DiMP Users Manual

Yuichi Tazaki

目次

第 1 章

はじめに

DiMP は優先度つき拘束解決アルゴリズムにもとづく多自由度ロボットの軌道計画用 C++ ライブラリです．ロボットおよび作業環境を剛体と関節から構成されるマルチボディにより表現し，ロボットが行うべき作業をタスクとして与えることで，一連の運動学・動力学条件のもとでタスクを達成する軌道を計算することができます．また，個々のタスクに異なる優先度を設定することができ，自由度の高いタスク設計が可能です．

DiMP は筆者らの最近の研究成果にもとづいて開発されているため，その仕様は流動的です．比較的仕様が安定した部分から順次本マニュアルにおいて解説を加えていくつもりです．

第 2 章

インストール

2.1 ディレクトリ構成

DiMP のディレクトリ構成は以下のようになっています.

ディレクトリ名	説明
doc	ドキュメント類
lib	.lib ファイル
Base	Springhead Base モジュール
DiMP1	DiMP 旧バージョン
DiMP2	DiMP 現行バージョン
GL	OpenGL
Examples	デモプログラム

2.2 インストールとビルド

DiMP は Visual Studio 2010（以下 VS2010）による開発を想定しています．VS2010 上でソリューションファイル `DiMP10.sln` を開いて下さい．ビルドを実行すると DiMP ライブラリおよび各サンプルプロジェクトがビルドされ，実行可能な状態になります.

第 3 章

DiMP による動作計画

3.1　動作計画の流れ

DiMP ではロボットおよび作業環境をマルチボディとして表現します．マルチボディとは，複数の剛体が関節で連結され，運動学条件や動力学法則にしたがって運動する物理モデルです．マルチボディシミュレーションの技術は，仮想環境におけるロボットのテストや，物理的リアリティを求められるゲームなどにおいて用いられています．シミュレーションでは，ある時刻における系の状態と外部からの入力にもとづいて次の時刻の状態を決定するという計算手続きを繰り返し行うことで系の時間発展を求めていきます．これに対し，DiMP が対象とする軌道計画はシミュレーションとは異なり，与えられたタスク（作業）を達成する上で望ましいマルチボディの軌道を設計するという問題です．

DiMP による軌道計画の手順を図 3.1 で説明します．はじめに，ロボットや作業環境を構成するマルチボディおよび作業それ自体を表現するタスクを構成していきます．これには，剛体の作成，関節の作成，タスクの作成および時間軸の設定の 4 つを行うことになります．これらの作業は基本的に順不同で行って構いません．ただし未作成の剛体を関節で連結することはできないなどの個々の依存関係には注意が必要です．

次にグラフの初期化を行います．ここでいうグラフとはマルチボディにタスクと時間軸を加えた構造を指します．グラフの初期化を実行すると，内部的には動作計画に用いられる一連の変数や拘束条件が自動的に生成されます．その後，動作計画を実行する前に，必要に応じて細かな詳細設定を行います．これには個々の拘束条件の優先度設定などがあります．これが済めば実際に動作計画の計算を実行します．DiMP で用いられている動作計画アルゴリズムは変数の更新処理を繰り返し行う反復型アルゴリズムです．したがって，実際には一度の関数呼び出しで動作計画を行うのではなく，ユーザ自身が更新処理を必要な回数だけ繰り返し呼び出すことになります．

