式(4.2)の制御則に含まれる積和演算の計算順序に依存する。従って、スケジューリング問題では、式(4.2)に示した制御則の最適な計算順序を求めることになる。

最初に、式(4.2)の制御則に含まれる非零係数の添字を要素とするリスト（全順序集合）A, Bを、以下のように定義する。零係数をリストから除くのは、これらに関する無意味な演算を行わないためである。

$$\left[\begin{array}{lcl} A & := & \{p \mid P \geq p \geq 1,\ a_p \neq 0\} \\ B & := & \{(p,q) \mid P \geq p \geq 0,\ Q \geq q \geq 1,\ b_{pq} \neq 0\} \end{array}\right. \tag{4.3}$$

ここで、式(4.2)の制御則の計算に必要な出力データ$y[k-p]$ $(P-1 \geq p \geq 1)$と入力データ$u_q[k-p]$ $(P \geq p \geq 0,\ Q \geq q \geq 1)$は、すべてのRAM内において、図4.6に示すような時系列順に格納されている。そこで、図4.6のデータ配置に対応して、式(4.3)のリストA, Bの要素は辞書式順序（$\succ$）に並んでいるものとする。すなわち、$\hat{p} > p$ならば$\hat{p} \succ p$, $(\hat{p},\hat{q}) \succ (p,q)$。また、$\hat{p} = p$かつ$\hat{q} > q$ならば$(\hat{p},\hat{q}) \succ (p,q)$である。

さらに、リストA, Bを重複なく、それぞれ$(N+1)$個のサブリストに分割する。このとき、サブリストA_n, B_nの個数$(N+1)$は任意であり、あとで述べるスケジューリング問題を解くことによって決定される。

$$\left[\begin{array}{lcl} A & = & A_N \cup \cdots \cup A_n \cup \cdots \cup A_1 \cup A_0 \\ B & = & B_N \cup \cdots \cup B_n \cup \cdots \cup B_1 \cup B_0 \end{array}\right. \tag{4.4}$$

ただし、各サブリストA_n, B_n $(N \geq n \geq 0)$の要素も、それぞれ辞書式順序に並んでいる。また、$m > n$ならば、A_m（B_m）のすべての要素はA_n（B_n）の要素よりも大きい。さらに、サブリストA_n, B_nには、空のリストも認めるものとする。

address	data	address	data
$AY(1)$	$y[k-P]$	$AU(1)$	$u_Q[k-P]$
$AY(2)$	$y[k-P+1]$	⋮	⋮
$AY(3)$	$y[k-P+2]$	$AU(Q-1)$	$u_2[k-P]$
⋮	⋮	$AU(Q)$	$u_1[k-P]$
⋮	$y[k-p]$	$AU(Q+1)$	$u_Q[k-P+1]$
⋮	⋮	⋮	⋮
$AY(P-1)$	$y[k-2]$	⋮	$u_q[k-p]$
$AY(P)$	$y[k-1]$	⋮	⋮
$AY(P+1)$	$y[k]$	$AU(Q*P+1)$	$u_Q[k]$
		$AU(Q*P)$	$u_{Q-1}[k]$
		⋮	⋮
		$AU(Q*P+Q)$	$u_1[k]$

図4. 6　RAM内のデータ・アドレス

ここで、式(4.4)に示したサブリストの集合 $\{A_n, B_n \mid N \geq n \geq 0\}$ に基づき、式(4.2)に示した制御則の計算順序を、次の手順4.1によって定める。

【手順4.1】
begin
1 : $Sum := 0;\ \ Mul := 0;$　// 初期化処理
2 : $for\ n = N\ down\ to\ 0\ do\ \{$
3 :　　$for\ p \in A_n\ do\ \{$　// A_nに関する積和
4 :　　$Sum := Sum + Mul;\ \ Mul := a_p * y[k-p];$
5 :　　$\}$
6 :　　$for\ (p,q) \in B_n\ do\ \{$　// B_nに関する積和
7 :　　$Sum := Sum + Mul;\ \ Mul := b_{pq} * u_q[k-p];$
8 :　　$\}$
9 : $\}$
10 : $y[k] := Sum + Mul;$
end. ■

上記の手順4.1において、変数 Sum と Mulは、それぞれ図4.1に示したDSPのアーキテクチャのレジスタAccとPRに対応する。従って、手順4.1の4行目、あるいは、7行目に示した加算と乗算は、1つの積和演算として同時に実行される。

手順4.1における3~5行目の処理をサブリスト A_nに関する積和と呼び、A_nのすべての要素 p に対して、その順番に4行目の積和演算を繰り返し実行することを意味する。例えば、$A_n = \{5, 4\}$ であるとすると、続けて $Sum := Sum + Mul;\ Mul := a_5 * y[k-5];$

と $Sum := Sum + Mul;\ Mul := a_4 * y[k-4];$ を実行する。

同様に、手順4.1における6~8行目の処理を、サブリスト B_n に関する積和と呼び、B_n のすべての要素 (p,q) に対して、その順番に7行目の積和演算を繰り返し実行することを意味する。このように、A_n，B_n は共にリストであり、含まれる要素の並び方によって、積和演算の実行順序が決められていることに注意されたい。

次に、手順4.1の処理時間について考える。DSPはパイプライン処理により高速に積和演算を実行できる。DSPが手順4.1の A_n，B_n に関する積和をパイプライン処理するためには、4.2節で述べたように、$p \in A_n$，$(p,q) \in B_n$ に対応するデータ $y[k-p]$，$u_q[k-p]$ が、RAM内において、それぞれ演算の実行順序に従って連続するアドレスに格納されている必要がある。そこで、図4.6に示したデータ配置に基づき、各データのアドレスが連続であるか否かを判定するために、リスト A，B の要素に対して、以下に示すような連続性という概念を導入する。

【定義4.1】(連続性)

式(4.3)のリスト A に含まれる2つの要素 $p \in A$，$\hat{p} \in A$ に対応するデータ $y[k-p]$，$y[k-\hat{p}]$ が、RAM内で隣接したアドレスに格納されているとき、それらの要素は連続であると言う。また、式(4.3)のリスト B の要素についても、同様に連続性を定義する。ここで、図4.6に示したデータ・アドレスより、リスト A あるいは B に含まれる2つの要素が連続となるのは、以下に示す何れかの場合である。

(1) $p,\ \hat{p} \in A\ (p > \hat{p})$ が、$p = \hat{p} + 1$ ならば p と $\hat{p}$ は連続である。

(2) $(p,1),\ (\hat{p},Q) \in B\ (p > \hat{p})$ が、$p = \hat{p} + 1$ ならば $(p,1)$ と $(\hat{p},Q)$ は連続である。

(3) $(p,q),\ (p,\hat{q}) \in B\ (q > \hat{q})$ が、$q = \hat{q} + 1$ ならば (p,q) と $(p,\hat{q})$ は連続である。 ■

さらに、上記の定義4.1に基づき、各サブリスト A_n，B_n に対して、以下のような条件4.1を新たに付加する。この条件4.1は、手順4.1の A_n，B_n に関する積和において、RAM内におけるデータ・アドレスの連続性を保証するものである。

【条件4.1】(連続性)

式(4.4)に示した分割方法 $\{A_n,\ B_n \mid N \geq n \geq 0\}$ において、各サブリスト A_n あるいは B_n に含まれるすべての要素は、並べられた順序に従って連続でなければならない。例えば、式(4.3)で $A := \{4,\ 2,\ 1\}$ と与えられたとすると、不連続な要素 $4 \in A$ と $2 \in A$ を、同一のサブリスト A_n には配分できない。 ■

式(4.4)に示した A，B の分割方法 $\{A_n,\ B_n \mid N \geq n \geq 0\}$ において、A_n，B_n の個数 $(N+1)$ は任意であり、空のリストも認めている。従って、上記の条件4.1は、手順4.1に基づく積和演算の計算順序に対して、何ら制約を加えない。

条件4.1が満たされるとき、手順4.1の A_n および B_n に関する積和は、それぞれDSPのパイプライン処理によって最短時間で実行できる。さらに、その処理時間に関して、以下に示す仮定4.1が成り立つ。仮定4.1は、さきに4.2節で述べた、一般的なDSPによる積和演算のパイプライン処理の特性を定式化したものである。

【仮定4.1】（積和演算）

サブリスト A_n（B_n）が条件4.1の連続性を満たすとき、手順4.1の A_n（B_n）に関する積和において、サブリスト A_n（B_n）の最初の要素から r 番目の要素までの積和演算に要するサイクル数は、A_n（B_n）に含まれる要素数 $|A_n|$（$|B_n|$）と演算回数 r に依存する。すなわち、これらの積和演算に要するサイクル数 $f(r;|A_n|)$（$f(r;|B_n|)$）は、$R = |A_n|$（$R = |B_n|$）として、次式で定義される関数 $f(r;R)$ から求められる。

$$f(r;R) := \alpha(R) * r + \beta(R) \tag{4.5}$$

ただし、$\alpha(R)$, $\beta(R)$ $(R \geq 0)$ は、以下のような性質を持つ。

$$\left[\begin{array}{rcll} \alpha(0) & = & \beta(0) = 0, & R = 0. \\ \alpha(R) & \geq & \alpha(R+1), & R \geq 1. \\ \beta(R) & \leq & \beta(R+1), & R \geq 1. \end{array}\right.$$

■

仮定4.1の式(4.5)において、$\alpha(R)$ はDSPが積和演算を1回実行するのに要するサイクル数を表し、$\beta(R)$ はDSPが積和演算のパイプライン処理を開始する前のレジスタの設定などに要するサイクル数を表している。これら$\alpha(R)$, $\beta(R)$ がサブリストの要素数 Rに依存するのは、1回のパイプライン処理により連続的に処理する積和演算の総数 R によって、全体の処理時間を最短とするための計算方法（プログラム）が異なるためである。すなわち、演算回数 Rが大きくなると、積和演算の1回当たりの処理時間$\alpha(R)$ は小さくなるが、逆にパイプライン処理の準備にかかる時間$\beta(R)$ は大きくなる。従って、式(4.5)における $(\alpha(R),\ \beta(R))$ は、DSPが R回の積和演算を最短の処理時間で実行するための最適な計算方法を示していると言える。

例えば、4．2節で紹介したDSP（TMS320C25）の場合、式(4.5)で定義した積和演算の処理時間を示す関数 $f(r;R)$ における $(\alpha(R),\ \beta(R))$ は、表4.1と表4.2に示した2種類の計算方法より、以下のように決まる。

$$(\alpha(R),\ \beta(R)) := \begin{cases} (0,\ 0), & R = 0. \\ (3,\ 1), & 1 \leq R \leq 3. \\ (2,\ 4), & 4 \leq R. \end{cases} \tag{4.6}$$

ここで、1回のパイプライン処理で実行される積和演算の総数が R回であるとき、それに要する最短の処理時間（サイクル数）は $f(R;R)$ と表される。以降、簡単のために、手順4.1の A_n, B_nに関する積和に要する処理時間を、次の式(4.7)で定義する $F(A_n)$, $F(B_n)$ により表すものとする。

$$\left[\begin{array}{rcl} F(A_n) & := & f(|A_n|;|A_n|) \\ F(B_n) & := & f(|B_n|;|B_n|) \end{array}\right. \tag{4.7}$$

以降では、手順4.1に基づいて、DSPによるデータ処理のタイミングを検討することにより、複数のDSPを用いた並列処理における3つの制約条件、すなわち、因果性条件、滞在時間条件、および、DSP個数条件を明らかにする。

4.4.2 因果性条件

提案するディジタル制御器において、出力$y[k]$は$[\{(k-1)\ mod\ L_{DSP}\}+1]$番目のDSPが、手順4.1に示した積和演算を行うことによって得られる。その際、このDSPが乗算$a_p * y[k-p]$を行うためには、それ以前に$[\{(k-p-1)\ mod\ L_{DSP}\}+1]$番目のDSPにより出力$y[k-p]$が算出され、RAMに書き込まれている必要がある。同様に、乗算$b_{pq} * u_q[k-p]$を実行するためには、それ以前に入力$u_q[k-p]$がA/D変換器によりサンプリングされ、RAMに書き込まれている必要がある。このようなデータ処理における時間的な制約を因果性条件と呼ぶ。

初めに、出力データ$y[k-p]$に対する因果性条件について考える。$y[k-p]$に対応する係数a_pの添字pが、あるサブリストA_n $(N \geq n \geq 0)$のr番目の要素であるとする。このとき、$y[k-p]$に関して因果性条件が満たされるためには、手順4.1において、乗算$a_p * y[k-p]$ $(p \in A_n)$が実行されてから$y[k]$が算出されるまでに要する処理時間が、$y[k-p]$が算出されてから$y[k]$が算出されるまでに経過する時間、すなわち、pサンプリング周期$(p * N_S)$よりも短くなければならない。

そこで、手順4.1に基づく上記の処理時間について考える。まず、手順4.1の4行目において、乗算$a_p * y[k-p]$ $(p \in A_n)$が開始されてから、3~5行目のA_nに関する積和が終了するまでに要するサイクル数$N_f(r;|A_n|)$は、pをA_nのr番目の要素、$R = |A_n|$として、次式で定義する関数$N_f(r;R)$から求められる。

$$N_f(r;R) := f(R;R) - f(r;R) + \Delta \tag{4.8}$$

ただし、DSPが1回の乗算に要する実質的なサイクル数をΔとする。4.2節でも述べたように、このΔは乗算器のハードウェアによって決まる定数であり、$\Delta \leq \alpha(R)$の関係を満たす。例えば、前述のTMS320C25では$\Delta = 2$である。

続いて、手順4.1において実行されるB_nに関する積和に要するサイクル数は、式(4.7)で定義したように$F(B_n)$である。また、A_m，B_m $(n-1 \geq m \geq 0)$に関する積和に要する処理時間についても、同様に$F(A_m)$，$F(B_m)$と表すことができる。

さらに、手順4.1の10行目の加算と、出力$y[k]$のRAMとD/A変換器への書き込みに要するサイクル数をN_{out}とすると、出力データ$y[k-p]$ $(p \in A_n)$に対する因果性条件は、以下に示すような不等式として表される。

$$p * N_S \geq N_f(r;|A_n|) + F(B_n) + \cdots + F(A_0) + F(B_0) + N_{out} + 1 \tag{4.9}$$

ただし、$p \in A_n$はA_nのr番目の要素であるとする。

式(4.9)の左辺は、因果性条件を考えている出力データ$y[k-p]$が、$y[k]$の算出される

pサンプリング周期前に算出されることを意味している。また、式(4.9)の右辺は、手順4.1において、乗算$a_p * y[k-p]$ $(p \in A_n)$が開始されてから$y[k]$が算出されるまでに要するサイクル数に、1サイクルの余有時間を加えたものである。この1サイクルの余有時間は、異なるDSPによる出力データ$y[k-p]$のRAMへの書き込みと読み出しが、同時に発生することを防ぐためのものである。

同様にして、入力データ$u_q[k-p]$に関する因果性条件について考える。手順4.1において、$u_q[k-p]$に対応する$(p,q) \in B_n$が、あるB_n $(N \geq n \geq 0)$のr番目の要素であったとする。さらに、任意のkについて$u_q[k]$がRAMに書き込まれてから、$y[k]$が算出されるまでに要するサイクル数を$N_L(q)$とすると、入力データ$u_q[k-p]$ $((p,q) \in B_n)$に対する因果性条件は、次式の不等式によって表される。

$$p * N_S + N_L(q) \geq N_f(r; |B_n|) + F(A_{n-1}) + \cdots + F(B_0) + N_{out} + 1 \tag{4.10}$$

ただし、$(p,q) \in B_n$はB_nのr番目の要素であるとする。

上記の式(4.9)，式(4.10)に示した$y[k-p]$と$u_q[k-p]$に関する因果性条件を、それぞれタイミング・チャートを用いて表すと、図4.7のようになる。

ところで、任意のkに対するQ個の入力は、$u_Q[k]$, ..., $u_1[k]$の順番で、A/D変換器によりサンプリングされて、DMAによってRAMへ書き込まれる。これらQ個の入力$u_q[k]$ $(Q \geq q \geq 1)$が、共通データ・バス上でデータ競合しないためには、各入力$u_q[k]$が適当な間隔をあけてサンプリングされる必要がある。そこで、1つの入力$u_q[k]$のサンプリングとRAMへの書き込みに要するサイクル数（定数）をN_uとすると、あるkに対して、入力$u_q[k]$がRAMに書き込まれてから、出力$y[k]$が算出されるまでに要するサイクル数$N_L(q)$は、次の関係を満たさなければならない。

$$N_L(q+1) = N_L(q) + N_u, \quad (q = 1, 2, \ldots, Q-1) \tag{4.11}$$

また、ディジタル制御器の滞在時間N_Lとは、最初の入力$u_Q[k]$のサンプリングが開始されてから、出力$y[k]$が算出されるまでに要する遅延時間であるので、

$$N_L = N_L(Q) + N_u \tag{4.12}$$

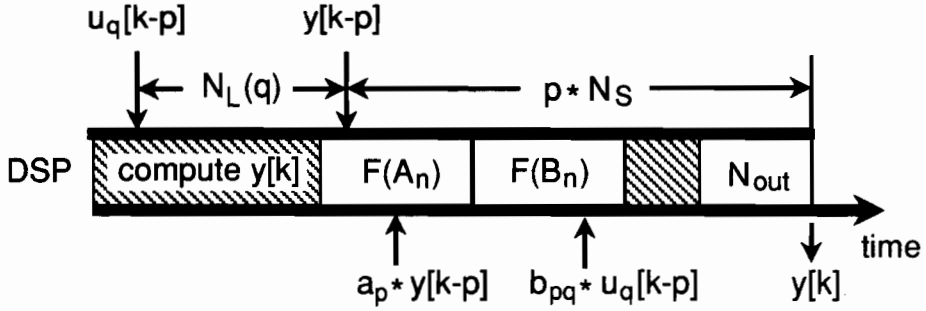


図 4. 7　因果性条件

ここで、式(4.11)，式(4.12)より、次式の関係が成り立つ。

$$N_L = N_L(q) + (Q - q + 1) * N_u \tag{4.13}$$

式(4.13)を用いて式(4.10)から$N_L(q)$を消去すると、入力データ$u_q[k-p]$に対する因果性条件は、次式のように滞在時間N_Lを陽として表すことができる。

$$\begin{aligned} p * N_S + N_L \quad \geq \quad & (Q - q + 1) * N_u + N_f(r; |B_n|) \\ & + F(A_{n-1}) + \cdots + F(B_0) + N_{out} + 1 \end{aligned} \tag{4. 14}$$

ただし、$(p, q) \in B_n$はB_nのr番目の要素であるとする。

4.4.3 滞在時間条件

はじめに述べたように、フィードバック制御系に組み込まれるディジタル制御器では、その滞在時間N_Lはサンプリング周期N_Sよりも短くなければならない。すなわち、滞在時間N_Lの上限値はサンプリング周期N_Sである。

一方、滞在時間N_Lの下限値は、Q個の入力$u_q[k]$ $(Q \geq q \geq 1)$が、式(4.11)に示したように一定の間隔N_uでサンプリングされることから、入力数Qと間隔N_uによって制限される。従って、滞在時間N_Lは、次の式(4.15)に示す関係を満たす必要がある。

$$N_S \geq N_L \geq Q * N_u \tag{4.15}$$

ここで、式(4.15)を滞在時間条件と呼ぶものとする。入力$u_q[k]$ $(Q \geq q \geq 1)$と滞在時間条件の関係は、図4.8のように表すことができる。