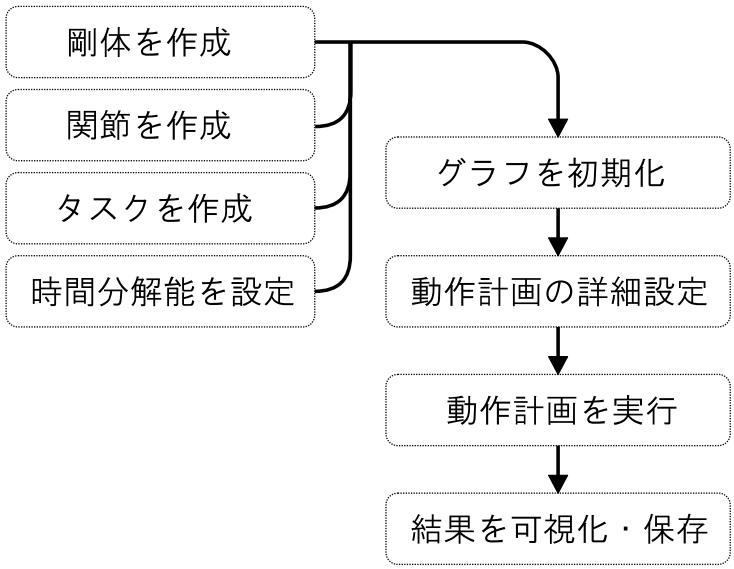

図 3.1　DiMP による動作計画の流れ

3.2　ライブラリの共通事項

3.2.1　ヘッダファイル

ユーザは `Graph.h` ヘッダファイルをインクルードすることで DiMP2 の全機能を使用できます．DiMP2 の `include` ディレクトリにパスを通した上で

```
#include <Graph.h>
```

としてください．

3.2.2　名前空間

DiMP のすべてのクラスは名前空間 `DiMP2` の中で定義されています．プログラムの冒頭で

```
using namespace DiMP2;
```

として名前空間をインポートしてもいいですが，DiMP2 の各クラスはどれも簡単な名前がつけられていますので他のライブラリとの名前の衝突の恐れがあります．このため，その都度 `DiMP2::Object` という風に修飾つきで参照する方が安全です．`typedef` によって

```
typedef DiMP2::Graph DGraph;
```

などのように短い別名を定義するのも良いでしょう．

3.2.3 クラスやメンバの扱い

簡便さと実行速度を優先するために，DiMP のクラスのメンバ関数やメンバ変数はすべて public になっています．したがって，内部を十分に理解しているユーザは任意の変数を好きに書き換えることができます．ただし，そうでないユーザは安全のために本マニュアルに記載されたメンバにのみアクセスするようにしてください．

3.2.4 変数の型

ライブラリでは，C++ の基本型に加えて以下の基本型が使用されます．

real_t	実数
vec3_t	3 次元ベクトル
vec4_t	4 次元ベクトル
quat_t	クォータニオン
pose_t	vec3_t と quat_t の複合型

実数型として float と double のどちらを使用するかはマクロで切り替えられます．精度と実行速度のどちらを優先すべきかを考慮して適切な方を選択してください．デフォルトでは double 型が使われるようになっています．float 型に切り替えるには

```
#define DIMP2_USE_SINGLE_PRECISION
#include <Graph.h>
```

としてください．

上の基本型は Springhead の Base モジュールが提供するクラスの別名です．よく使われるベクトル・行列演算は一通りサポートされていますので，詳細は Springhead のマニュアルを参照して下さい．

3.3 グラフ

グラフのクラスは DiMP2::Graph です．Graph クラスのインスタンスの生成方法は自由です．

```
Graph graph;
```

という風に変数として定義してもよいですし,

```
Graph* graph = new Graph;
```

というように動的確保しても構いません．ただしこの場合は delete を忘れないでください．また,

```
class MyGraph : public Graph{
    // ...
};
```

と継承して使うのも有効です．この場合は graph->などが省略できるのが利点です.

3.3.1　重力

重力は剛体に共通して作用する力ですので, その設定はシーングラフに対して行います.

Graph		
void	SetGravity(const vec3_t&)	重力加速度を設定

重力の初期値は $[0.0, -9.8, 0.0]$ です．動作計画に重力が不要な場合は

```
graph.SetGravity(vec3_t());
```

としてください.

3.4　剛体

剛体を表すのは Object クラスです．剛体を作成するには

```
Object* obj = graph->CreateObject();
```

とします（ただし graph は Graph のインスタンスとします).

3.4.1　物性の設定

剛体の物理的属性を設定するために以下のメンバ関数があります.