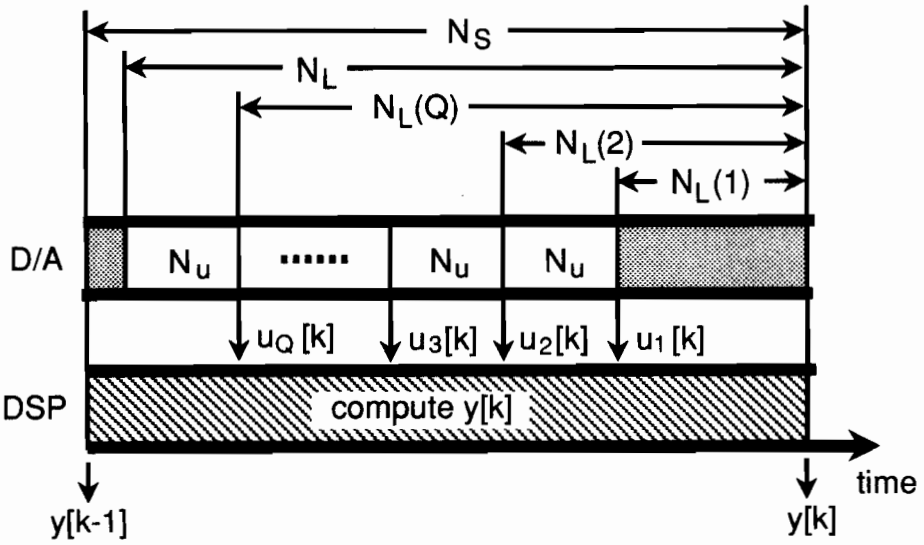


図4. 8　滞在時間条件

4.4.4 DSP 個数条件

提案したディジタル制御器において、DSP の個数 L_{DSP} は任意に与えられる。また、出力 $y[k-L_{DSP}]$ と $y[k]$ は、共に $[\{(k-1)\ mod\ L_{DSP}\}+1]$ 番目の DSP により計算される。従って、この DSP は出力 $y[k-L_{DSP}]$ の計算を終了してから、L_{DSP} サンプリング周期（$L_{DSP} * N_S$）以内に、次の出力である $y[k]$ の計算を完了する必要がある。すなわち、DSP が手順 4.1 に基づき 1 つの出力 $y[k]$ の計算に要する全サイクル数を N_{all} とすると、次の関係が成り立たなければならない。

$$N_S * L_{DSP} \geq N_{all} \tag{4.16}$$

式 (4.16) において、N_{all} は手順 4.1 の実行に必要な全サイクル数に、出力 $y[k]$ を算出した後に、手順 4.1 の先頭へ戻るための分岐命令などに要するサイクル数 N_{bak} を加えたものである。従って、手順 4.1 の 1 行目の初期化処理に要するサイクル数を N_{int} とすると、1 つの出力の計算に要する全サイクル数 N_{all} は、次のようになる。

$$N_{all} := N_{int} + F(A_N) + F(B_N) + \cdots + F(A_0) + F(B_0) + N_{out} + N_{bak} \tag{4.17}$$

式 (4.16) に式 (4.17) を代入すると、DSP 個数条件として、次式が得られる。

$$\begin{aligned} N_S * L_{DSP} \ \geq\ & N_{int} + F(A_N) + F(B_N) + \cdots + F(B_1) \\ & + F(A_0) + F(B_0) + N_{out} + N_{bak} \end{aligned} \tag{4. 18}$$

式 (4.18) に示した DSP 個数条件は、図 4.9 のように表すことができる。

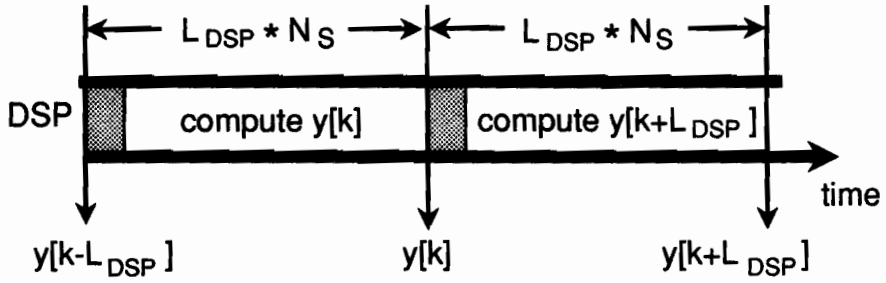


図 4. 9　DSP 個数条件

4.4.5 多目的スケジューリング問題

任意に与えられた個数のDSPを用いてディジタル制御器を実現する際に、ディジタル制御器の重要な性能特性であるサンプリング周期N_Sと滞在時間N_Lを共に最短とするような、制御則に含まれる積和演算の計算順序を求めるスケジューリング問題は、以下のような組合せ最適化問題として定式化できる。ここで、最適化問題4.0の許容解であるリストA，Bの分割方法$\{A_n,\ B_n \mid N \geq n \geq 0\}$には、サブリスト$A_n$，$B_n$の個数の決定も含まれていることに注意されたい。

【最適化問題4.0】
入力： 係数の添字リストA，Bと、DSPの個数L_{DSP}。
目的： 条件4.1（連続性）を満たすA，Bの分割方法$\{A_n,\ B_n \mid N \geq n \geq 0\}$において、式(4.9)，(4.14)の因果性条件、式(4.15)の滞在時間条件、式(4.18)のDSP個数条件を、すべて満たす最短のサンプリング周期N_S^*と滞在時間N_L^*、および、それらを与える分割方法$\{A_n^*,\ B_n^* \mid N^* \geq n \geq 0\}$を求める。 ■

十分に大きなサンプリング周期N_Sと、式(4.15)の滞在時間条件を満たす十分に大きな滞在時間N_L $(N_S \geq N_L)$のもとでは、式(4.9)，(4.14)に示した因果性条件と、式(4.18)に示したDSP個数条件を表わす各不等式は、常に成り立つ。従って、上記の最適化問題4.0に対する許容解は必ず存在する。

4.5 分枝限定法に基づく最適化アルゴリズム

ここでは、前節の最適化問題 4.0 に対して、最適化アルゴリズムを提案する [86]–[88]。最適化問題 4.0 において、実現可能なサンプリング周期 N_S と滞在時間 N_L の範囲に関する幾何学的な解釈は以下の通りである。まず、十分に大きな N_S と N_L のもとで、式 (4.9), (4.14) の因果性条件と、式 (4.18) の DSP 個数条件を表す各不等式は常に成り立つ。さらに、式 (4.15) の滞在時間条件（$N_S \geq N_L$）を考慮すると、最適化問題 4.0 の全ての許容解に対する目的関数空間 $(N_S,\ N_L)$ は、概ね図 4.10 のようになると予想される。

最適化問題 4.0 は 2 つの目的関数 N_S, N_L を持つため、その許容解の善し悪しは、一方の目的関数値のみでは決まらない。そこで、最適化問題 4.0 に対する弱パレート（Pareto）最適解とパレート最適解 [79] を定義する。まず、最適化問題 4.0 の許容解の全体集合を X、ある許容解 $x \in X$, $x = \{A_n,\ B_n \mid N \geq n \geq 0\}$ に対して実現可能な最短のサンプリング周期を $N_S(x)$、最短の滞在時間を $N_L(x)$ とする。

【定義 4.2】（弱パレート最適解）

$x^* \in X$ が最適化問題 4.0 の弱パレート最適解であるとは、

$$\nexists x \in X\ (\ N_S(x) < N_S(x^*)\ \wedge\ N_L(x) < N_L(x^*)\ )$$

■

【定義 4.3】（パレート最適解）

$x^* \in X$ が最適化問題 4.0 のパレート最適解であるとは、

$$\begin{aligned}\nexists x \in X\ (\ N_S(x) \leq N_S(x^*)\ \wedge\ N_L(x) < N_L(x^*)\ )\\ \vee\ (\ N_S(x) < N_S(x^*)\ \wedge\ N_L(x) \leq N_L(x^*)\ )\end{aligned}$$

■

ある許容解 x^* が最適化問題 4.0 のパレート最適解であるとは、それよりも優れた許容解が X 内に存在しないことを意味する。最適化問題 4.0 の弱パレート最適解とパレート最適解に、それぞれ対応する目的関数 $(N_S,\ N_L)$ の範囲を図 4.10 に示す。

提案する最適化アルゴリズムは、2 つの目的関数を持つ最適化問題 4.0 を、2 つの異なる単一目的の最適化問題に分解して考えることにより、最適化問題 4.0 に対するパレート最適解の 1 つを求めるものである。すなわち、最初に $N_L = N_S$ の制約条件のもとで、サンプリング周期 N_S のみを最短とする最適化問題 4.1 を解いて、最短サンプリング周期 N_S^* を求める。次に、得られた N_S^* を新たな制約条件として、滞在時間 N_L を最短とする最適化問題 4.2 を解いて、その最適解を最適化問題 4.0 の解とする。また、各最適化問題の求解には、分枝限定法を採用する。このとき、定義 4.2 より最適化問題 4.1 の最適解 $x1^*$

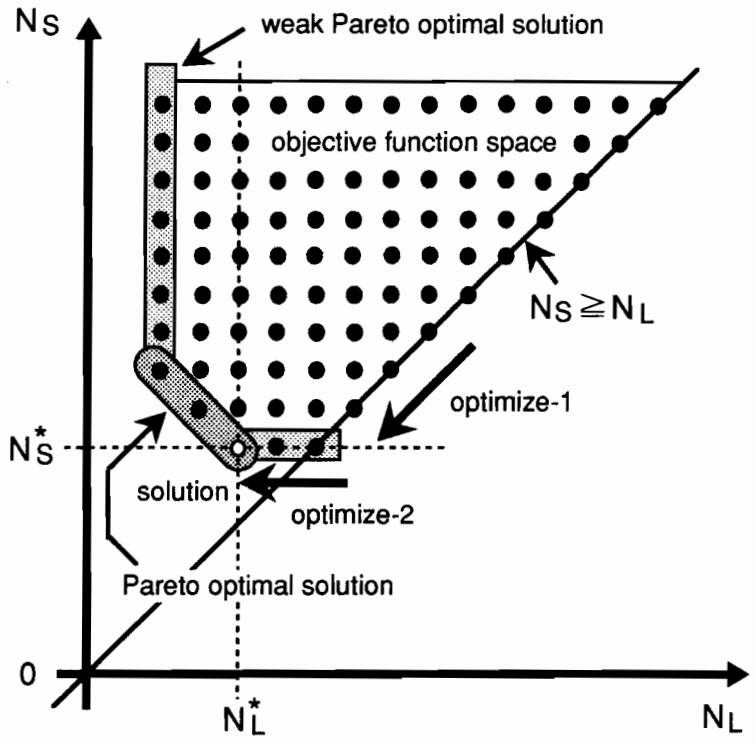


図4. 10 目的関数空間 $(N_S,\ N_L)$

は、最適化問題4.0の弱パレート最適解である。また、定義4.3より最適化問題4.2の最適解$x2^*$は、最適化問題4.0のパレート最適解となる。

4.5.1 サンプリング周期の最適化問題

多目的の最適化問題4.0から、サンプリング周期N_Sのみを最短とすることを目的とした最適化問題4.1を導出する。まず、式(4.15)の滞在時間条件を満たす最大の滞在時間N_Lの値は、サンプリング周期N_Sの値に等しい。そこで、滞在時間を$N_L = N_S$とすると、式(4.15)の滞在時間条件は、次式のようにN_Sの下限値を与える。

$$N_S \geq Q * N_u \tag{4.19}$$

さらに、式(4.14)の入力データ$u_q[k-p]$ $((p,q) \in B_n)$に関する因果性条件は、滞在時間を$N_L = N_S$とすると、滞在時間N_Lが消去されて、次式のようになる。

$$\begin{aligned}(p+1) * N_S \ \geq\ & (Q-q+1) * N_u + N_f(r{:}|B_n|) + F(A_{n-1}) \\ & + \cdots + F(A_0) + F(B_0) + N_{out} + 1\end{aligned} \tag{4. 20}$$

ただし、$(p.q) \in B_n$はB_nのr番目の要素である。

【最適化問題 4.1】
入力： 係数の添字リスト A，Bと DSP の個数 L_{DSP}。
目的： 条件 4.1（連続性）を満たす A，Bの分割方法 $\{A_n,\ B_n \mid N \geq n \geq 0\}$ において、式 (4.9)，(4.20) の因果性条件、式 (4.19) のサンプリング周期の下限値、式 (4.18) の DSP 個数条件を、すべて満たす最短のサンプリング周期 N_S^*と、それを与える分割方法 $\{A_n^*,\ B_n^* \mid N^* \geq n \geq 0\}$ を求める。 ■

十分に大きなサンプリング周期 N_Sのもとで、式 (4.9)，(4.20) の因果性条件、式 (4.19) の下限値、式 (4.18) の DSP 個数条件を表す各不等式は、常に成り立つ。従って、上記の最適化問題 4.1 に対する許容解は必ず存在する。

4.5.2 サンプリング周期の最適化アルゴリズム

サンプリング周期の最短化を目的とした最適化問題 4.1 に対して、分枝限定法に基づく最適化アルゴリズムを提案する。分枝限定法 [14] とは、3 章でも紹介したように、いわば要領の良い列挙法であり、難解な最適化問題を、幾つかの小規模な部分問題に分割して解くものである。従って、分枝限定法は、原問題の部分問題を次々に生成する分枝操作と、不要な部分問題を終端する限定操作から構成される。

▼ 部分問題

最適化問題 4.1 の部分問題では、前節で示した手順 4.1 に基づき、DSP により最後に処理されるサブリストから、B_0，A_0，B_1，...，A_Nの順番に、各サブリストの適切な要素を考える。従って、部分問題 e は、サブリスト B_mの要素を考える式 (4.21) の部分問題 e_bと、サブリスト A_mの要素を考える式 (4.22) の部分問題 e_aに大別される。

$$e_b = \{B_m,\ A_{m-1},\ B_{m-1},\ \ldots,\ A_0,\ B_0\} \tag{4.21}$$

$$\left[\begin{array}{lcl} A & = & A^c \cup A_{m-1} \cup \ldots \cup A_1 \cup A_0 \\ B & = & B^c \cup B_m \cup B_{m-1} \cup \ldots \cup B_1 \cup B_0 \end{array}\right.$$

$$e_a = \{A_m,\ B_m,\ A_{m-1},\ B_{m-1},\ \ldots,\ A_0,\ B_0\} \tag{4.22}$$

$$\left[\begin{array}{lcl} A & = & A^c \cup A_m \cup A_{m-1} \cup \ldots \cup A_1 \cup A_0 \\ B & = & B^c \cup B_m \cup B_{m-1} \cup \ldots \cup B_1 \cup B_0 \end{array}\right.$$

ただし、A^c，B^cは添字リスト A，Bの未分割な要素の集合である。

式 (4.21)，(4.22) に示した各部分問題において、部分集合 A^c，B^cの要素は、既に分割されたサブリスト A_n，B_n $(m \geq n \geq 0)$ の要素よりも辞書式順序で大きい。また、各サブリスト A_n，B_nは、条件 4.1（連続性）を満たすものとする。ここで、ある部分問題 $e = \{A_n,\ B_n \mid m \geq n \geq 0\}$ に対して，$A^c = B^c = \emptyset$ が成り立つとき，その部分問題 e

は、明らかに最適化問題 4.1 の許容解 x である。

▼ 分枝操作

前述のように、サブリストは B_0, A_0, B_1, ..., A_Nの順番に決定される。従って、部分問題 e_bからは新たな e_aを派生させ、e_aからは e_bを派生させる。

初めに、式 (4.21) の部分問題 e_bを継承して、新たな部分問題 e_aを生成する方法について説明する。式 (4.21) の部分問題 e_bにおいて、サブリスト B_m, A_{m-1}, ..., A_0, B_0の要素は既に確定している。さらに、最も順位が低い A^cの要素を $p_1 \in A^c$、この要素に連続する A^cの要素を p_2, p_3, ..., p_dとするとき、部分問題 e_bからは、次の式 (4.23) に示す $(d+1)$ 個の新たな部分問題 $e_a = \{A_m, B_m, A_{m-1}, \ldots, B_0\}$ を派生させる。

$$\left[\begin{array}{lcl} e_a^0 & = & \{\ \emptyset,\ B_m,\ A_{m-1},\ \ldots,\ B_0\} \\ e_a^1 & = & \{\{\ p_1\ \},\ B_m,\ A_{m-1},\ \ldots,\ B_0\} \\ e_a^2 & = & \{\{p_2,\ p_1\},\ B_m,\ A_{m-1},\ \ldots,\ B_0\} \\ e_a^3 & = & \{\{p_3,\ p_2,\ p_1\},\ B_m,\ A_{m-1},\ \ldots,\ B_0\} \\ & \vdots & \\ e_a^d & = & \{\{p_d,\ \ldots,\ p_3,\ p_2,\ p_1\},\ B_m,\ A_{m-1},\ \ldots,\ B_0\} \end{array}\right. \tag{4.23}$$

ただし、A_mの要素 $p_d > \cdots > p_3 > p_2 > p_1$は連続である。

部分問題 e_bから派生する部分問題の数は、$p_1 \in A^c$に連続する A^cの要素の数 d に依存するが、$A^c = \emptyset$ ならば少なくとも e_a^1は生成される。また、e_a^0は $B_m \neq \emptyset$ である場合のみ生成して、空のサブリストが並ぶような無意味な分割方法は除くものとする。

次に、式 (4.22) の部分問題 e_aを継承して、新たな部分問題 e_bを生成する方法について説明する。最も順位が低い B^cの要素を $(p_1, q_1) \in B^c$、この要素に連続する B^cの要素の数を、(p_1, q_1) を含めて d とする。このとき、部分問題 e_aからは、次の式 (4.24) に示す $(d+1)$ 個の新たな部分問題 $e_b = \{B_{m+1}, A_m, \ldots, B_0\}$ を派生させる。

$$\left[\begin{array}{lcl} e_b^0 & = & \{B_{m+1}^0,\ A_m,\ B_m,\ \ldots,\ B_0\} \\ e_b^1 & = & \{B_{m+1}^1,\ A_m,\ B_m,\ \ldots,\ B_0\} \\ e_b^2 & = & \{B_{m+1}^2,\ A_m,\ B_m,\ \ldots,\ B_0\} \\ e_b^3 & = & \{B_{m+1}^3,\ A_m,\ B_m,\ \ldots,\ B_0\} \\ & \vdots & \\ e_b^d & = & \{B_{m+1}^d,\ A_m,\ B_m,\ \ldots,\ B_0\} \end{array}\right. \tag{4.24}$$

ただし、$B_{m+1}^0 = \emptyset$, $B_{m+1}^1 = \{\ (p_1, q_1)\ \}$, $B_{m+1}^d = \{(p_d, q_d),\ \ldots,\ (p_1, q_1)\}$ である。

▼ 限定操作

初めに、式 (4.22) の部分問題 e_aに対して，最適化問題 4.1 の制約条件、すなわち、因果性条件、サンプリング周期の下限値、DSP 個数条件を、すべて満たす最短のサンプリング周期 $N_S(e_a)$ を求める方法を示す。ただし、部分問題 e_aにおいて、先頭の A_m以外の

サブリスト B_m，A_n，B_n $(m-1 \geq n \geq 0)$ に含まれるすべての要素に対して、これらの制約条件を満たす最短のサンプリング周期$\hat{N}_S$は、すでに得られているとする。

まず、部分問題 e_aのサブリスト A_mの要素に対して、式 (4.9) の因果性条件を満たす最短のサンプリング周期 $N_{S1}(e_a)$ は、次式のように求められる。

$$\left[\begin{array}{rcl} N_S^+(p) & := & ceil\left(\dfrac{N_f(r;|A_m|)+\cdots+F(B_0)+N_{out}+1}{p}\right) \\ N_{S1}(e_a) & := & \max\{\ N_S^+(p) \mid p \in A_m\} \end{array}\right. \tag{4.25}$$

ただし、$p \in A_m$は A_mの r番目の要素である。また、関数 $ceil(R)$ は実数 R以上で最小の整数を表すものとする。

また、部分問題 e_aに対して、式 (4.18) の DSP 個数条件を満たす最短のサンプリング周期 $N_{S2}(e_a)$ は、サブリスト A_n，B_n $(m \geq n \geq 0)$ から次式のように求められる。

$$N_{S2}(e_a) := ceil\left(\frac{N_{int}+F(A_m)+\cdots+F(B_0)+N_{out}+N_{bak}}{L_{DSP}}\right) \tag{4.26}$$

さらに、式 (4.19) に示したサンプリング周期 N_Sの下限値は、あとで述べる最適化アルゴリズムにおいて、既知のサンプリング周期$\hat{N}_S$の初期値として考慮される。従って、部分問題 e_aに対して，最適化問題 4.1 の制約条件をすべて満たす最短のサンプリング周期 $N_S(e_a)$ は、式 (4.25)，(4.26) から次式のように求められる。

$$N_S(e_a) := \max\{\ N_{S1}(e_a),\ N_{S2}(e_a),\ \hat{N}_S\ \} \tag{4.27}$$

次に、式 (4.21) の部分問題 e_bに対して，最適化問題 4.1 の制約条件をすべて満たす最短のサンプリング周期 $N_S(e_b)$ を求める方法を示す。ただし、部分問題 e_bにおいても、先頭の B_m以外のサブリスト A_n，B_n $(m-1 \geq n \geq 0)$ の要素に対して、これらの制約条件を満たす最短のサンプリング周期$\hat{N}_S$は、すでに得られているとする。

まず、部分問題 e_bのサブリスト B_mの要素に対して、式 (4.20) の因果性条件を満たす最短のサンプリング周期 $N_{S1}(e_b)$ は、次式のように求められる。

$$\left[\begin{array}{rcl} N_S^+((p,q)) & := & ceil\left(\dfrac{(Q-q+1)*N_u+N_f(r;|B_m|)\cdots+F(B_0)+N_{out}+1}{p+1}\right) \\ N_{S1}(e_b) & := & \max\{\ N_S^+((p,q)) \mid (p,q) \in B_m\} \end{array}\right. \tag{4.28}$$

ただし、$(p,q) \in B_m$は B_mの r番目の要素である。

また、部分問題 e_bに対して、式 (4.18) の DSP 個数条件を満たす最短のサンプリング周期 $N_{S2}(e_b)$ は、次式のように求められる。

$$N_{S2}(e_b) := ceil\left(\frac{N_{int}+F(B_m)+\cdots+F(B_0)+N_{out}+N_{bak}}{L_{DSP}}\right) \tag{4.29}$$

ここで、部分問題 e_bに対して，最適化問題 4.1 の制約条件をすべて満たす最短のサンプリング周期 $N_S(e_b)$ は、式 (4.28)，(4.29) から次式のように求められる。

$$N_S(e_b) := \max\{\ N_{S1}(e_b),\ N_{S2}(e_b),\ \hat{N}_S\ \} \tag{4.30}$$

提案する分枝限定法において、探索過程のある時点で得られている最良の許容解（暫定解）を $x1$、それに対するサンプリング周期 $N_S(x1)$、すなわち、暫定値を Z（$Z = N_S(x1)$）とする。このとき、ある部分問題 e に対して $N_S(e)$ を求めて、式 (4.31) の関係が成り立てば、この部分問題 e から新たな部分問題を派生させる。また、式 (4.31) の関係が成立しなければ、部分問題 e を終端して以後の考察から除くものとする。

$$N_S(e) < Z \tag{4.31}$$

ところで、サブリスト A_m，B_mに含まれる要素が多い場合、式 (4.25) あるいは式 (4.28) に基づいて、因果性条件を満たす最短のサンプリング周期 $N_{S1}(e)$ を求めることは、計算時間において大きな負担である。しかし、以下に示す定理 4.1 を利用すると、幾つかの要素については、因果性条件の判定が不要となる。

【定理 4.1】

式 (4.18) の DSP 個数条件が、式 (4.19) に示した下限値よりも大きなサンプリング周期 N_Sで成り立つならば、要素$\hat{p} \in A_n$，$(\hat{p}, q) \in B_n$ $(P \geq \hat{p} \geq L_{DSP})$ は、式 (4.9)，式 (4.20) に示した因果性条件を必ず満たす。 ■

[証明]

まず初めに、$\hat{p} \in A_n$ $(\hat{p} \geq L_{DSP})$ が、式 (4.9) の因果性条件を満たすことを証明する。式 (4.5)，式 (4.8) に示した $f(r;R)$，$N_f(r;R)$ の定義と、$\Delta \leq \alpha(R)$ の関係より、

$$F(A_n) \geq N_f(r;|A_n|) = F(A_n) - f(r;|A_n|) + \Delta \tag{4.32}$$

ただし、$\hat{p} \in A_n$ $(\hat{p} \geq L_{DSP})$ は A_nの r番目の要素である。

式 (4.32) を式 (4.18) の DSP 個数条件の右辺に代入すると、

$$\begin{aligned} N_S * L_{DSP} &\geq N_{int} + F(A_N) + \cdots + F(B_0) + N_{out} + N_{bak} \\ &\geq F(A_n) + \cdots + F(B_0) + N_{out} + N_{bak} \\ &\geq N_f(r;|A_n|) + \cdots + F(B_0) + N_{out} + N_{bak} \end{aligned} \tag{4. 33}$$

さらに、$\hat{p} \geq L_{DSP}$であるので、式 (4.9) の因果性条件が成り立つ。

$$\begin{aligned} \hat{p} * N_S &\geq N_S * L_{DSP} \\ &\geq N_{int} + N_f(r;|A_n|) + \cdots + F(B_0) + N_{out} + N_{bak} \end{aligned} \tag{4. 34}$$

次に、$(\hat{p}, q) \in B_n$ $(\hat{p} \geq L_{DSP})$ が、式 (4.20) に示した因果性条件を満たすことを証明

する。まず、式 (4.32) と同様にして、次式が導かれる。

$$F(B_n) \geq N_f(r;|B_n|) = F(B_n) - f(r;|B_n|) + \Delta \tag{4.35}$$

ただし、$(\hat{p},q) \in B_n$ $(\hat{p} \geq L_{DSP})$ は B_n の r 番目の要素である。

式 (4.35) を式 (4.18) の DSP 個数条件の右辺に代入すると、

$$\begin{aligned} N_S * L_{DSP} &\geq N_{int} + F(A_N) + \cdots + F(B_0) + N_{out} + N_{bak} \\ &\geq F(B_n) + \cdots + F(B_0) + N_{out} + N_{bak} \\ &\geq N_f(r;|B_n|) + \cdots + F(B_0) + N_{out} + N_{bak} \end{aligned} \tag{4.36}$$

式 (4.36) の両辺に $(Q-q+1)*N_u$ を加えると、

$$\begin{aligned} &N_S * L_{DSP} + (Q-q+1) * N_u \\ &\geq (Q-q+1) * N_u + N_f(r;|B_n|) + \cdots + F(B_0) + N_{out} + N_{bak} \end{aligned} \tag{4.37}$$

また、式 (4.19) に示した制約条件より、次式の関係が成り立つ。

$$N_S \geq Q * N_u \geq (Q-q+1) * N_u, \qquad (Q \geq q \geq 1) \tag{4.38}$$

ここで、式 (4.37), (4.38), $\hat{p} \geq L_{DSP}$ の関係から、

$$\begin{aligned} (\hat{p}+1) * N_S &\geq N_S * L_{DSP} + (Q-q+1) * N_u \\ &\geq (Q-q+1) * N_u + N_f(r;|B_n|) + \cdots + F(B_1) \\ &\quad + F(A_0) + F(B_0) + N_{out} + N_{bak} \end{aligned} \tag{4.39}$$

□

定理 4.2 の結果は、ディジタル制御器において、出力 $y[k]$ と $y[k-L_{DSP}]$ が同じ DSP により計算されることからも容易に推察できる。すなわち、$y[k]$ の計算に用いられるデータ $y[k-\hat{p}]$, $u_q[k-\hat{p}]$ $(P \geq \hat{p} \geq L_{DSP})$ は、直前の出力 $y[k-L_{DSP}]$ の計算でも使われている。従って、$y[k-L_{DSP}]$ の計算が終了していれば、これらのデータは $y[k]$ の計算を開始する以前に RAM 内に存在し、因果性条件を考える必要はない。

定理 4.2 より、式 (4.25) において、要素 $\hat{p} \in A_m$ $(P \geq \hat{p} \geq L_{DSP})$ に対して $N_S^+(\hat{p})$ を求める必要はない。また、A_m に含まれるすべての要素が、この条件を満たすならば、式 (4.27) において、因果性条件に基づく $N_{S1}(e_a)$ を考慮する必要もない。このことは、式 (4.28) における $N_S^+((\hat{p},q))$ $(P \geq \hat{p} \geq L_{DSP})$ の評価でも同様である。

▼ 選択操作

分枝限定法に基づく探索の過程において、サンプリング周期 N_S が評価されていない活性な部分問題の中から、いかに効率的に適切な部分問題を選ぶかが生成される部分問

題の総数、すなわち、アルゴリズムの計算時間を左右する。そこで、提案する最適化アルゴリズムでは、前述の積和演算の実行時間に関する仮定4.1から導かれる次の定理4.2をヒューリスティクスとして、部分問題に対する評価順序を決める。

【定理4.2】
仮定4.1のもとで、次の関係が成り立つ。

$$f(R_1-R_2;R_1-R_2)+f(R_2;R_2)\geq f(R_1;R_1),\quad 0\leq R_2\leq ceil\left(\frac{R_1}{2}\right). \tag{4.40}$$

ただし、$ceil\,(R_1/2)$は実数$R_1/2$以上の最小整数を表す。 ■

[証明]
式(4.5)に示した積和演算の処理時間を表す関数$f(r;R)$の定義より、

$$\begin{aligned} & f(R_1-R_2;R_1-R_2)+f(R_2;R_2)-f(R_1;R_1)\\ = \;& \alpha(R_1-R_2)*R_1+\beta(R_1-R_2)\\ & +(\alpha(R_2)-\alpha(R_1-R_2))*R_2+\beta(R_2)\\ & -(\alpha(R_1)*R_1+\beta(R_1)) \end{aligned} \tag{4.41}$$

ここで、$(R_1-R_2)\geq R_2$であるので、仮定4.1で述べた$\alpha(R)$の性質より、

$$\alpha(R_2)\geq\alpha(R_1-R_2) \tag{4.42}$$

さらに、仮定4.1より$(\alpha(R),\beta(R))$は、DSPがR回の積和演算を最短時間で実行するための計算方法を示していることから、

$$\alpha(R_1-R_2)*R_1+\beta(R_1-R_2)\geq\alpha(R_1)*R_1+\beta(R_1) \tag{4.43}$$

式(4.42), (4.43)を式(4.41)に代入すると、

$$f(R_1-R_2;R_1-R_2)+f(R_2;R_2)-f(R_1;R_1)\geq 0 \tag{4.44}$$

□

上記の定理4.2は、DSPによる積和演算は何度かに分割して行うよりも、まとめて連続的に実行する方が、全体の処理時間を短縮できることを意味している。DSPが連続的に実行する積和演算の回数を増やすことは、できるだけ多くの要素を1つのサブリストA_n, B_nに詰め込むことに相当する。従って、提案する部分問題の選択方法としては、式(4.23), (4.24)に示した$(d+1)$個の部分問題を、それぞれサブリストA_m, B_{m+1}に含まれる要素が多い順番にe^d, ..., e^1, e^0と選ぶものとする。

▼ 最適化アルゴリズムの構成

提案する最適化アルゴリズムでは、探索方法として深さ優先探索法を採用する。そこで、サンプリング周期 $N_S(e)$ の評価が済んでいない活性な部分問題 e を保存するために、スタック Sを用いる。また、$N_S(e)$ の評価を式 (4.27), (4.30) に基づき高速に行うために、サンプリング周期$\hat{N}_S$を部分問題と共にスタック Sに積むものとする。

最適化問題 4.1 の制約条件である式 (4.19) に示した N_Sの下限値を考慮すると、部分問題 e に対する $N_S(e)$ は、次式の関係を満たす必要かある。

$$N_S(e) \geq \hat{N}_S \geq Q * N_u \tag{4.45}$$

そこで、提案する最適化アルゴリズムでは、サンプリング周期の下限値 $(Q * N_u)$ を$\hat{N}_S$ の初期値とする。これにより、探索範囲をある程度限定できる。

最後に、スタック Sが空になると最適化アルゴリズムの計算は終了して、その時点における暫定解 $x1$ と暫定値 $Z = N_S(x1)$ が、最適化問題 4.1 の最適解 $x1^*$ と実現可能な最短サンプリング周期 N_S^*を与える。

【最適化アルゴリズム 4.1】

begin

1 : $Z := \infty$;　$\hat{N}_S := Q * N_u$;　　// 初期値
2 : $B_0 := \emptyset$;　$e_b := \{\ B_0\ \}$;
3 : $push((e, \hat{N}_S), S)$;
4 : *while* (*not empty*(S)) {
5 :　　$(e, \hat{N}_S) := pop(S)$;
6 :　　*compute* $N_S(e)$ *with* $\hat{N}_S$;
7 :　　*if* $(N_S(e) < Z)$ {
8 :　　*if* $(A^c = B^c = \emptyset)$ { $x1 := e$;　$Z := N_S(x1)$; }
9 :　　*else* {　　// 分枝操作
10 :　　*make new* e^0, e^1, ..., e^d *from* e;　$\hat{N}_S := N_S(e)$;
11 :　　$push((e^0, \hat{N}_S), S)$;　$push((e^1, \hat{N}_S), S)$;　$\cdots$;　$push((e^d, \hat{N}_S), S)$;
12 :　　}　　// 定理 4.2 より e^dを最優先
13 :　　}　　// 部分問題 e の終端
14 : }
15 : *output* $x1$ *and* Z;　　// 最適解の出力（$N_S^* = Z$）

end.

ただし、$\emptyset$ は空のリスト、∞ は十分に大きな整数を表す。　■

4.5.3 滞在時間の最適化問題

最適化問題 4.1 を解いて得られた最短サンプリング周期 N_S^*のもとで、滞在時間 N_Lを最短とすることを目的とした最適化問題 4.2 について考える。

【最適化問題 4.2】
入力： 係数の添字リスト A, Bと DSP の個数 L_{DSP}、最短サンプリング周期 N_S^*。
目的： 条件 4.1（連続性）を満たす A, Bの分割方法 $\{A_n,\ B_n \mid N \geq n \geq 0\}$ において、最短サンプリング周期 N_S^*のもとで、式 (4.9), (4.14) の因果性条件、式 (4.15) の滞在時間条件、式 (4.18) の DSP 個数条件を、すべて満たす最短の滞在時間 N_L^*と、それを与える分割方法 $\{A_n^*,\ B_n^* \mid N^* \geq n \geq 0\}$ を求める。 ■

最適化問題 4.1 の最適解は、$N_L = N_S^*$とすれば、最適化問題 4.2 に対する自明の許容解となる。従って、最適化問題 4.2 の許容解は必ず存在する。

4.5.4 滞在時間の最適化アルゴリズム

最適化問題 4.2 に対しても、最適化問題 4.1 と同様にして、分枝限定法に基づく最適化アルゴリズムを構成する。

まず、最適化問題 4.2 に対する部分問題 e は、さきに式 (4.21) に示した部分問題 e_bと、式 (4.22) に示した部分問題 e_aである。また、これらの部分問題を生成するための分枝操作についても、最適化問題 4.1 と同じである。しかし、各部分問題の評価方法は、若干異なり複雑である。最適化問題 4.2 の制約条件において、目的関数である滞在時間 N_Lを含んでいるものは、式 (4.18) の入力データに関する因果性条件と、式 (4.19) の滞在時間条件のみであることに注意されたい。

▼ 限定操作

初めに、式 (4.22) の部分問題 e_aについては、式 (4.9) の出力データ $y[k-p]$ $(p \in A_m)$ に関する因果性条件が滞在時間 N_Lを含まないことから、そのサンプリング周期 $N_S(e_a)$ が与えられた N_S^*よりも大きいか否かのみを判定する。まず、式 (4.9) の因果性条件と、DSP 個数条件を満たす最短のサンプリング周期 $N_{S1}(e_a)$, $N_{S2}(e_a)$ を、それぞれ式 (4.25) と式 (4.26) から求める。さらに、次の式 (4.46) から部分問題 e_aに対する $N_S(e_a)$ を評価する。ここで、式 (4.27) の$\hat{N}_S$に代わり、N_S^*を用いることに注意されたい。

$$N_S(e_a) := \max\{\ N_{S1}(e_a),\ N_{S2}(e_a),\ N_S^*\ \} \tag{4.46}$$

分枝限定法の探索過程において、ある部分問題 e_aに対して $N_S(e_a)$ を求めて、式 (4.47) の関係が成り立てば、e_aから新たな部分問題 e_bを派生させる。また、式 (4.47) の関係が成立しなければ、e_aを終端して以後の考察から除くものとする。

$$N_S(e_a) = N_S^* \tag{4.47}$$

一方、式 (4.21) の部分問題 e_bについては、式 (4.14) の入力データ $u_q[k-p]$ $((p,q) \in B_m)$ に関する因果性条件が N_Sと N_Lを含むため、そのサンプリング周期 $N_S(e_b)$ と滞在時間 $N_L(e_b)$ を求める必要がある。まず、DSP 個数条件を満たす最短のサンプリング周期 $N_{S2}(e_b)$ を式 (4.26) から求めて、次式から部分問題 e_bに対する $N_S(e_b)$ を評価する。

$$N_S(e_b) := \max\{\ N_{S2}(e_b),\ N_S^*\ \} \tag{4.48}$$

ここで、次の関係が成立しなければ、部分問題 e_bを終端する。

$$N_S(e_b) = N_S^* \tag{4.49}$$

次に、部分問題 e_bのサブリスト B_mの要素に対して、式 (4.14) の因果性条件を満たす最短の滞在時間 $N_{L1}(e_b)$ は、$N_S = N_S^*$として次の式 (4.50) から求められる。

$$\left[\begin{array}{rcl} N_L^+((p,q)) & := & ((Q-q+1) * N_u + N_f(r;|B_m|) + \cdots + F(B_1) \\ & & \quad + F(A_0) + F(B_0) + N_{out} + 1) - p * N_S^* \\ & & \\ N_{L1}(e_b) & := & \max\{\ N_L^+((p,q)) \mid (p,q) \in B_m\ \} \end{array}\right. \tag{4.50}$$