Object		
void	SetMass(real_t m)	質量を設定
void	SetInertia(real_t I)	慣性モーメントを設定
void	SetDynamical(bool on)	動力学フラグを設定

質量および慣性モーメントには正の値を設定してください．三次元剛体には慣性行列が対応しますが，DiMP では慣性行列が (inertia × 単位行列) という形をとることを想定しています．これは，物体形状を一定密度の球で近似することと等価です．質量と慣性モーメントの初期値はいずれも 1.0 です．

動力学フラグとは，その剛体が力の影響を受けて速度を変化するかどうかを設定するフラグです．外部の干渉を受けずに空間上に静止したり，等速直線運動する物体など作成するには

```
object->SetDynamical(false);
```

とします．動力学フラグの初期値は false です．

3.4.2 初期軌道の設定

DiMP は指定された様々な拘束条件を満足するマルチボディの軌道を反復的手続きによって計算します．このため，計画開始時点での各剛体の軌道である初期軌道の設定が結果に大きく影響します．デフォルトで，すべての剛体の初期軌道はワールド座標原点上で静止する軌道となっていますが，これを変更するには以下の関数を用います．

Object		
void	SetInitialPose (vec3_t p, quat_t q)	初期軌道における位置と向きを設定
void	SetInitialVelocity(vec3_t v, vec3_t w)	初期軌道における速度と角速度を設定

3.4.3 軌道を固定する

床などの空間上に静止した物体や，はじめから運動の軌道が決められた物体など，いくつかの剛体の軌道を動作計画の対象外としたい場合には，それらの剛体の軌道を固定します．

Object		
void	LockTrajectory()	初期軌道における位置と向きを設定

3.5　コネクタ

コネクタとは，剛体に対して後述する形状や関節を取り付ける位置と向きを指定するための要素です．コネクタを作成・削除するには以下の関数を使います．

Object		
Connector*	CreateConnector(const pose_t& p)	コネクタを作成
void	DeleteConnector(Connector* con)	コネクタを削除

コネクタを作成後に位置・向きを設定・取得するには以下の関数を使います．

Connector		
void	SetPose(const pose_t& p)	コネクタを作成
void	GetPose(pose_t& p)	コネクタを削除

以下に例を示します．

```
pose_t p;
p.Pos() = vec3_t(0,0,1);
p.Ori() = quat_t::Rot(Rad(30.0), 'x');
Connector* con = object->CreateConnector(p);
```

ここでは剛体の座標原点から $[0, 0, 1]$ の位置に，x 軸に関して 30 度傾いたコネクタが作成されます．

3.6　形状

剛体はそのままでは形状を持ちません．軌道を可視化する際や，干渉回避拘束を課すためには形状を作成して剛体に割り当てる必要があります．

形状を作成・削除するには Graph のメンバ関数を呼びます．

Graph

Box*	CreateBox(string_t, const vec3_t&)	直方体を作成
Sphere*	CreateSphere(string_t, real_t)	球を作成
Cylinder*	CreateCylinder(string_t, real_t, real_t)	円柱を作成
void	DeleteGeometry(Geometry* geo)	形状を削除

それぞれの形状の種類については後述します．作成済の形状を削除するには DeleteGeometry を呼びます．ここで引数の Geometry 型はすべての形状の基本型です．

例えば，一辺の長さが 1.0 の立方体を作るには

```
Box* box = graph->CreateBox("box1", vec3_t(1.0, 1.0, 1.0));
```

とします．

剛体に形状を割り当てたり，取り外したりするには以下の関数を用います．

Graph

void	AttachGeometry(Geometry* geo, Connector* con)	形状を取り付ける
void	DetachGeometry(Geometry* geo, Connector* con)	形状を取り外す

AttachGeometry の引数 geo には割り当てる形状，con は形状を取り付けたいコネクタを指定します．DetachGeometry は con に取り付け済の形状 geo を取り外します．

同じ形状を複数の異なるコネクタに取り付けることが可能です．

```
graph->AddGeometry(box, con1);
graph->AddGeometry(box, con2);
```