ただし、$(p,q) \in B_m$は B_mの r番目の要素である。

ここで、部分問題 e_bを派生させた元の部分問題に対する最短の滞在時間を$\hat{N}_L$とするとき、e_bに対する最短の滞在時間 $N_L(e_b)$ は、次の式 (4.51) から求められる。最適化問題 4.2 の制約条件である式 (4.15) に示した滞在時間 N_Lの下限値は、あとで述べる最適化アルゴリズムにおいて、滞在時間$\hat{N}_L$の初期値として考慮される。

$$N_L(e_b) := \max\{\ N_{L1}(e_b),\ \hat{N}_L\ \} \tag{4.51}$$

提案する分枝限定法において、探索過程のある時点で得られている暫定解を $x2$、それに対する暫定値を Z $(Z = N_L(x2))$ とする。このとき、ある部分問題 e_bに対して $N_L(e_b)$ を求めて、式 (4.52) の関係が成り立てば、e_bから新たな部分問題を派生させる。また、式 (4.52) の関係が成立しなければ、e_bを終端して以後の考察から除くものとする。

$$N_L(e_b) < Z \tag{4.52}$$

▼ 最適化アルゴリズムの構成

提案する最適化アルゴリズムでは、探索方法として深さ優先探索法を採用する。そこで、サンプリング周期 $N_S(e)$ や滞在時間 $N_L(e)$ の評価が済んでいない活性な部分問題 e

を保存するために、スタック Sを用いる。また、$N_L(e)$ の評価を式 (4.51) に基づき高速に行うために、滞在時間$\hat{N}_L$を部分問題と共にスタック Sに積むものとする。

最適化問題 4.2 の制約条件である式 (4.15) に示した N_Lの下限値を考慮すると、部分問題 e に対する $N_L(e)$ は、次式の関係を満たす必要かある。

$$N_S^* \geq N_L(e) \geq \hat{N}_L \geq Q * N_u \tag{4.53}$$

そこで、提案する最適化アルゴリズムでは、滞在時間の下限値 $(Q * N_u)$ を$\hat{N}_L$の初期値とする。さらに、暫定解 $x2$ の初期値を最適化問題 4.1 の最適解 $x1^*$、暫定値 Zの初期値を最短サンプリング周期 N_S^*とする。これにより、探索範囲をかなり限定できる。

最後に、スタック Sが空になると最適化アルゴリズムの計算は終了して、その時点における暫定解 $x2$ と暫定値 $Z = N_L(x2)$ が、最適化問題 4.2 の最適解 $x2^*$と実現可能な最短の滞在時間 N_L^*を与える。

【最適化アルゴリズム 4.2】

begin

1 : $x2 := x1^*$; $Z := N_S^*$; $\hat{N}_L := Q * N_u$; // 初期値
2 : $B_0 := \emptyset$; $e_b := \{ B_0 \}$;
3 : $push((e_b, \hat{N}_L), S)$;
4 : $while\ (not\ empty(S))\ \{$
5 : $(e, \hat{N}_L) := pop(S)$;
6 : $compute\ N_S(e)$;
7 : $if\ (N_S(e) = N_S^*)\ \{$
8 : $compute\ N_L(e)\ with\ \hat{N}_L$; // $(N_L(e_a) := \hat{N}_L;)$
9 : $if\ (N_L(e) < Z)\ \{$
10 : $if\ (A^c = B^c = \emptyset)\ \{\ x2 := e;\ Z := N_L(x2);\ \}$
11 : $else\ \{$ // 分枝操作
12 : $make\ new\ e^0,\ e^1,\ \ldots,\ e^d\ from\ e;\ \hat{N}_L := N_L(e);$
13 : $push((e^0, \hat{N}_L), S);\ \cdots;\ push((e^d, \hat{N}_L), S);$
14 : $\}$ // 定理 4.2 より e^dを最優先
15 : $\}$ // 部分問題 e の終端
16 : $\}$ // 部分問題 e の終端
17 : $\}$
18 : $output\ x2\ and\ Z$; // 最適解の出力 $(N_L^* = Z)$

end. ■

4.6 適用例による評価

提案した最適化アルゴリズムを、具体的な例題に対して適用する。ディジタル制御器を実現するための DSP に TMS320C25[81] を採用すると、式 (4.5) で定義した積和演算に要する時間を表す関数 $f(r;R)$ は、次のように与えられる。

$$f(r;R) := \alpha(R) * r + \beta(R) \tag{4.54}$$

$$(\alpha(R),\ \beta(R)) = \begin{cases} (0,\ 0), & R = 0. \\ (3,\ 1), & 1 \leq R \leq 3. \\ (2,\ 4), & 4 \leq R. \end{cases}$$

また、DSP が 1 回の乗算に要するサイクル数は$\Delta = 2$ である。さらに、手順 4.1 における各時間定数（サイクル数）を、初期化処理 $N_{int} = 2$、出力処理 $N_{out} = 3$、先頭に戻る処理 $N_{bak} = 2$ とし、各入力 $u_q[k]$ のサンプリングには $N_u = 3$ を要すとする。

▼ 例題 1

最適化問題 4.0 の入力例として、次の式 (4.55) に示す添字リスト A, B を与えて、DSP の個数を $L_{DSP} = 3$ とした。ただし、制御則の入力数は $Q = 1$ である。

$$\begin{bmatrix} A &=& \{\ 5,\ 4,\ 2,\ 1\ \} \\ B &=& \{\ 6,\ 5,\ 4,\ 3,\ 0\ \} \end{bmatrix} \tag{4.55}$$

初めに、式 (4.55) のリスト A, Bを最適化問題 4.1 の入力例として、最適化アルゴリズム 4.1 を適用すると、最適化問題 4.1 に対する最適解が、以下のように求まる。

$$\begin{bmatrix} N_S^* = 13, \quad A_3 = \{\ 5\ \}, \quad A_2 = \{\ 4\ \}, \quad A_1 = \{\ 2\ \}, \quad A_0 = \{\ 1\ \}, \\ B_3 = \{\ 6,\ 5,\ 4,\ 3\ \}, \quad B_2 = \emptyset, \quad B_1 = \{\ 0\ \}, \quad B_0 = \emptyset. \end{bmatrix} \tag{4.56}$$

次に、式 (4.55) のリスト A, Bと $L_{DSP} = 3$、最適化問題 4.1 で得られた最短サンプリング周期 $N_S^* = 13$ を、最適化問題 4.2 の入力例として、最適化アルゴリズム 4.2 を適用すると、最適化問題 4.2 に対する最適解が、以下のように求まる。

$$\begin{bmatrix} N_L^* = 9, \quad A_1 = \{\ 5,\ 4\ \}, \quad A_0 = \{\ 2,\ 1\ \}, \\ B_1 = \{\ 6,\ 5,\ 4,\ 3\ \}, \quad B_0 = \{0\}. \end{bmatrix} \tag{4.57}$$

従って、最適化問題 4.0 に対するパレート最適解は、最適化問題 4.1 を解いて得られた最短サンプリング周期 $N_S^* = 13$ と、最適化問題 4.2 の最適解である最短の滞在時間 $N_L^* = 9$、および、式 (4.57) に示したリスト A, Bの分割方法である。

▼ 例題 2

最適化問題 4.0 の入力例として、次の式 (4.58) に示す添字リスト A, B を与えた。ただし、制御則の次数は $P = 9$、入力数は $Q = 3$、すべての係数は非零である。

$$\left[\begin{array}{lcl} A & = & \{\, p \mid 9 \geq p \geq 1 \,\} \\ B & = & \{\, (p,q) \mid 9 \geq p \geq 0,\ 3 \geq q \geq 1 \,\} \end{array}\right. \tag{4.58}$$

上記の入力例において並列化する DSP の個数 L_{DSP} を変えて、最適化問題 4.0 のパレート最適解を求めた。提案した最適化アルゴリズムにより、各 L_{DSP} に対して得られた最短サンプリング周期 N_S^* と、最短の滞在時間 N_L^* を図 4.11 に示す。

並列化する DSP の個数 L_{DSP} の増加に伴って、最短サンプリング周期 N_S^* は単調に減少しているが、滞在時間 N_L^* は必ずしも減少していない。このようなパレート最適解に対する目的関数値 $(N_S^*,\ N_L^*)$ の特性は、提案した最適化アルゴリズムが、滞在時間 N_L よりもサンプリング周期 N_S を優先して最短化するためと考えられる。

また、DSP の個数 L_{DSP} が異なる各入力例に対して、最適化アルゴリズム 4.1 と最適化アルゴリズム 4.2 が要した各計算時間と、生成された部分問題の総数を表 4.3 に示す。表 4.3 から、各アルゴリズムの計算時間が、部分問題の総数によって決まることが確認できる。ただし、使用した計算機は Sun SPARC Station 20 である。

▼ 例題 3

最適化問題 4.0 の入力例として、次の式 (4.59) に示す添字リスト A, B を与えた。前述の例題 2 と同様に、制御則の次数は $P = 9$、入力数は $Q = 3$ であるが、零係数が含まれており、$b_{pq} = 0\ (9 \geq p \geq 7,\ 3 \geq q \geq 1)$ である。

$$\left[\begin{array}{lcl} A & = & \{\, p \mid 9 \geq p \geq 1 \,\} \\ B & = & \{\, (p,q) \mid 6 \geq p \geq 0,\ 3 \geq q \geq 1 \,\} \end{array}\right. \tag{4.59}$$

上記の入力例において並列化する DSP の個数 L_{DSP} を変えて、最適化問題 4.0 のパレート最適解を求めた。各 L_{DSP} に対して得られた最短サンプリング周期 N_S^* と、最短の滞在時間 N_L^* を図 4.12 に示す。図 4.11 に示した例題 2 の結果と比較すると、制御則に含まれる零係数の計算が除かれるため、同じ個数の DSP を使用した場合に、さらに短い最短サンプリング周期が実現されることが確認できる。

また、DSP の個数 L_{DSP} が異なる各入力例に対して、最適化アルゴリズム 4.1 と最適化アルゴリズム 4.2 が要した各計算時間と、生成された部分問題の総数を表 4.4 に示す。表 4.3 に示した例題 2 に対する計算時間と比較すると、制御則から零係数を除くことは、アルゴリズムの計算時間においても有利であることが分かる。

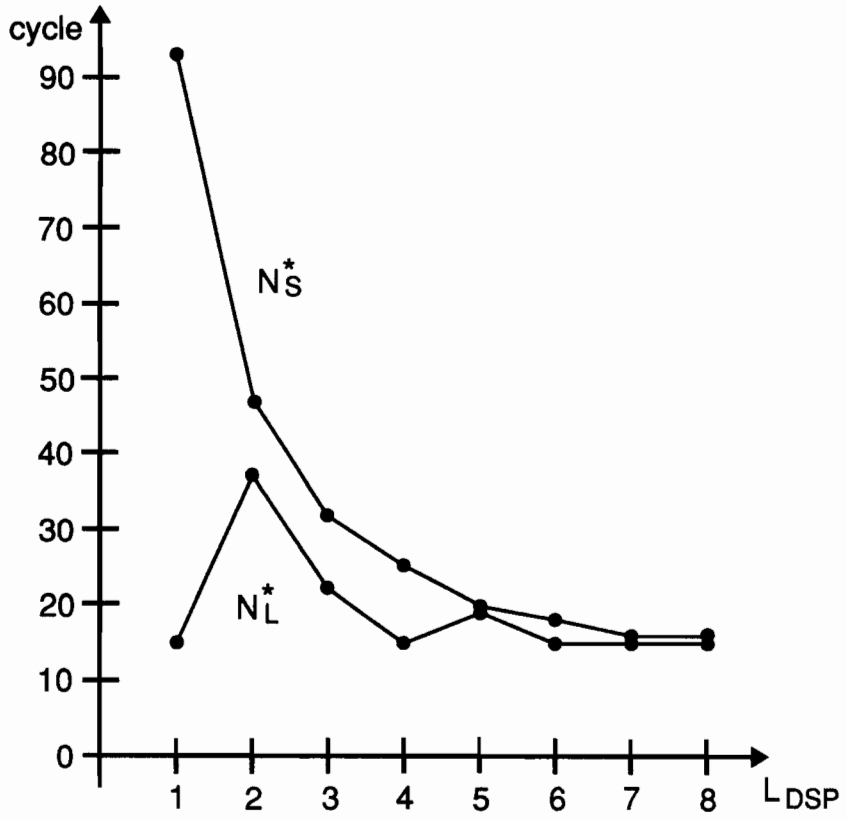


図 4. 11　サンプリング周期と滞在時間

表 4. 3　最適化アルゴリズムの計算時間

	algorithm - 4.1		algorithm - 4.2	
L_{DSP}	sub-problem	time [sec]	sub-problem	time [sec]
1	1019	0.04	1019	0.02
2	26836	0.57	25696	0.50
3	257317	5.05	238190	4.42
4	1151756	22.66	1164325	22.04
5	2036141	41.56	2525552	47.70
6	1160187	23.63	3018413	58.00
7	10349	0.25	1215111	23.72
8	15884	0.40	7351492	159.41

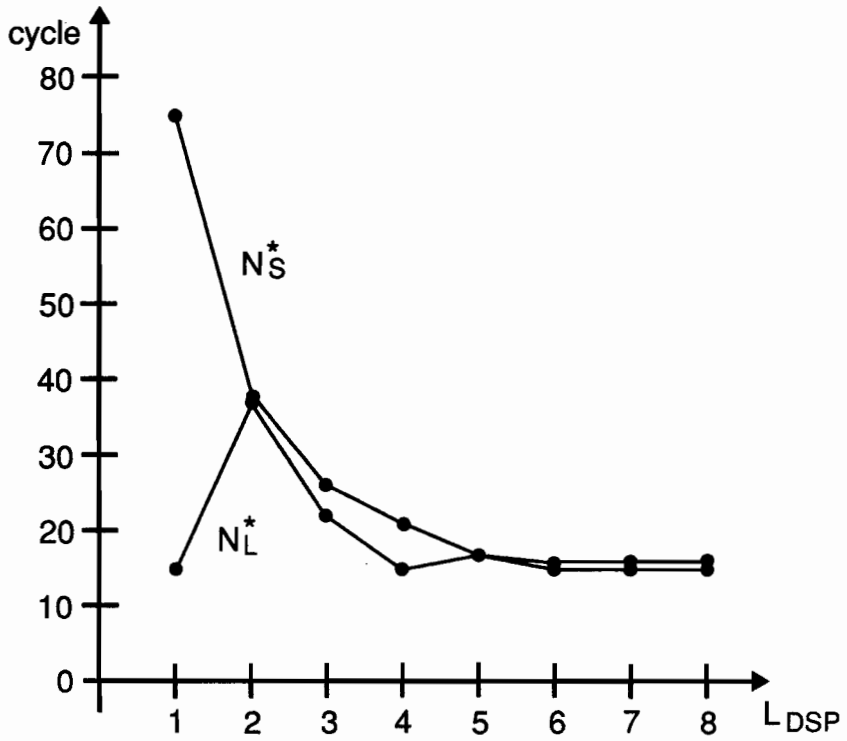


図 4. 12　サンプリング周期と滞在時間

表 4. 4　最適化アルゴリズムの計算時間

	algorithm - 4.1		algorithm - 4.2	
L_{DSP}	sub-problem	time [sec]	sub-problem	time [sec]
1	632	0.02	632	0.04
2	13660	0.30	13033	0.27
3	86750	1.75	70390	1.33
4	220724	4.54	235456	4.50
5	31477	0.67	46823	0.89
6	3443	0.12	104677	2.19
7	4852	0.16	480236	11.51
8	5513	0.20	799956	21.52

4.7 結言

この章では、複数の信号処理用プロセッサ（DSP）を用いた並列処理によるディジタル制御器の実現方法を提案した。ロボット・アームの動的制御のように、フィードバック制御系に組み込まれて使用されるディジタル制御器に対しては、そのサンプリング周期と滞在時間を、共に短縮することが要求される。提案したディジタル制御器は、従来のシストリックアレイによるディジタルフィルタの実現方法とは異なり、制御則の次数などに関係なく、並列化するプロセッサ（DSP）の個数を増やすことによって、サンプリング周期を短縮することが可能である。さらに、滞在時間については、常にサンプリング周期よりも短くすることができる。このほか、提案したディジタル制御器では、サンプリング周期ごとに制御則を計算する DSP を切り換えるため、各 DSP におけるパイプライン処理の機能を有効に活用できる。

また、この章では、与えられた個数の DSP を用いてディジタル制御器を実現する際に、ディジタル制御器の重要な特性であるサンプリング周期と滞在時間を、共に最短とするための多目的スケジューリング問題について検討を行った。

初めに、複数の DSP による並列処理において、満たされる必要がある 3 つの制約条件、すなわち、因果性条件、滞在時間条件、DSP 個数条件を明らかにして、上記の多目的スケジューリング問題を、サンプリング周期と滞在時間の 2 つの目的を持つ、組合せ最適化問題として定式化した。

次に、この組合せ最適化問題に対して、分枝限定法に基づく最適化アルゴリズムを提案した。提案した最適化アルゴリズムは、サンプリング周期のみが最短となる弱パレート最適解の 1 つを求めた後に、滞在時間についても最短化を図り、多目的の最適化問題に対するパレート最適解の 1 つを得るものである。

最後に、幾つかの多目的スケジューリング問題の例題を解くことによって、提案した最適化アルゴリズムの特性を評価した。

第5章

結論

本研究は、ロボット・アームの高速かつ高精度な制御に有効な動的制御に関連して、その実用化の大きな障害となっている制御則の計算時間の短縮を目的とした、複数のプロセッサによる並列処理手法を提案したものである。この章では、本研究により得られた成果を総括して述べる。

ロボットの動的制御に寄与する制御則は、多変数多項式として表される非線形な下位制御則と、伝達関数などにより表される線形な上位制御則に分けられる。そこで、これらの制御則に対して独立に、その特性を考慮した並列処理手法を提案した。

まず、ロボットの運動方程式に基づく非線形な下位制御則（ロボット制御則）は、数式処理システムにより多変数多項式として導出できる。ところが、このようなロボット制御則は、冗長な演算を数多く含むため、そのまま計算しては効率が良くない。そこで、第2章では、多変数多項式から冗長な演算を削除するために、数式の簡単化手法を提案した。この数式の簡単化手法によれば、数式処理システムにより導出したロボット制御則から、含まれる演算回数が最少であるという意味で最適化された計算公式を、機械的に得ることができる。また、従来の高速計算アルゴリズムと比較して、提案した数式の簡単化手法を用いる利点としては、以下のような事項が挙げられる。

(1) 多変数多項式によって表される、任意のロボット制御則に対して適用できる。
(2) ロボットの幾何学的な構造の違いも反映した効率的な計算公式が得られる。

第3章では、多変数多項式として与えられたロボット制御則の計算に対して、MIMD分散メモリ型の並列計算機モデルを用いた並列処理手法を提案した。この並列処理手法は、ロボット制御則の計算を積項単位で複数のプロセッサへ割り当てた後に、さきに述べた数式の簡単化手法を用いて、各プロセッサごと逐次的に処理される計算式に含まれる演算回数を減らすものである。提案した非線形な下位制御則に対する並列処理手法の利点としては、以下のような事項が挙げられる。

(1) 多変数多項式によって表される、任意のロボット制御則に対して適用できる。
(2) ロボット制御則に関係なく、任意の個数のプロセッサを並列化できる。
(3) プロセッサ間の通信が少ないために、プロセッサの結合方法などハードウェア的な違いに影響されず、多くの並列計算機において実現できる。

次に、与えられたプロセッサ数のもとで、並列処理に要する時間を最短とするために、ロボット制御則の簡単化と並列化を共に考慮したスケジューリング問題について考え、これを最適化問題として定式化した。また、この最適化問題が、*NP*困難であることを証明した。さらに、上記の最適化問題に対して、3種類の異なる特徴を持つスケジューリング・アルゴリズムを提案した。すなわち、大規模な問題にも適用可能な近似アルゴリズム、分枝限定法に基づき最適解を厳密に求める最適化アルゴリズム、これら2つのアルゴリズムを組み合わせた準最適化アルゴリズムである。