こうすると，見かけ上は形状が複製されたようになりますが，内部的には形状の実体は一つです．したがって，box に加えた変更は取り付けられたすべてのコネクタに影響します．

3.6.1 直方体

Graph

Box*	CreateBox(string_t name, const vec3_t& sz)

コネクタの原点を中心とし，各軸に平行な直方体を作成します．sz の各成分はそれぞれ直方体の x，y，z 軸に平行な辺の長さを指定します．

3.6.2　球

Graph	
Sphere*	CreateSphere(string_t name, real_t radius)

コネクタの原点を中心とし，半径 radius の球を作成します．

3.6.3　円柱

Graph	
Cylinder*	CreateCylinder(string_t name, real_t radius, real_t length)

コネクタの原点を中心とし，長さ方向が z 軸に一致する円柱を作成します．radius は円柱のふたの半径，length は円柱の長さです．

3.7　関節

関節は剛体同士の相対運動を拘束する要素です．剛体を関節で連結するには，まず連結したい剛体と，関節を取り付ける位置と向きを決めるコネクタを作成します．その上で以下の関数を用いて関節を作成・削除します．

Graph		
Hinge*	CreateHinge (string_t, Connector*, Connector*)	ヒンジを作成
Slider*	CreateSlider (string_t, Connector*, Connector*)	スライダを作成
Balljoint*	CreateBalljoint (string_t, Connector*, Connector*)	ボールジョイントを作成
Planejoint*	CreatePlanejoint(string_t, Connector*, Connector*)	平面ジョイントを作成
void	DeleteJoint (Joint* jnt)	関節を削除

DiMP で作成できる関節の種類には後述するヒンジ，スライダ，ボールジョイント，平面ジョイントがあります．一度作成した関節を削除するには DeleteJoint を呼びます．引数の Joint 型はすべての関節種類の基本型です．

例として，剛体をヒンジで連結するには以下のようにします．

```
Hinge* hinge = graph->CreateHinge("hinge1", socket, plug);
```

ただし，socket，plug は連結したい二つの剛体のそれぞれに対して作成されたコネクタです．関節は，二つのコネクタ（ソケットとプラグ）の相対運動を所定の方向に限定することで，結果としてコネクタの取り付けらえた剛体同士の運動を制限します．

3.7.1 ヒンジ

ヒンジは 1 自由度回転関節です．ヒンジは，ソケットとプラグの原点を一致させ，かつ両者の z 軸の向きを一致させます．結果として，剛体の相対運動はコネクタの z 軸に関する回転運動となります．

自由度	
0	ソケット x 軸に対するプラグ x 軸の角度 [rad]

3.7.2 スライダ

スライダは 1 自由度直動関節です．スライダは，ソケットとプラグの向きを一致させ，かつプラグの原点がソケットの z 軸上にあるように拘束します．結果として，剛体の相対運動はコネクタの z 軸に関する直動運動となります．

自由度	
0	ソケットに対するプラグ原点の z 座標

3.7.3 ボールジョイント

ボールジョイントは 3 自由度回転関節です．ボールジョイントは，ソケットとプラグの原点を一致させます．また，ソケットとプラグの相対的な向きは三つの回転角（ヨー角，ピッチ角，ロール角）によって表されます．

自由度	
0	ヨー角 [rad]
1	ピッチ角 [rad]
2	ロール角 [rad]

3.7.4　平面ジョイント

平面ジョイントは 2 自由度直動，1 自由度回転関節です．平面ジョイントは，プラグの原点をソケットの xy 平面上に拘束し，かつ両者の z 軸の向きを一致させます．結果として，剛体はソケットとプラグの xy 平面同士が接するように相対運動します．

自由度	
0	ソケットに対するプラグ原点の x 座標
1	ソケットに対するプラグ原点の y 座標
2	ソケット x 軸に対するプラグ x 軸の角度 [rad]

3.7.5　初期値の設定

動作計画開始時の関節の変位や速度はデフォルトで 0 ですが，異なる値に設定するには以下の関数を使います．

Joint		
void	SetInitialPos(uint i, real_t pos)	初期関節変位を設定
void	SetInitialVel(uint i, real_t vel)	初期関節速度を設定