第4章では、差分方程式として与えられた線形な上位制御則の計算に対して、複数の信号処理用プロセッサ（DSP）を用いた並列処理手法と、ディジタル制御器の構成方法を提案した。ロボットの動的制御のように、ディジタル制御器がフィードバック制御系に組み込まれる場合には、そのサンプリング周期と滞在時間を、共に短縮することが必要となる。提案したディジタル制御器では、サンプリング周期ごとに制御則を計算するDSPを切り換えることで、このような要求を満たしている。ここで、提案した並列処理手法の利点としては、以下のような事項が挙げられる。

(1)　制御則の次数などに関係なく、任意の個数のDSPを並列化できる。
(2)　並列化するDSPの個数に比例して、サンプリング周期を短縮できる。
(3)　滞在時間を、常にサンプリング周期よりも短くできる。
(4)　各DSPにおけるパイプライン処理の機能を有効に活用できる。

次に、与えられた個数のDSPを用いたディジタル制御器において、サンプリング周期と滞在時間を、共に最短とするための多目的スケジューリング問題について考え、これを多目的の最適化問題として定式化した。さらに、この多目的の最適化問題に対して、分枝限定法に基づき、そのパレート最適解を求める最適化アルゴリズムを提案した。

ロボット制御則の計算は、際限なく繰り返し実行される処理である。従って、ロボット制御則の計算時間の短縮に費された時間は、速度の向上が僅かであっても、その計算処理が実行されるたびに戻ってくる。この意味において、2種類の並列処理手法を提案するのみならず、スケジューリング問題について考えることにより、各並列処理手法による計算の高速化の可能性を追求した意義は大きい。

最後に、本研究の成果が、ロボットの動的制御の実用化に貢献すれば幸いである。

謝　辞

本研究を遂行するにあたり、神戸大学工学部 羽根田 博正 教授、並びに、太田 有三 教授には、終始暖かい御指導、御教示を賜りました。ここに、心より感謝の意を表します。

また、本論文の執筆に際して、多くの御助言を賜りました神戸大学工学部 高森 年 教授に、厚く御礼申し上げます。

さらに、共に研究を行った神戸大学工学部電気電子工学科・羽根田研究室の卒業生、牧 秀隆、森 崇、藤原 正和、神吉 良英、天野 昌幸、津田 政彦、浪越 孝宏、福居 毅至の各氏に、深く感謝いたします。

最後に、日頃の研究を通じて、様々な御支援を頂きました山中 和彦 技官をはじめ、羽根田研究室の諸氏に感謝いたします。

付録A　ロボット制御系のリアプノフ関数

リアプノフ関数によって安定性が保証されるロボット制御系の構成方法を紹介する。このロボット制御系は、Koditschek[34] により提案されたものである。

N関節（N自由度）のロボット・アームの動的モデルを、次の式 (A.1) に示すようなラグランジュ運動方程式によって表す。

$$\mathbf{f}(t) = M(\mathbf{q}(t)) * \ddot{\mathbf{q}}(t) + H(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{q}}(t) + g(\mathbf{q}(t)) \tag{A.1}$$

ただし、$\mathbf{q}(t) \in \mathbf{R}^N$は関節変位を表すベクトル、$\mathbf{f}(t) \in \mathbf{R}^N$は関節に加える駆動トルクまたは力を表すベクトルである。式 (2.2) 右辺の第 1 項は慣性力を、第 2 項はコリオリ力と遠心力を、第 3 項は重力の影響を表す。

式 (A.1) に示したロボットの運動方程式を、次の式 (A.2) のように変形する。

$$\mathbf{f}(t) = M(\mathbf{q}(t)) * \ddot{\mathbf{q}}(t) + \frac{1}{2} * \dot{M}(\mathbf{q}(t)) * \dot{\mathbf{q}}(t) + S(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{q}}(t) + g(\mathbf{q}(t)) \tag{A.2}$$

ただし、

$$S(\dot{\mathbf{q}}(t), \mathbf{q}(t)) := H(\dot{\mathbf{q}}(t), \mathbf{q}(t)) - \frac{1}{2} * \dot{M}(\mathbf{q}(t)) * \dot{\mathbf{q}}(t)$$

式 (A.2) の運動方程式は、以下のような性質を持つことが知られている [4]。

性質 1 ：$M(\mathbf{q}(t)) \in \mathbf{R}^{N \times N}$は、有界かつ正定な対称行列である。

$$\forall \mathbf{z} \in \mathbf{R}^N,\ \mathbf{z} \neq 0 : \quad \mathbf{z}^T * M(\mathbf{q}(t)) * \mathbf{z} > 0 \tag{A.3}$$

$$M(\mathbf{q}(t)) = M(\mathbf{q}(t))^T \tag{A.4}$$

性質 2 ：$S(\dot{\mathbf{q}}(t), \mathbf{q}(t)) \in \mathbf{R}^{N \times N}$は、歪み対称行列である。

$$\forall \mathbf{z} \in \mathbf{R}^N : \quad \mathbf{z}^T * S(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \mathbf{z} = 0 \tag{A.5}$$

時間 t における目標軌道 $\mathbf{p}$ と関節変位 $\mathbf{q}$ の軌道誤差を $\mathbf{e}(t)$ とする。

$$\mathbf{e}(t) := \mathbf{p}(t) - \mathbf{q}(t) \tag{A.6}$$

ここで、式 (A.2) に示したロボットに対して、次の式 (A.7) に示すようなスカラ関数をリアプノフ関数の候補として考える。

$$V(\dot{\mathbf{e}}(t), \mathbf{e}(t)) := \frac{1}{2} * \dot{\mathbf{e}}(t)^T * M(\mathbf{q}(t)) * \dot{\mathbf{e}}(t) + \frac{1}{2} * \mathbf{e}(t)^T * K_p * \mathbf{e}(t) \tag{A.7}$$

ただし、$K_p \in \mathbf{R}^{N \times N}$は、任意の正定行列である。

式 (A.7) のスカラ関数 $V(\dot{\mathbf{e}}(t), \mathbf{e}(t))$ が、リアプノフ関数であることを示す。まず、性質 1 の式 (A.3) より、状態変数ベクトル $\mathbf{x} = [\dot{\mathbf{e}},\ \mathbf{e}]^T$に関して、スカラ関数 $V(\mathbf{x})$ が、以下のように正定値関数となることは明かである。

$$\forall \mathbf{x} \in \mathbf{R}^{2N},\ \mathbf{x} \neq 0 : \quad V(\mathbf{x}) > 0 \tag{A.8}$$

次に、スカラ関数 $V(\mathbf{x}(t))$ を時間微分すると、性質 1 の式 (A.4) より、

$$\begin{aligned}\dot{V}(\mathbf{x}(t)) &= \dot{\mathbf{e}}(t)^T * M(\mathbf{q}(t)) * (\ddot{\mathbf{p}}(t) - \ddot{\mathbf{q}}(t)) \\ &\quad + \frac{1}{2} * \dot{\mathbf{e}}(t)^T * \dot{M}(\mathbf{q}(t)) * \dot{\mathbf{e}}(t) + \dot{\mathbf{e}}(t)^T * K_p * \mathbf{e}(t) \end{aligned} \tag{A.9}$$

式 (A.9) に対して、式 (A.2) の右辺の $M(\mathbf{q}(t)) * \ddot{\mathbf{q}}(t)$ を求めて代入すると、

$$\begin{aligned}\dot{V}(\mathbf{x}(t)) &= \dot{\mathbf{e}}(t)^T * \{M(\mathbf{q}(t)) * \ddot{\mathbf{p}}(t) + \frac{1}{2} * \dot{M}(\mathbf{q}(t)) * \dot{\mathbf{q}}(t) \\ &\quad + S(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{q}}(t) + g(\mathbf{q}(t)) - \mathbf{f}(t)\} \\ &\quad + \frac{1}{2} * \dot{\mathbf{e}}(t)^T * \dot{M}(\mathbf{q}(t)) * \dot{\mathbf{e}}(t) + \dot{\mathbf{e}}(t)^T * K_p * \mathbf{e}(t) \end{aligned} \tag{A.10}$$

ここで、式 (A.2) のロボットに加える関節トルク $\mathbf{f}(t)$ を、次の式 (A.11) に示すようなロボット制御則によって定めるものとする。

$$\left[\begin{array}{rcl} \mathbf{f}_L(t) &:=& M(\mathbf{q}(t)) * \ddot{\mathbf{p}}(t) + H(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{p}}(t) + g(\mathbf{q}(t)) \\ &:=& M(\mathbf{q}(t)) * \ddot{\mathbf{p}}(t) + \frac{1}{2} * \dot{M}(\mathbf{q}(t)) * \dot{\mathbf{p}}(t) \\ &&\quad + S(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{p}}(t) + g(\mathbf{q}(t)) \\ \mathbf{f}_U(t) &:=& K_p * \mathbf{e}(t) + K_v * \dot{\mathbf{e}}(t) \\ \mathbf{f}(t) &:=& \mathbf{f}_L(t) + \mathbf{f}_U(t) \end{array}\right. \tag{A.11}$$

ただし、$K_v \in \mathbf{R}^{N \times N}$ は、任意の正定行列である。

式 (A.11) より求められる関節トルク $\mathbf{f}(t)$ を、式 (A.10) に代入すると、

$$\dot{V}(\mathbf{x}(t)) = -\dot{\mathbf{e}}(t)^T * S(\dot{\mathbf{q}}(t), \mathbf{q}(t)) * \dot{\mathbf{e}}(t) - \dot{\mathbf{e}}(t)^T * K_v * \dot{\mathbf{e}}(t) \tag{A.12}$$

さらに、式 (A.12) において、性質 2 の式 (A.5) を考慮すると、

$$\forall \dot{\mathbf{e}}(t) \in \mathbf{R}^N,\ \dot{\mathbf{e}}(t) \neq 0 : \quad \dot{V}(\mathbf{x}(t)) = -\dot{\mathbf{e}}(t)^T * K_v * \dot{\mathbf{e}}(t) < 0 \tag{A.13}$$

式 (A.8), 式 (A.13) より、スカラ関数 $V(\mathbf{x})$ はリアプノフ関数であり、ロボット制御系における平衡点 $\mathbf{x} = \mathbf{0}$ は、リアプノフの定理より安定となる。

付録B　共通因子・共通部分式の選択アルゴリズム

提案した局所的な簡単化手法のうち、「部分的因数分解」、「共通部分式の括り出し」、「共通因子の括り出し」において、共通因子および共通部分式を選択するための具体的な手続き $factor\_x(S,\ A)$ を示す。ここで、各簡単化手法の対象となる積和式の構造は、行列 A により与えられる。また、最終的に選択される共通因子あるいは共通部分式は、要素（定数および変数）あるいは積項の集合 Sとして返される [33]。

初めに、上記の各簡単化手法について、手続き $factor\_x(S,\ A)$ $(x = 1,\ 2,\ 3)$ の引数となる行列 A の構造と、簡単化の効果 E_xの計算方法を示す。

1．部分的因数分解における共通因子の選択

ある積和式に含まれる M個の積項が、N種類の要素から構成されるとき、行列 $A = [a[m,n]]$ $(1 \leq m \leq M,\ 1 \leq n \leq N)$ を、次のように定義する。すなわち、m 番目の積項が、n 番目の要素の h $(h \geq 1)$ 乗を含むならば $a[m,n] := h$、含まなければ $a[m,n] := 0$ とする。また、「部分的因数分解」の効果 E_1は、次式より計算する。

$$E_1(S[i],\ p[k]) := |S[i]| * (p[k] - 1) \tag{B.1}$$

ただし、$S[i]$ は共通因子となる要素の集合、$p[k]$ は共通因子を含む積項の数である。

2．共通部分式の括り出しにおける共通部分式の選択

異なる M個の積和式が、N種類の積項から構成されるとき、行列 $A = [a[m,n]]$ $(1 \leq m \leq M,\ 1 \leq n \leq N)$ を、次のように定義する。すなわち、m 番目の積和式が、n 番目の積項を含むならば $a[m,n] := 1$、含まなければ $a[m,n] := 0$ とする。また、「共通部分式の括り出し」の効果 E_2は、次式より計算する。

$$E_2(S[i],\ p[k]) := (|S[i]| - 1) * (p[k] - 1) \tag{B.2}$$

ただし、$S[i]$ は共通部分式となる積項の集合、$p[k]$ は共通部分式を含む積和式の数である。

3．共通因子の括り出しにおける共通因子の選択

異なる積和式に含まれる M個の積項が、N種類の要素から構成されるとき、行列 $A = [a[m,n]]$ $(1 \leq m \leq M,\ 1 \leq n \leq N)$ を、次のように定義する。すなわち、m 番目の積項が、n 番目の要素の h $(h \geq 1)$ 乗を含むならば $a[m,n] := h$、含まなければ $a[m,n] := 0$ とする。また、「共通因子の括り出し」の効果 E_3は、次式より計算する。

$$E_3(S[i],\ p[k]) := (|S[i]| - 1) * (p[k] - 1) \tag{B.3}$$

ただし、$S[i]$ は共通因子となる要素の集合、$p[k]$ は共通因子を含む積項の数である。

```
procedure factor_x(S, A)
variable
E[i] (1 ≤ i ≤ N) : integer;    // 簡単化の効果
P = [p[n]], R = [r[n]] (1 ≤ n ≤ N) : N × 1 vector of integer;
A = [a[m,n]], B = [b[m,n]] (1 ≤ m ≤ M, 1 ≤ n ≤ N) : M × N matrix of integer;
K, S[i] := ∅ (0 ≤ i ≤ N) : set of integer;    // 要素・積項の集合
S := ∅ : set of S[i];    // 要素・積項の集合（共通因子・共通部分式）
done : boolean;    // TRUE or FALSE
constant
M1 = [1...1] : M × 1 vector of integer;
N1 = [1...1] : N × 1 vector of integer;
begin
01 : done := FALSE;  i := 1;
02 : while (not done) {
03 :       for m = 1 to M do {    // 行列 A から行列 Bを作成する
04 :             for n = 1 to N do {
05 :                   if (a[m,n] > 0) b[m,n] := 1;  else b[m,n] := 0;
06 :             }
07 :       }
08 :       P := (M1)^T * B;  K := {k | p[k] > p[n] and p[k] ≥ 2};
09 :       if (|K| = 0) done := TRUE;    // |K| は Kの要素数
10 :       if (|K| > 1) {
11 :             R := (N1)^T * (B)^T * B;  K := {k ∈ K | r[k] > r[n]};
12 :             if (|K| > 1) K := {k ∈ K | k > n};
13 :       }   // |K| = 1 and K = { k }
14 :       S[i] := S[i − 1] ∪ K;  E[i] := E_x(S[i], p[k]);  i := i + 1;
15 :       for m = 1 to M do {    // 行列 A の更新
16 :             if (a[m,k] > 0) a[m,k] := a[m,k] − 1;
17 :             else for n = 1 to N do { a[m,n] := 0; }
18 :       }
19 : }
20 : S := {S[j] | E[j] ≥ 1 and E[j] > E[i], (0 ≤ i ≤ N, i ≠ j)};
21 : if (|S| > 1) {
22 :       S := {S[j] ∈ S | |S[j]| > |S[i]|, (0 ≤ i ≤ N, i ≠ j)};
23 :       if (|S| > 1) S := {S[j] ∈ S | j < i, (0 ≤ i ≤ N, i ≠ j)};
24 : }   // |S| = 1 and S = { S[j] }
25 : return(S);
end.
```

付録C　マジック・テーブル

マジック・テーブルとは、汎用的な2次元配列のデータ構造を提供するクラスである。このクラスが提供する2次元配列のデータ構造は、多重リスト構造によって実装されており、以下のような特徴を有する。

(1) 配列のサイズは可変であり、常に必要最小限のメモリを使用できる。

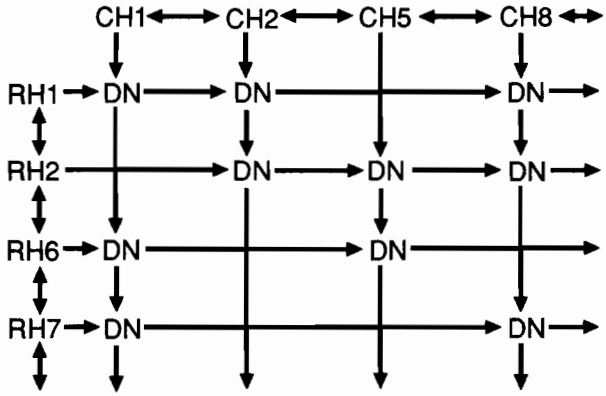
(2) クラスとして定義されているために、継承による機能の拡張が容易である。

(3) データ処理に関連する豊富な操作が、メソッドとして用意されている。

(4) 実装の細部を意識することなく、通常の配列と同様に扱うことができる。

(5) 通常の配列と比較すると、若干、処理時間を要する。

マジック・テーブルの概念図を図C.1に示す。マジック・テーブルは、配列の要素に相当するデータ・ノード（DN）、行ヘッダー（RH）、列ヘッダー（CH）から構成される。各データ・ノードは、隣接するデータ・ノードを指す行方向（→）のポインタと、列方向（↓）のポインタを持つ。また、図C.1には示されていないが、各データ・ノード（DN）は、所属する行ヘッダー（RH）と列ヘッダー（CH）へのポインタも持つ。



図C.1　マジック・テーブルの概念図

マジック・テーブルは、データ・ノード（DN）、ヘッダー（RH・CH）、マジック・テーブル自身に対応する3種類のクラスにより定義される。以下に、これらのクラスの宣言部を、オブジェクト指向言語の1つであるGNU C++を用いて記述する。

```
class GP_node { // データ・ノード (DN) のクラス
friend class GP_header;
friend class GP_table;
protected:
        Ctype dat;                  // データの型
        GP_node *c_next;            // 列ノードへのポインタ
        class GP_header *c_header; // 列ヘッダーへのポインタ
        GP_node *r_next;            // 行ノードへのポインタ
        class GP_header *r_header; // 行ヘッダーへのポインタ

        // コンストラクタ
        GP_node():dat(0),c_next(NULL),c_header(NULL),
        r_next(NULL),r_header(NULL){}
        GP_node( const GP_node &source ):dat(source.dat),
        c_next(NULL),c_header(NULL),r_next(NULL),r_header(NULL){}
};

class GP_header { // 行 (RH)・列 (CH) ヘッダーのクラス
friend class GP_table;
protected:
        Htype dat;                  // データの型 (別に定義)
        GP_header *before;          // 前のヘッダーへのポインタ
        GP_header *next;            // 後のヘッダーへのポインタ
        GP_node *node;              // データ・ノードへのポインタ
public:
        // コンストラクタ
        GP_header():dat(0),before(NULL),next(NULL),node(NULL){}
        GP_header( const GP_header &source ):
        dat(source.dat),before(NULL),next(NULL),node(NULL){}
};
```

```
class GP_table { // マジック・テーブルのクラス
protected:
        GP_header *row_header;     // 行ヘッダーへのポインタ
        GP_header *column_header;  // 列ヘッダーへのポインタ

        // コンストラクタ
        GP_header* r_header_search( const Htype& h_data ) const;
        GP_header* c_header_search( const Htype& h_data ) const;
        GP_header* r_header_insert( const Htype& h_data );
        GP_header* c_header_insert( const Htype& h_data );
        int r_header_delete( const Htype& h_data );
        int c_header_delete( const Htype& h_data );
public:
```

マジック・テーブルには、以下のようなメンバ関数が用意されている。

```
        init                    // マジック・テーブルの初期化
        read_cell               // ノードへの読み込み
        write_cell              // ノードへの書き込み
        delete_cell             // ノードの削除
        delete_row              // 指定行の削除
        delete_column           // 指定列の削除
        get_row_size            // 指定行のノード数を調べる
        get_column_size         // 指定列のノード数を調べる
        get_row_header          // 行ヘッダーの集合を返す
        get_column_header       // 列ヘッダーの集合を返す
        get_row_cell_header     // ノードに対応する列ヘッダー集合を返す
        get_column_cell_header // ノードに対応する行ヘッダー集合を返す
        print                   // テーブルの内容を表示する
        operator=               // 代入演算子 (=) によりテーブルをコピーする
};
```

付録D　最適化コンパイラのプログラム

最適化コンパイラの主要部分を、オブジェクト指向言語GNU C++を用いて記述する。まず、数式を保持するための5種類の「表」、すなわち、「式の表」、「項の表」、「関数の表」、「定数式の表」、「定数項の表」を保持する「基底クラス」を定義する。これら5種類の「表」は、付録Cに示した汎用的な「表」のクラスである「マジック・テーブル」のオブジェクトとして実現される。次に、上記の「基底クラス」を継承して、機能を拡張するという方法で、最適化コンパイラの各フェーズの処理をメソッドとするクラスが定義される。最終的に、すべてのクラスの機能を継承した「コード化」のクラスが、最適化コンパイラ自身を定義することになる。従って、最適化コンパイラのメイン・プログラムでは、「コード化」のクラスのオブジェクトとして最適化コンパイラを生成した後、各フェーズの処理を実行するためのメソッドを順番に起動している。

1．主要なクラスの宣言部

```
class Base { // 基底クラス
protected:
        GP_table formula_table;          // 式の表
        GP_table term_table;             // 項の表
        GP_table func_table;             // 関数表
        GP_table const_formula_table;    // 定数式の表
        GP_table const_term_table;       // 定数項の表
        Dictionary symbol_table;         // 記号表
        ifstream inputfile;              // ソース・プログラム
public:
        Base( const ifstream& file ) : inputfile( file ) {}
};

class Parser : public Base { // 構文解析のクラス
protected:
        void pars( const int& code);       // 構文解析
        int get_token( int token=DIC_VAR); // 字句解析
        int formula(GP_list& st1);         // 式の処理
        void term(GP_list& st1);           // 項の処理
        int regist_variable();             // 変数登録
        int regist_constant();             // 定数登録
public:
        Parser( const ifstream& file ) : Base( file ) {}
        void parsing();                    // 構文解析の実行
};
```

```
class Simpler : public Parser { // 簡単化のクラス
protected:
        // 局所的簡単化の実行
        int Exec_LOM1();        // 部分的因数分解
        int Exec_LOM2();        // 共通部分式の括り出し
        int Exec_LOM3();        // 共通因数の括り出し
        int Exec_LOM4();        // 定数の畳み込み
        int Exec_LOM5();        // 代入式の消去
public:
        // 大域的簡単化の実行
        Simpler( const ifstream& file ) : Parser( file ) {}
        void optimizing( const int& verbose );
};

class Encoder : public Simpler { // コード化のクラス
protected:
        WK_table priority_table;  // データ依存関係表
        void Pr_table_init();     // データ依存関係表に計算式をセット
        void Pr_table_C_init();   // データ依存関係表に定数式をセット
public:
        Encoder( const ifstream& file ) : Simpler( file ) {}
        void encoding( const int& mode );       // 計算式のコードを出力
        void encoding_const( const int& mode ); // 定数式のコードを出力
};
```

2．最適化コンパイラのメイン・プログラム

```
int main( int argc, char *argv[] )
{
        int outputmode = 0;      // 0:REDUCE 形式, 1:C 言語形式
        int verbose = 0;         // 1:処理過程の表示
        char *filename = NULL;   // ソース・プログラム

        if (argc != 2) exit(1);
        else filename = argv[1];
        ifstream inputfile(filename);  // ソース・プログラムの指定

        Encoder object( inputfile );   // 最適化コンパイラの生成
```

```
        object.parsing();                          // 構文解析の実行
        object.optimizing( verbose );              // 簡単化の実行
        object.encoding_const( outputmode );       // 定数式の出力
        object.encoding( outputmode );             // 計算式の出力

        exit(0);
}
```

3．最適化コンパイラの使用方法

最適化コンパイラを使用するには、以下に示すようにキー入力する。**input_file** とは、テキスト形式のソース・プログラム（ロボット制御則）である。得られたオブジェクト・プログラム（**output_file**）をファイルに落とす場合には、**>output_file** のようにリダイレクトを使用する。

```
opt [options] input_file [>optput_file] <return>
```

オプション [**options**] には以下の種類があり、省略した場合は**-r** と判断される。

-h：最適化コンパイラの使用方法を表示する。
-r：オブジェクト・プログラムを REDUCE 形式で出力する。
-c：オブジェクト・プログラムを C 言語形式で出力する。
-v：実行過程をリアルタイムで表示する。
-t：最適化に要した時間を最後に出力する。
-n：ソース・プログラムをそのまま出力する。

付録E　ソース／オブジェクト・プログラム

最適化コンパイラの評価に使用したソース・プログラム（ロボット制御則）と、得られたオブジェクト・プログラム（REDUCE形式）を示す。

例題1：多関節ロボット・アームの逆動力学計算

```
%%%%%%%% source-1(source program) %%%%%%%%
\\ 定数パラメータの指定
constant: g,j1zz,j2xx,j2zz,j3xx,j3zz,L1z,L2x,L3x,mr2x,mr3x;

f1 :=   j1zz*ddq1 + j2xx*sin(q2)**2*ddq1 + 2*j2xx*sin(q2)*cos(q2)*dq1*dq2
      - 2*j3xx*sin(q2)**2*sin(q3)**2*ddq1
      - 4*j3xx*sin(q2)**2*sin(q3)*cos(q3)*dq1*dq2
      - 4*j3xx*sin(q2)**2*sin(q3)*cos(q3)*dq1*dq3 + j3xx*sin(q2)**2*ddq1
      - 4*j3xx*sin(q2)*cos(q2)*sin(q3)**2*dq1*dq2
      - 4*j3xx*sin(q2)*cos(q2)*sin(q3)**2*dq1*dq3
      + 2*j3xx*sin(q2)*cos(q2)*sin(q3)*cos(q3)*ddq1
      + 2*j3xx*sin(q2)*cos(q2)*dq1*dq2 + 2*j3xx*sin(q2)*cos(q2)*dq1*dq3
      + j3xx*sin(q3)**2*ddq1 + 2*j3xx*sin(q3)*cos(q3)*dq1*dq2
      + 2*j3xx*sin(q3)*cos(q3)*dq1*dq3
      + 4*L2x*mr3x*sin(q2)**2*sin(q3)*dq1*dq2
      + 2*L2x*mr3x*sin(q2)**2*sin(q3)*dq1*dq3
      - 2*L2x*mr3x*sin(q2)**2*cos(q3)*ddq1 + 2*L2x*mr3x*cos(q3)*ddq1
      - 2*L2x*mr3x*sin(q2)*cos(q2)*sin(q3)*ddq1
      - 4*L2x*mr3x*sin(q2)*cos(q2)*cos(q3)*dq1*dq2
      - 2*L2x*mr3x*sin(q2)*cos(q2)*cos(q3)*dq1*dq3
      - 2*L2x*mr3x*sin(q3)*dq1*dq2 - 2*L2x*mr3x*sin(q3)*dq1*dq3$
f2 :=   g*mr2x*cos(q2) - g*mr3x*sin(q2)*sin(q3)
      + g*mr3x*cos(q2)*cos(q3) - j2xx*sin(q2)*cos(q2)*dq1**2
      + j2zz*ddq2 + 2*j3xx*sin(q2)**2*sin(q3)*cos(q3)*dq1**2
      + 2*j3xx*sin(q2)*cos(q2)*sin(q3)**2*dq1**2
      - j3xx*sin(q2)*cos(q2)*dq1**2 - j3xx*sin(q3)*cos(q3)*dq1**2
      + j3zz*ddq2 + j3zz*ddq3 - 2*L2x*mr3x*sin(q2)**2*sin(q3)*dq1**2
      + 2*L2x*mr3x*sin(q2)*cos(q2)*cos(q3)*dq1**2
      + L2x*mr3x*sin(q3)*dq1**2 - 2*L2x*mr3x*sin(q3)*dq2*dq3
      - L2x*mr3x*sin(q3)*dq3**2 + 2*L2x*mr3x*cos(q3)*ddq2
      + L2x*mr3x*cos(q3)*ddq3$
f3 := - g*mr3x*sin(q2)*sin(q3) + g*mr3x*cos(q2)*cos(q3)
      + 2*j3xx*sin(q2)**2*sin(q3)*cos(q3)*dq1**2
      + 2*j3xx*sin(q2)*cos(q2)*sin(q3)**2*dq1**2
```

```
      - j3xx*sin(q2)*cos(q2)*dq1**2 - j3xx*sin(q3)*cos(q3)*dq1**2
      + j3zz*ddq2 + j3zz*ddq3 - L2x*mr3x*sin(q2)**2*sin(q3)*dq1**2
      + L2x*mr3x*sin(q2)*cos(q2)*cos(q3)*dq1**2
      + L2x*mr3x*sin(q3)*dq1**2 + L2x*mr3x*sin(q3)*dq2**2
      + L2x*mr3x*cos(q3)*ddq2$

%%%%%%%% object-1(object program) %%%%%%%%
\\ 変数の計算プログラム
\\ 中間変数 v の添字が一部欠落しているのは、「代入式の削除」の結果である。
 v1 := ddq2 + ddq3$              v16 := cos(q3)$
 v2 := dq3 + dq2$                v17 := v16*v13$
 v3 := - dq3 - dq2$              v18 := - v11 + v17$
 v4 := dq1**2$                   v19 := v6*v18$
 v5 := sin(q2)$                  v20 := v16*ddq2$
 v6 := v4*v5$                    v21 := v19 + v12*dq2 + v20$
 v7 := a17 - v6*j2xx$            v22 := - v12*dq3 + v19 + v20$
 v8 := sin(q3)$                  v23 := - v8*dq3**2 + v16*ddq3$
 v9 := v8**2$                    v24 := v16*a18$
v10 := - v9*a20$                 v25 := - v16*a21 + v10 + a22$
v11 := v8*v5$                    v26 := - v24 + a15 + v10$
v12 := v8*dq2$                   v28 := - v9*a15 + a23 - v24$
v13 := cos(q2)$                  v29 := v26*dq3 + v25*dq2$
v14 := - v8*a14 - v4*v13*j3xx$   v30 := v16*a15$
v15 := v13*v8$                   v31 := - a18 + v30$

v32 := v28*v5 + v15*v31$
v34 := v32*v5 + j1zz + v9*j3xx + v24$
v35 := v30*v2 + v3*a18$
v37 := v15 + v16*v5$
v38 := - v16*j3xx + a16 + v37*v5*a15$
v39 := v17*a14 + v4*v38*v8 + v14*v5 + v1*j3zz$
 f3 := v39 + v21*a16$
 f2 := ddq2*j2zz + v22*a18 + v7*v13 + v23*a16 + v39$
v41 := dq3*a18 + dq2*a21 + v3*v16*a20$
v42 := v11*v41 + v29*v13$
v43 := v35*v8 + v42*v5$
 f1 := v34*ddq1 + v43*dq1$
```

```
\\ 定数の計算プログラム
a14 := g*mr3x$                  a19 := j2xx*2$
a15 := j3xx*2$                  a20 := j3xx*4$
a16 := L2x*mr3x$                a21 := L2x*mr3x*4$
a17 := g*mr2x$                  a22 := a15 + a19$
a18 := L2x*mr3x*2$              a23 := j2xx + j3xx$
```

例題２：極座標ロボット・アームの逆動力学計算

```
%%%%%%%% source-2(source program) %%%%%%%%
\\ 定数パラメータの指定
constant: g,j1yy,j2xx,j2yy,j2zz,j3xx,j3yy,j3zz,r2y,r3z,L2,m2,m3;

f1 := - j3zz*dq1*dq2*sin(2*q2) + j3zz*ddq1 - j3zz*ddq1*sin(q2)**2
      + j3xx*dq1*dq2*sin(2*q2) + j3xx*ddq1*sin(q2)**2
      - j2zz*dq1*dq2*sin(2*q2) + j2zz*ddq1 - j2zz*ddq1*sin(q2)**2
      + j2xx*dq1*dq2*sin(2*q2) + j2xx*ddq1*sin(q2)**2 + j1yy*ddq1
      + 2*m3*dq1*dq3*q3*sin(q2)**2 + m3*dq1*dq2*q3**2*sin(2*q2)
      + m3*ddq1*q3**2*sin(q2)**2 - 2*m3*L2*dq2*dq3*cos(q2)
      + m3*L2*dq2**2*q3*sin(q2) - m3*L2*ddq3*sin(q2) + m3*L2**2*ddq1
      - m3*L2*ddq2*q3*cos(q2) + 2*m3*r3z*dq1*dq3*sin(q2)**2
      + 2*m3*r3z*dq1*dq2*q3*sin(2*q2) + 2*m3*r3z*ddq1*q3*sin(q2)**2
      + m3*r3z*L2*dq2**2*sin(q2) - m3*r3z*L2*ddq2*cos(q2) + m2*L2**2*ddq1
      + 2*m2*r2y*L2*ddq1 + g*m3*q3*cos(q1)*sin(q2) - g*m3*L2*sin(q1)
      + g*m3*r3z*cos(q1)*sin(q2) - g*m2*L2*sin(q1) - g*m2*r2y*sin(q1)$
f2 := (j3zz*dq1**2*sin(2*q2) + 2*j3yy*ddq2 - j3xx*dq1**2*sin(2*q2)
      + j2zz*dq1**2*sin(2*q2) + 2*j2yy*ddq2 - j2xx*dq1**2*sin(2*q2)
      + 4*m3*dq2*dq3*q3 - m3*dq1**2*q3**2*sin(2*q2) + 2*m3*ddq2*q3**2
      - 2*m3*L2*ddq1*q3*cos(q2) + 4*m3*r3z*dq2*dq3
      - 2*m3*r3z*dq1**2*q3*sin(2*q2) + 4*m3*r3z*ddq2*q3
      - 2*m3*r3z*L2*ddq1*cos(q2) + 2*g*m3*q3*cos(q2)*sin(q1)
      + 2*g*m3*r3z*cos(q2)*sin(q1))/2$
f3 := - m3*dq2**2*q3 - m3*dq1**2*q3*sin(q2)**2 + m3*ddq3 - m3*r3z*dq2**2
      - m3*r3z*dq1**2*sin(q2)**2 + g*m3*sin(q1)*sin(q2)$
```

```
%%%%%%%% object-2(object program) %%%%%%%%
\\ 変数の計算プログラム
 v1 := 2*q2$                    v11 := a24 + a26*q3$
 v2 := sin(v1)$                 v12 := m3*q3$
 v3 := sin(q2)$                 v13 := - a19 - v12$
 v4 := cos(q2)$                 v14 := a24 + v12$
 v5 := cos(q1)$                 v15 := q3*v14 + a39$
 v6 := sin(q1)$                 v16 := dq1*v2*v15 - a25*dq3*v4$
 v7 := 1/(2)$                   v17 := dq1*dq3*v11 + ddq1*v15$
 v8 := - a22*ddq1 + a20*v6$     v18 := dq2**2$
 v9 := a21*v6 - a25*ddq1$       v19 := ddq3 - q3*v18$
v10 := - a36 - a38*q3$          v20 := a18*v5 + a38*v18$

v21 := a32*v5 + a36*v18 - a38*ddq3 + v3*v17 + q3*v20$
 f1 := v3*v21 + a41*ddq1 + ddq2*v4*v10 + a40*v6 + dq2*v16$
v22 := dq2*dq3$
v23 := dq1**2$
v24 := v3*v13*v23 + a18*v6$
 f3 := - a19*v18 + v3*v24 + m3*v19$
v25 := v2*v23$
v26 := - m3*v25 + a26*ddq2$
v27 := a27*v22 - a24*v25 + a23*ddq2 + q3*v26 + v4*v9$
v28 := a23*v22 + q3*v27 + a43*v25 + v4*v8 + a42*ddq2$
 f2 := v28*v7$

\\ 定数の計算プログラム
a18 := m3*g$                a31 := L2*m2*g$
a19 := r3z*m3$              a32 := r3z*m3*g$
a20 := r3z*m3*g*2$          a33 := L2*m3*g$
a21 := m3*g*2$              a34 := r2y*L2*m2*2$
a22 := r3z*L2*m3*2$         a35 := L2**2*m2$
a23 := r3z*m3*4$            a36 := r3z*L2*m3$
a24 := r3z*m3*2$            a37 := L2**2*m3$
a25 := L2*m3*2$             a38 := L2*m3$
a26 := m3*2$                a39 := j2xx-j2zz+j3xx-j3zz$
a27 := m3*4$                a40 := - a30 - a31 - a33$
a28 := j2yy*2$              a41 := j1yy + j2zz + j3zz + a34 + a35 + a37$
a29 := j3yy*2$              a42 := a28 + a29$
a30 := r2y*m2*g$            a43 := - j2xx + j2zz - j3xx + j3zz$
```

例題３：多関節ロボット・アームの線形化補償器

```
%%%%%%%% source-3(source program) %%%%%%%%
\\ 定数パラメータの指定
constant: g, L1, L2, L3, b1, b2, b3, b4, b5;

f1 :=   2*b5*L2*cos(q3)*ddp1 - b5*L2*cos(q3)*sin(q2)*cos(q2)*dq3*dp1
      - 2*b5*L2*cos(q3)*sin(q2)*cos(q2)*dq2*dp1
      - b5*L2*cos(q3)*sin(q2)*cos(q2)*dq1*dp3
      - 2*b5*L2*cos(q3)*sin(q2)*cos(q2)*dq1*dp2
      - 2*b5*L2*cos(q3)*sin(q2)**2*ddp1 - b5*L2*sin(q3)*dq3*dp1
      - b5*L2*sin(q3)*dq2*dp1 - b5*L2*sin(q3)*dq1*dp3
      - b5*L2*sin(q3)*dq1*dp2 - 2*b5*L2*sin(q3)*sin(q2)*cos(q2)*ddp1
      + b5*L2*sin(q3)*sin(q2)**2*dq3*dp1
      + 2*b5*L2*sin(q3)*sin(q2)**2*dq2*dp1
      + b5*L2*sin(q3)*sin(q2)**2*dq1*dp3
      + 2*b5*L2*sin(q3)*sin(q2)**2*dq1*dp2 + b3*ddp1
      - b3*sin(q2)*cos(q2)*dq3*dp1 - b3*sin(q2)*cos(q2)*dq2*dp1
      - b3*sin(q2)*cos(q2)*dq1*dp3 - b3*sin(q2)*cos(q2)*dq1*dp2
      - b3*sin(q2)**2*ddp1 - b3*sin(q3)*cos(q3)*dq3*dp1
      - b3*sin(q3)*cos(q3)*dq2*dp1 - b3*sin(q3)*cos(q3)*dq1*dp3
      - b3*sin(q3)*cos(q3)*dq1*dp2
      - 2*b3*sin(q3)*cos(q3)*sin(q2)*cos(q2)*ddp1
      + 2*b3*sin(q3)*cos(q3)*sin(q2)**2*dq3*dp1
      + 2*b3*sin(q3)*cos(q3)*sin(q2)**2*dq2*dp1
      + 2*b3*sin(q3)*cos(q3)*sin(q2)**2*dq1*dp3
      + 2*b3*sin(q3)*cos(q3)*sin(q2)**2*dq1*dp2
      - b3*sin(q3)**2*ddp1 + 2*b3*sin(q3)**2*sin(q2)*cos(q2)*dq3*dp1
      + 2*b3*sin(q3)**2*sin(q2)*cos(q2)*dq2*dp1
      + 2*b3*sin(q3)**2*sin(q2)*cos(q2)*dq1*dp3
      + 2*b3*sin(q3)**2*sin(q2)*cos(q2)*dq1*dp2 + b2*ddp1
      + 2*b3*sin(q3)**2*sin(q2)**2*ddp1 - b2*sin(q2)*cos(q2)*dq2*dp1
      - b2*sin(q2)*cos(q2)*dq1*dp2 - b2*sin(q2)**2*ddp1$
f2 :=   b5*L2*cos(q3)*ddp3 + 2*b5*L2*cos(q3)*ddp2
      + 2*b5*L2*cos(q3)*sin(q2)*cos(q2)*dq1*dp1
      - b5*L2*sin(q3)*dq3*dp3 - b5*L2*sin(q3)*dq3*dp2
      - b5*L2*sin(q3)*dq2*dp3 + b5*L2*sin(q3)*dq1*dp1
      - 2*b5*L2*sin(q3)*sin(q2)**2*dq1*dp1 + b3*ddp3
      + b3*sin(q2)*cos(q2)*dq1*dp1 + b3*sin(q3)*cos(q3)*dq1*dp1
      - 2*b3*sin(q3)*cos(q3)*sin(q2)**2*dq1*dp1
```

```
      - 2*b3*sin(q3)**2*sin(q2)*cos(q2)*dq1*dp1
      + b2*sin(q2)*cos(q2)*dq1*dp1 + b1*ddp2
      + g*b5*cos(q3)*cos(q2) - g*b5*sin(q3)*sin(q2) + g*b4*cos(q2)$
f3 :=   b5*L2*cos(q3)*ddp2 + b5*L2*cos(q3)*sin(q2)*cos(q2)*dq1*dp1
      + b5*L2*sin(q3)*dq2*dp2 + b5*L2*sin(q3)*dq1*dp1
      - b5*L2*sin(q3)*sin(q2)**2*dq1*dp1 + b3*ddp3 + b3*ddp2
      + b3*sin(q2)*cos(q2)*dq1*dp1 + b3*sin(q3)*cos(q3)*dq1*dp1
      - 2*b3*sin(q3)*cos(q3)*sin(q2)**2*dq1*dp1
      - 2*b3*sin(q3)**2*sin(q2)*cos(q2)*dq1*dp1
      + g*b5*cos(q3)*cos(q2) - g*b5*sin(q3)*sin(q2)$

%%%%%%%% object-3(object program) %%%%%%%%
\\ 変数の計算プログラム
 v1 := cos(q3)$                 v17 := dp2 + dp3$
 v2 := sin(q2)$                 v18 := dp1*v16 + dq1*v17$
 v3 := cos(q2)$                 v19 := v1*v18 + ddp1*v4$
 v4 := sin(q3)$                 v20 := dp1*v14 + dq1*v15 + a16*v19$
 v5 := - v3*v4 - v1*v2$         v21 := - b3 - a13*v1$
 v6 := a13*ddp3 + a17*ddp2$     v22 := dq1*dp2 + dp1*dq2$
 v7 := - dp3 - dp2$             v23 := dq3*dp1 + dq1*dp3$
 v8 := - dq2*dp3 + dq3*v7$      v24 := - ddp1*v1 + v4*v23$
 v9 := b3*v1 + a13$             v25 := - a17*ddp1 + a16*v24$
v10 := a16*v2*v5 + v9$          v26 := v2*v4$
v11 := v4*v10 + b3*v2*v3$       v27 := - v26 + v1*v3$
v12 := - dq3 - dq2$             v28 := b2*v3 + a17*v27$
v13 := dq1*v7 + dp1*v12$        v29 := v4**2$
v14 := a17*dq2 + a13*dq3$       v30 := 1 - v29$
v15 := a13*dp3 + a17*dp2$       v31 := a17*v1$
v16 := dq3 + dq2$               v32 := - v31 + a18$

v33 := v2**2*v32 + b2 + b3*v30 + v31$
v34 := a16*v29 + v32$
v35 := v34*v22 + v4*v25 + v21*v23$
v36 := v3*v35 + v20*v26$
 f1 := ddp1*v33 + v4*v13*v9 + v2*v36$
v37 := dp1*dq1$
v38 := b3*ddp3 + v11*v37 + a15*v27$
v39 := v2*v37$
 f2 := b1*ddp2 + a14*v3 + v38 + a13*v4*v8 + v28*v39 + v1*v6$
v40 := v27*v39 + dq2*dp2*v4 + ddp2*v1$
```

```
 f3 := b3*ddp2 + v38 + a13*v40$

\\ 定数の計算プログラム
a13 := L2*b5$                    a16 := b3*2$
a14 := g*b4$                     a17 := L2*b5*2$
a15 := g*b5$                     a18 := - b2 - b3$
```

例題 4 ：極座標ロボット・アームの線形化補償器

```
%%%%%%%% source-4(source program) %%%%%%%%
\\ 定数パラメータの指定
constant: g,j1yy,j2xx,j2yy,j2zz,j3xx,j3yy,j3zz,r2y,r3z,L2,m2,m3;

f1 := ( - j3zz*dp2*dq1*sin(2*q2) - j3zz*dp1*dq2*sin(2*q2)
      + 2*j3zz*ddp1 - 2*j3zz*ddp1*sin(q2)**2 + j3xx*dp2*dq1*sin(2*q2)
      + j3xx*dp1*dq2*sin(2*q2) + 2*j3xx*ddp1*sin(q2)**2
      - j2zz*dp2*dq1*sin(2*q2) - j2zz*dp1*dq2*sin(2*q2)
      + 2*j2zz*ddp1 - 2*j2zz*ddp1*sin(q2)**2 + j2xx*dp2*dq1*sin(2*q2)
      + j2xx*dp1*dq2*sin(2*q2) + 2*j2xx*ddp1*sin(q2)**2 + 2*j1yy*ddp1
      + 2*m3*dp3*dq1*q3*sin(q2)**2 + m3*dp2*dq1*q3**2*sin(2*q2)
      + 2*m3*dp1*dq3*q3*sin(q2)**2 + m3*dp1*dq2*q3**2*sin(2*q2)
      + 2*m3*ddp1*q3**2*sin(q2)**2 - 2*m3*L2*dp3*dq2*cos(q2)
      - 2*m3*L2*dp2*dq3*cos(q2) + 2*m3*L2*dp2*dq2*q3*sin(q2)
      - 2*m3*L2*ddp3*sin(q2) - 2*m3*L2*ddp2*q3*cos(q2) + 2*m3*L2**2*ddp1
      + 2*m3*r3z*dp3*dq1*sin(q2)**2 + 2*m3*r3z*dp2*dq1*q3*sin(2*q2)
      + 2*m3*r3z*dp1*dq3*sin(q2)**2 + 2*m3*r3z*dp1*dq2*q3*sin(2*q2)
      + 4*m3*r3z*ddp1*q3*sin(q2)**2 + 2*m3*r3z*L2*dp2*dq2*sin(q2)
      - 2*m3*r3z*L2*ddp2*cos(q2) + 2*m2*L2**2*ddp1 + 4*m2*r2y*L2*ddp1
      + 2*g*m3*q3*cos(q1)*sin(q2) - 2*g*m3*L2*sin(q1) - 2*g*m2*L2*sin(q1)
      + 2*g*m3*r3z*cos(q1)*sin(q2) - 2*g*m2*r2y*sin(q1))/2$
f2 := (j3zz*dp1*dq1*sin(2*q2) + 2*j3yy*ddp2 - j3xx*dp1*dq1*sin(2*q2)
      + j2zz*dp1*dq1*sin(2*q2) + 2*j2yy*ddp2 - j2xx*dp1*dq1*sin(2*q2)
      + 2*m3*dp3*dq2*q3 + 2*m3*dp2*dq3*q3 - m3*dp1*dq1*q3**2*sin(2*q2)
      + 2*m3*ddp2*q3**2 - 2*m3*L2*ddp1*q3*cos(q2) + 2*m3*r3z*dp3*dq2
      + 2*m3*r3z*dp2*dq3 - 2*m3*r3z*dp1*dq1*q3*sin(2*q2)
      + 4*m3*r3z*ddp2*q3 - 2*m3*r3z*L2*ddp1*cos(q2)
      + 2*g*m3*q3*cos(q2)*sin(q1) + 2*g*m3*r3z*cos(q2)*sin(q1))/2$
f3 := - m3*dp2*dq2*q3 - m3*dp1*dq1*q3*sin(q2)**2
      + m3*ddp3 - m3*r3z*dp2*dq2
      - m3*r3z*dp1*dq1*sin(q2)**2 + g*m3*sin(q1)*sin(q2)$
```

```
%%%%%%%% object-4(object program) %%%%%%%%
\\ 変数の計算プログラム
 v1 := 2*q2$                    v11 := dp1*dq3 + dq1*dp3$
 v2 := sin(v1)$                 v12 := q3*ddp2$
 v3 := sin(q2)$                 v13 := m3*q3$
 v4 := cos(q2)$                 v14 := - a19 - v13$
 v5 := cos(q1)$                 v15 := - v13 - a24$
 v6 := sin(q1)$                 v16 := a24 + v13$
 v7 := 1/(2)$                   v17 := q3*v16 + a41$
 v8 := - a22*ddp1 + a20*v6$     v18 := dp2*dq2$
 v9 := - a25*ddp1 + a21*v6$     v19 := a21*v5 + a25*v18$
v10 := dp2*dq1 + dp1*dq2$       v20 := dq1*dp1$

v21 := a18*v6 + v3*v14*v20$
 f3 := m3*ddp3 + v14*v18 + v3*v21$
v22 := v2*v20$
v23 := dq2*dp3$
v24 := dp2*dq3$
v25 := v24 + v23$
v26 := v25 + v12$
v27 := a23*ddp2 + a26*v26 + v15*v22 + v4*v9$
v28 := q3*v27 + a45*v22 + v4*v8 + a24*v25 + a44*ddp2$
 f2 := v7*v28$
v29 := - v12 - v24 - v23$
v30 := a25*v29 - a22*ddp2$
v31 := a26*q3$
v32 := a24 + v31$
v33 := a23 + v31$
v34 := q3*v33 + a40$
v35 := ddp1*v34 + v11*v32$
v36 := v3*v35 + a20*v5+a22*v18 - a25*ddp3 + q3*v19$
v37 := v3*v36 + a43*ddp1 + a42*v6 + v2*v17*v10 + v4*v30$
 f1 := v37*v7$
```

\\ 定数の計算プログラム

```
a18 := g*m3$                    a28 := j3yy*2$
a19 := r3z*m3$                  a29 := g*r2y*m2*2$
a20 := g*r3z*m3*2$              a30 := g*L2*m2*2$
a21 := g*m3*2$                  a31 := g*L2*m3*2$
a22 := r3z*L2*m3*2$             a32 := r2y*L2*m2*4$
a23 := r3z*m3*4$                a33 := L2**2*m2*2$
a24 := r3z*m3*2$                a34 := L2**2*m3*2$
a25 := L2*m3*2$                 a35 := j1yy*2$
a26 := m3*2$                    a36 := j2xx*2$
a27 := j2yy*2$                  a37 := j2zz*2$

a38 := j3xx*2$
a39 := j3zz*2$
a40 := a36 - a37 + a38 - a39$
a41 := j2xx - j2zz + j3xx - j3zz$
a42 := - a29 - a30 - a31$
a43 := a32 + a33 + a34 + a35 + a37 + a39$
a44 := a27 + a28$
a45 := - j2xx + j2zz - j3xx + j3zz$
```

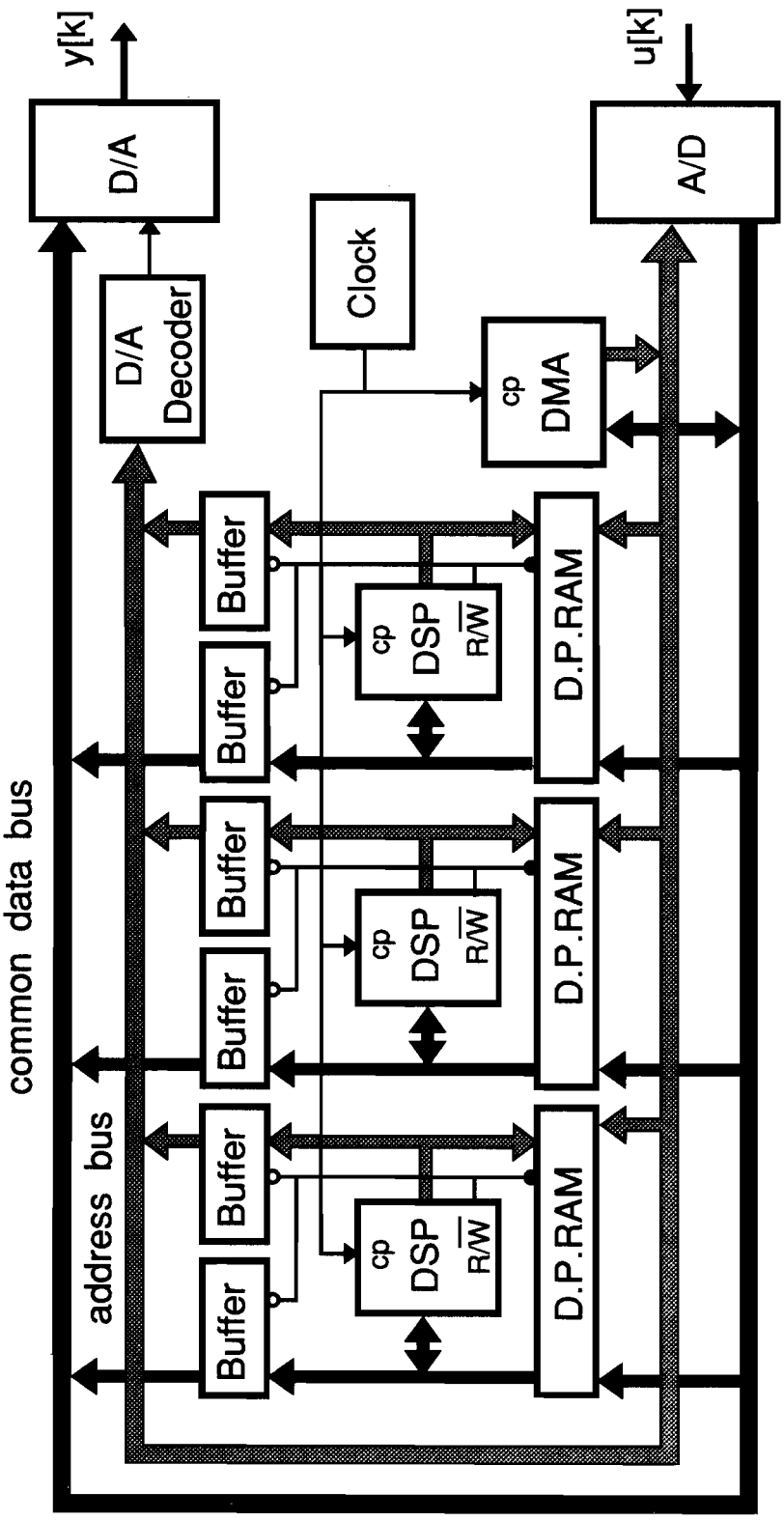
付録F　ディジタル制御器のハードウェア構成

実際に試作したディジタル制御器のハードウェア構成を、図F.1に示す[86],[89]。このディジタル制御器では、3個のDSP（TMS320C25）[81]を並列化している。

入力データ $u_q[k]$ $(Q \geq q \geq 1)$ をブロードキャストするDMAについては、専用の順序回路によって実装することも可能であが、並列化するDSPと同じタイプのDSPをDMAとして使用すると、同期などの調整が容易となる。従って、図F.1のディジタル制御器では、DSP（TMS320C25）によりDMAを実現している。また、すべてのDSPとDMA（DSP）は、共通な外部クロック信号（40 [MHz]）により同期を取っている。

各DSPの外部データ・メモリ（RAM）には、RAMに対するデータの書き込みと読み出しの競合を防止するため、デュアルポートメモリ（D.P.RAM / 4 [kw]）を採用している。すなわち、D.P.RAMの2つの入出力ポートを、共通データ・バス側を書き込み専用ポート、DSP側を読み出し専用ポートに固定して使用することで、同じアドレスに対する書き込みと読み出しの要求が、同時に行われない限りデータの競合は生じない。また、このような状況が発生しないことは、スケジューリング問題において、因果性条件に1サイクルの時間余有を加えたことで保証されている。さらに、幾つかのDSPとDMAによるD.P.RAMへの書き込みが同時に起こらないことも、各DSPに対する適切なスケジューリングによって保証される。

最後に、共通データ・バスと共通アドレス・バスには、データや信号の逆流を防止するために、一方向性バッファ（Buffer）が設けてある。各バッファの開閉は、対応するDSPによるデータの書き込み要求によって行われ、DSPがD/A変換器とD.P.RAMに対して出力データ $y[k]$ を書き込む際に開放される。



図F. 1　ディジタル制御器のハードウェア構成

参考文献

[1] M. W. Spong and M. Vidyasagar：Robot Dynamics and Control, John Wiley & Sons (1986)

[2] 古田勝久：「ロボットの動的制御」, 計測と制御, vol.30, no.5, pp.376–382 (1991)

[3] 吉川恒夫：ロボット制御基礎論, コロナ社 (1988)

[4] 有本卓：ロボットの力学と制御, 朝倉書店 (1990)

[5] 笠原博徳:「ロボット制御における並列処理」, コンピュートロール, no.19, pp.97–103, コロナ社 (1987)

[6] J. Y. S. Luh, M. W. Waiker and R. P. C. Paul："On-Line Computational Scheme for Mechanical Manipulators," Trans. of the ASME J. of Dynamic Systems, Measurement, and Control, vol.102, pp.69–76 (1980)

[7] R. Nigam and C. S. George："A Multiprocessor-Based Controller for the Control of Mechanical Manipulators," IEEE J. of Robotics and Automation, vol.RA-1, no.4, pp.173–182 (1985)

[8] E. Binder and J. H. Herzog："Distributed Computer Architecture and Fast Parallel Algorithms in Real-Time Robot Control," IEEE Trans. on Systems, Man, and Cybernetics, vol.SMC-16, no.4, pp.543–549 (1986)

[9] K. Hashimoto and H. Kimura："A New Parallel Algorithm for Inverse Dynamics," Int. J. of Robotics Research, vol.8, no.1, pp.63–79 (1989)

[10] J. Y. S. Luh and C. S. Lin："Scheduling of Parallel Computation for a Computer-Controlled Mechanical Manipulator," IEEE Trans. on Systems, Man, and Cybernetics, vol.SCM-12, no.2, pp.214–234 (1982)

[11] H. Kasahara and S. Narita:"Parallel Processing of Robot-Arm Control Computation on a Multimicroprocessor System," IEEE J. of Robotics and Automation, vol.RA-1, no.2, pp.104–113 (1985)

[12] 笠原博徳：並列処理技術, 第1章, コロナ社 (1991)

[13] 安浦寛人：「並列計算機と並列計算モデル」, 情報処理, vol.33, no.9, pp.1024–1032 (1992)

[14] 茨木俊秀：組合せ最適化－分枝限定法を中心として－, 産業図書 (1983)

[15] 持田侑宏：「DSP の現状と動向」, 情報処理, vol.30, no.11, pp.1291–1299 (1989)

[16] 丸田力男, 西谷隆夫：ジグナルプロセッサとその応用, 昭晃堂 (1988)

[17] 小野定康, 石井六哉：シグナルプロセッサのソフトウェア, コロナ社 (1989)

[18] J. M. Hollerbach："A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity," IEEE Trans. on Systems, Man, and Cybernetics, vol.SMC-10, no.11, pp.730–736 (1980)

[19] 中村仁彦，横小路泰義，花房秀郎，吉川恒夫：「ロボットマニピュレータの運動学と動力学の統合化計算」，計測自動制御学会論文集，vol.23，no.5，pp.491–498 (1987)

[20] 川崎晴久，神崎一男：「マニユレータモデルにおける最小動力学パラメータと逆動力学計算法」，日本ロボット学会誌，vol.11，no.1，pp.100–110 (1993)

[21] 川崎晴久，望月進一，神崎一男：「マニピュレータのモデルベースド適応制御における効率的計算法と軌道制御実験」，計測自動制御学会論文集，vol.30，no.4，pp.435–442 (1994)

[22] D. C. H. Yang and S. W. Tzeng："Simplification and Linearization of Manipulator Dynamics by the Design of Inertia Distribution," Int. J. of Robotics Research, vol.5, no.3, pp.120–128 (1986)

[23] D. T. Horak："A Simplified Modeling and Computational Scheme for Manipulator Dynamics," Trans. of the ASME J. of Dynamic Systems, Measurement, and Control, vol.106, pp.350–353 (1984)

[24] 後藤英一，一松信，広田良吾：計算機による数式処理のすすめ（Bit 別冊），共立出版 (1986)

[25] 佐々木建昭，元吉文男，渡辺隼郎：数式処理システム，昭晃堂 (1986)

[26] 米澤明憲：「オブジェクト指向計算の現状と展望」，情報処理，vol.29，no.4，pp.290–294 (1988)

[27] J. Y. S. Luh, M. W. Walker and R. P. C. Paul："Resolved-Acceleration Control of Mechanical Manipulators," IEEE Trans. on Automatic Control, vol.AC-25, no.3, pp.468–474 (1980)

[28] 吉川恒夫：「ロボットの位置と力のハイブリッド制御」，日本ロボット学会誌，vol.3，no.6，pp.531–537 (1985)

[29] N. Hogan："Impedance Control: An Approach to Manipulation: Part-I–III," Trans. of the ASME J. of Dynamic Systems, Measurement, and Control, vol.107, pp.1–24 (1985)

[30] 大須賀公一：「非線形メカニカルシステムの適応制御」，計測自動制御学会論文集，vol.22，no.7，pp.38–44 (1986)

[31] 小菅一弘，古田勝久，横山竜昭：「ロボットの仮想内部モデル追従制御系－メカニカル・インピーダンス制御への応用－」，計測自動制御学会論文集，vol.24，no.1，pp.55–62 (1988)

[32] J. J. E. Slotine and W. Li："On the Adaptive Control of Robot Manipulators," Int. J. of Robotics Research, vol.6, no.3, pp.49–59 (1987)

[33] 吉川恒夫，井村順一，村井雅彦：「ロボットマニピュレータのロバストな軌道追従制御」，システム制御情報学会論文誌，vol.3，no.7，pp.218–225 (1990)

[34] D. Koditschek："Natural Motion for Robot Arms," Proc. of 23rd IEEE Conference on Decision and Control, pp.733–735 (1984)

[35] M. C. Leu and N. Hemati："Automated Symbolic Derivation of Dynamic Equation of Motion for Robotic Manipulators," Trans. ASME, no.108, pp.172–189 (1986)

[36] J. Koplik and M. C. Leu："Computer Generation of Robot Dynamics Equations and the Related Issues," J. of Robotic Systems, vol.3, no.3, pp.301–319 (1986)

[37] C. P. Neuman and J. J. Murray："Symbolically Efficient Formulations for Computational Robot Dynamics," J. of Robotic Systems, vol.4, no.6, pp.743–769 (1987)

[38] 斉藤制海，梅野孝治，阿部健一，桑原耕治，前川明寛：「REDUCE を用いたロボットアームの運動学および動力学方程式自動生成システム」，電気学会論文誌 C，vol.109，no.2，pp.89–96 (1989)

[39] J. F. Nethery and M. W. Spong："Robotica : A Mathematica Package for Robot Analysis," IEEE Robotics & Automation Magazine, vol.1, no.1, pp.13–20 (1994)

[40] 遠山茂樹：機械系のためのロボティクス，第 2 章，総合電子出版社 (1989)

[41] K. Tagawa, Y. Ohta and H. Haneda："Symbolic Approach to Implementation of Linearizing Compensator for Robotic Manipulators," Proc. of IEEE 17th Int. Conference on Industrial Electronics, Control, and Instrumentation, pp.1251–1256 (1991)

[42] 田川聖治，太田有三，藤原正和，羽根田博正：「ロボットの逆動力学計算に対する最適化システム」，計測自動制御学会論文集，vol.29，no.2，pp.220–226 (1993)

[43] M. A. Breuer："Generation of Optimal Code for Expressions via Factorization," Communications of the ACM, vol.12, no.6, pp.333–340 (1969)

[44] A. V. エイホ，J. D. ウルマン（土居範久・訳）：コンパイラ，培風館 (1986)

[45] 田川聖治，津田政彦，浪越孝宏，太田有三，羽根田博正：「最適化コンパイラによるロボット制御プログラムの開発」，日本ロボット学会誌，vol.13，no.8，pp.1199–1205 (1995)

[46] 制御系モデル C プログラム・ジェネレータ（μ-CPG）取扱説明書，エムティティ株式会社 (1994)

[47] 小野定康，藤井哲郎：「マイクロプロセッサシステム開発環境の動向」，電子情報通信学会誌，vol.76，no.7，pp.738–743 (1993)

[48] 前田浩一：「ロボットアームの動的モデルと同定」，日本ロボット学会誌，vol.7，no.2，pp.95–100 (1989)

[49] H. Mayeda, K. Yoshida and K. Osuka："Base Parameters of Manipulator Dynamic Models," IEEE Trans. on Robotics and Automation, vol.RA-6, no.3, pp.312-321 (1990)

[50] 杉本明：「オブジェクト指向によるソフトウェア部品化と再利用支援環境」，システムと制御，vol.31，no.9，pp.674–681 (1987)

[51] 本位田真一，山城明宏：「オブジェクト指向分析・設計」，情報処理，vol.35，no.5，pp.392–401 (1994)

[52] J. ランボー，M. ブラハ，W. プレメラニ，F. エディ，W. ローレンセン（羽生田栄一・訳）：オブジェクト指向方法論 OMT －モデル化と設計－，トッパン (1992)

[53] R. S. ウイナー，L. J. ピンソン（前川守・訳）：C++：オブジェクト指向プログラミング，トッパン (1989)

[54] A. V. エイホ，J. E. ホップクロフト，J. D. ウルマン（大野義夫・訳）：データ構造とアルゴリズム，培風館 (1987)

[55] K. Tagawa, T. Heishi, T. Matumoto, Y. Ohta and H. Haneda："Iterative Task Division Method for Multiprocessor Scheduling Problem," Proc. of IEEE 20th Int. Conference on Industrial Electronics, Control, and Instrumentation, pp.1773–1778 (1994)

[56] D. J. Kuck, Y. Muraoka and S. C. Chen："On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," IEEE Trans. on Computers, vol.C-21, no.12, pp.1293–1310 (1972)

[57] S. Lakshmivarahan and S. K. Dhall：Analysis and Design of Parallel Algorithms, Part 2, McGraw-Hill (1990)

[58] 村岡洋一：超並列処理コンパイラ，第3章，コロナ社 (1990)

[59] 茨木俊秀：「並列計算の虚と実」，オペレーションズ・リサーチ，vol.8，pp.359–363 (1992)

[60] 笠井琢美：計算量の理論，近代科学社 (1987)

[61] 有川節夫，宮野悟：オートマトンと計算可能性，第5章，培風館 (1986)

[62] 野下浩平，高岡忠雄，町田元：基本算法，第4章，岩波書店 (1985)

[63] A. V. エイホ，J. E. ホップクロフト，J. D. ウルマン（野崎昭弘，野下浩平・訳）：アルゴリズムの設計と解析 I・II，サイエンス社 (1977)

[64] S. A. Cook："The Complexity of Theorem Proving Procedures," Proc. of 3rd ACM Symposium on Theory of Computing, pp.151–158 (1971)

[65] M. R. Garey and D. S. Johnson：Computers and Intractability A Guide to the Theory of NP-Completeness, W. H. Freeman and Company (1976)

[66] K. B. Irani and K. W. Chen："Minimization of Interprocessor Communication for Parallel Computation," IEEE Trans. on Computers, vol.C-31, no.11, pp.1067–1075 (1982)

[67] 木村博昭，福島実：トランスピュータによる並列処理，海文堂 (1990)

[68] K. Tagawa, Y. Kanki, Ohta and H. Haneda："A Parallel Processing Scheme for Dynamic Control of Robotic Manipulators," Proc. of IEEE 19th Int. Conference on Industrial Electronics, Control, and Instrumentation, pp.1419–1424 (1993)

[69] 田川聖治，神吉良英，太田有三，羽根田博正：「ロボットの動的制御における計算の並列化アルゴリズムとオブジェクト指向による実現」，システム制御情報学会論文誌，vol.7，no.12，pp.505–511 (1994)

[70] R. L. Graham："Bounds on Multiprocessing Timing Anomalies," SIAM J. Applied Mathematics，vol.17，no.2，pp.416–429 (1969)

[71] K. Tagawa, T. Fukui, Y. Kanki, Y. Ohta and H. Haneda:"Optimal Code Generation for Serial and Parallel Processing of Robot-Arm Control Computation," Proc. of IEEE 21st Int. Conference on Industrial Electronics, Control, and Instrumentation，pp.1112–1117 (1995)

[72] 田川聖治，福居毅至，神吉良英，太田有三，羽根田博正：「ロボットの動的制御における計算の並列化問題に対する最適化および準最適化アルゴリズム」，日本ロボット学会誌，vol.14，no.6，pp.903–910 (1996)

[73] 茨木俊秀：アルゴリズムとデータ構造, 昭晃堂 (1991)

[74] 仙波一郎：組合せアルゴリズム，第5章，サイエンス社 (1991)

[75] R. J. Fateman："Optimal Code for Serial and Parallel Computation," Communications of the ACM，vol.12，no.12，pp.694–695 (1969)

[76] 恒川佳隆，志田純一，川又政征，樋口龍雄：「滞在時間がきわめて小さい状態空間ディジタルフィルタ用高並列VLSIアーキテクチャ」，計測自動制御学会論文誌，vol.27，no.9，pp.1034–1040 (1991)

[77] K. Tagawa, Y. Ohta and H. Haneda："Hybrid Simulator for Evaluating The Performance of Digital Control Systems," Proc. of IEEE 17th Int. Conference on Industrial Electronics, Control, and Instrumentation，pp.2323–2328 (1991)

[78] H. H. Lu, E. V. Lee and D. G. Messerschmitt:"Fast Recursive Filtering with Multiple Slow Processing Elements," IEEE Trans. on Circuit and Systems，vol.CAS-32，no.11，pp.1119–1129 (1985)

[79] 坂和正敏：非線形システムの最適化<一目的から多目的へ>，第7章，森北出版 (1986)

[80] 日本ロボット学会：ロボット制御に使い易いＤＳＰとは？(講習会テキスト)，日本ロボット学会 (1993)

[81] TMS320C25ユーザーズマニュアル，テキサスインスツルメンツ株式会社 (1988)

[82] 杉野暢彦，年清昭彦，渡部英二，西原明法：「ディジタル信号処理回路の計算順序決定とそのシグナルプロセッサ用コンパイラへの応用」，電子情報通信学会論文誌A，vol.J71-A，no.2，pp.327–335 (1988)

[83] H. Maki, Y. Ohta and H. Haneda："Optimal Configuration Design of Multi-DSP's for Digital Control Systems," Proc. of IEEE 15th Int. Conference on Industrial Electronics, Control, and Instrumentation，pp.494–499 (1989)

[84] K. Tagawa, Y. Ohta, H. Maki and H. Haneda:"Suboptimal Configuration Design of Multi-DSP's for Multi-Input Single-Output Digital Controller," Proc. of IEEE 16th Int. Conference on Industrial Electronics, Control, and Instrumentation, pp.351–356 (1990)

[85] 田川聖治，太田有三，牧秀隆，羽根田博正：「マルチ DSP システムによるスループット時間が最短であるディジタル制御器の実現」，電気学会論文誌 D，vol.113-D，no.1，pp.72–78 (1993)

[86] K. Tagawa, T. Mori, Y. Ohta and H. Haneda:"Design of a Multi-DSP System using TMS320C25 and Optimal Scheduling for Digital Controller," Proc. of IEEE 18th Int. Conference on Industrial Electronics, Control, and Instrumentation, pp.1391–1396 (1992)

[87] 田川聖治，太田有三，天野昌幸，羽根田博正：「ある種のマルチ DSP システムのスケジューリング問題に対する最適化アルゴリズム」，電気学会論文誌 C，vol.115-C，no.4，pp.597–603 (1995)

[88] 田川聖治，天野昌幸，太田有三，羽根田博正：「ある種のマルチ DSP システムにおける多目的最適化問題に対する分枝限定法の適用」，電気学会論文誌 C，vol.115-C，no.10，pp.1219–1220 (1995)

[89] 森崇：マルチ DSP システムに関する研究，神戸大学工学研究科修士論文 (1992)

関連発表論文

学術論文

[1] Kiyoharu Tagawa, Yuzo Ohta, Hidetaka Maki and Hiromasa Haneda : "Suboptimal Configuration Design of Multi-DSP's for Multi-Input Single-Output Digital Controller," Proceeding of the International Conference on Industrial Electronics, Control, and Instrumentation, pp.351–356 (1990)

[2] Kiyoharu Tagawa, Yuzo Ohta and Hiromasa Haneda : "Symbolic Approach to the Implementation of Linearizing Compensator for Robotic Manipulators," Proceeding of the International Conference on Industrial Electronics, Control, and Instrumentation, pp.1251–1256 (1991)

[3] Kiyoharu Tagawa, Takashi Mori, Yuzo Ohta and Hiromasa Haneda : "Design of a Multi-DSP System using TMS320C25 and Optimal Scheduling for Digital Controller," Proceeding of the International Conference on Industrial Electronics, Control, and Instrumentation, pp.1391–1396 (1992)

[4] Kiyoharu Tagawa, Yoshihide Kanki, Yuzo Ohta and Hiromasa Haneda : "A Parallel Processing Scheme for Dynamic Control of Robotic Manipulators," Proceeding of the International Conference on Industrial Electronics, Control, and Instrumentation, pp.1419–1424 (1993)

[5] 田川聖治，太田有三，牧秀隆，羽根田博正：「マルチ DSP システムによるスループット時間が最短であるディジタル制御器の実現」，電気学会論文誌 D，Vol.113-D，No.1，pp.72–78 (1993)

[6] 田川聖治，太田有三，藤原正和，羽根田博正：「ロボットの逆動力学計算に対する最適化システム」，計測自動制御学会論文集，Vol.29，No.2，pp.220–226 (1993)

[7] 田川聖治，神吉良英，太田有三，羽根田博正：「ロボットの動的制御における計算の並列化アルゴリズムとオブジェクト指向による実現」，システム制御情報学会論文誌，Vol.7，No.12，pp.505–511 (1994)

[8] 田川聖治，太田有三，天野昌幸，羽根田博正：「ある種のマルチ DSP システムのスケジューリング問題に対する最適化アルゴリズム」，電気学会論文誌 C，Vol.115-C，No.4，pp.597–603 (1995)

[9] Kiyoharu Tagawa, Tuyoshi Fukui, Yoshihide Kanki, Yuzo Ohta and Hiromasa Haneda :"Optimal Code Generation for Serial and Parallel Processing of Robot-Arm Control Computation," Proceeding of the International Conference on Industrial Electronics, Control, and Instrumentation, pp.1112–1117 (1995)

[10] 田川聖治，天野昌幸，太田有三，羽根田博正：「ある種のマルチＤＳＰシステムにおける多目的最適化問題に対する分枝限定法の適用」，電気学会論文誌 C，Vol.115-C，No.10，pp.1219–1220 (1995)

[11] 田川聖治，津田政彦，浪越孝宏，太田有三，羽根田博正：「最適化コンパイラによるロボット制御プログラムの開発」，日本ロボット学会誌，Vol.13，No.8，pp.1199–1205 (1995)

[12] 田川聖治，福居毅至，神吉良英，太田有三，羽根田博正：「ロボットの動的制御における計算の並列化問題に対する最適化および準最適化アルゴリズム」，日本ロボット学会誌，Vol.14，No.6，pp.903–910 (1996)

学術講演

[1] 牧秀隆，田川聖治，太田有三，羽根田博正：「マルチ DSP を用いたディジタル制御における並列処理手法に関する一考察」，第 3 回シグナル・システム・シンポジウム，pp.13–16 (1990)

[2] 田川聖治，太田有三，羽根田博正：「マルチ DSP システムにおけるスケジューリングに関する考察」，第 34 回システム制御情報学会研究発表講演会，pp.293–294 (1990)

[3] 田川聖治，太田有三，羽根田博正：「マルチ DSP システムにおけるスケジューリングに関する考察 II」，平成 2 年電気学会産業応用部門全国大会，pp.583–588 (1990)

[4] 田川聖治，藤原正和，太田有三，羽根田博正：「逆動力学計算の最適化システム」，第 36 回システム制御情報学会研究発表講演会，pp.589–590 (1992)

[5] 田川聖治，森崇，太田有三，羽根田博正：「マルチ DSP システムによる汎関数オブザーバの構成」，平成 4 年電気関係学会関西支部連合大会，G107 (1992)

[6] 神吉良英，田川聖治，太田有三，羽根田博正：「ロボット制御における計算の並列化アルゴリズム」，第 37 回システム制御情報学会研究発表講演会，pp.163–164 (1993)

[7] 神吉良英，田川聖治，太田有三，羽根田博正：「オブジェクト指向によるロボット制御における計算の並列化システム」，平成 5 年電気関係学会関西支部連合大会，G51 (1993)

[8] 津田政彦，田川聖治，太田有三，羽根田博正：「ロボット制御則の簡単化システム」，第 38 回システム制御情報学会研究発表講演会，pp.251–252 (1994)

[9] 天野昌幸，田川聖治，太田有三，羽根田博正：「分枝限定法によるマルチ DSP のスケジューリング手法」，第 38 回システム制御情報学会研究発表講演会，pp.455–456 (1994)

[10] 浪越孝宏，津田政彦，田川聖治，太田有三，羽根田博正：「ロボット制御則の最適化システム II」，第 39 回システム制御情報学会研究発表講演会，pp.359–360 (1995)

[11] 福居毅至，神吉良英，田川聖治，太田有三，羽根田博正：「ロボット制御則の並列化問題に対する分枝限定アルゴリズム」，第 39 回システム制御情報学会研究発表講演会，pp.177–178 (1995)