SetInitialPos は，関節の i 番目の自由度に対応する変位の初期値を pos に設定します．SetInitialVel は同様に i 番目の自由度の速度の初期値を vel に設定します．

3.7.6　範囲拘束の設定

関節の変位や速度，トルクに関して範囲拘束を課すことができます．

Joint		
void	SetPosRange (uint i, real_t rmin, real_t rmax)	関節変位の範囲を設定
void	SetVelRange (uint i, real_t rmin, real_t rmax)	関節速度の範囲を設定
void	SetTorqueRange(uint i, real_t rmin, real_t rmax)	関節トルクの範囲を設定

各関数について，i は拘束を課したい自由度，rmin は最小値，rmax は最大値です．

3.7.7 関節軌道の取得

動作計画の途中あるいは完了後に関節の軌道を取得する方法については 3.11.1 節を参照してください．

3.7.8 順機構学計算

動作計画において，軌道の始点は定数として扱われます．このため，初期時刻（軌道始点）における各剛体の位置や向きは，関節による連結を考慮して適切に設定されていなければなりません．これを手動で行うことも可能ですが，以下の関数で順運動学計算を行うのが便利です．

Object		
void	ForwardKinematics()	順運動学計算により位置と向きを決定

ForwardKinemaics は，呼び出した剛体を基準とし，関節で連結された剛体の位置と向きを順運動学計算により決定します．

3.8 時間

時間の設定は，動作計画において考える時間区間（タイムライン）の長さや，剛体や関節の軌道を表現するスプライン曲線の自由度を決定するために行います．

時間の構成要素をティックと呼びます．ティックはタイムライン上の一つの時刻を表します．タイムラインの始点は必ず時刻 0 です．時刻 0 から最大の時刻を持つティックまでの範囲がタイムラインとなります．また，その間に多くのティックを配置するほど，軌道の表現自由度が向上しますが，同時に計算コストが増大します．

時間およびティックに関する関数を以下に示します.

Graph

Tick*	AddTick(real_t)	ティックを追加
void	AddTicks(real_t ts, real_t dt, real_t te)	
void	AddTicks(real_t* tbegin, real_t* tend)	
void	RemoveTick(Tick*)	ティックを削除
void	ClearTicks()	
size_t	NTicks()	ティックの個数
Tick*	GetTick(uint i)	ティックを取得
real_t	GetHorizonLength()	タイムラインの長さを取得

AddTick は指定した時刻 t にティックを一つ作成します．まとめて複数のティックを作成するには AddTicks を使います．一つ目の AddTicks は始点を ts，終点を te とする区間に dt 間隔でティックを配置します．また，二つ目の AddTicks は配列 [tbegin, tend) に格納された数値を時刻としてティックを作成します.

ティックを作成するには以下のようにします.

```
graph->AddTick(0.0);
graph->AddTick(1.0);
graph->AddTick(2.0);
graph->AddTick(3.0);
```

この例ではタイムラインは $[0, 3]$ の 3 秒間となり，その間に 1 秒間隔で 4 つのティックが置かれます．この場合，シーングラフ中の剛体および関節の軌道は，3 つの区間から成るスプライン曲線で表現されることになります．ティックをタイムライン上に等間隔に配置する必要はありません．作成済のティックの時刻を取得・変更するには以下を用います.

Tick

void	SetTime(real_t)	時刻を設定
real_t	GetTime()	時刻を取得

```
graph->GetTick(2)->SetTime(2.1);
```

何らかのタスクを行う動作計画を行う際，タイムラインをどの程度の長さにし，またタ

イムライン上にいくつのティックを配置するかはとても重要で難しい問題です．今のところ，DiMP はこの決定をユーザにゆだねています．

3.9 タスク

タスクは動作計画の作業内容を表現する要素です．一般にタスク（作業）というと，例えば「街に出て買い物をして帰ってくる」とか，「キッチンで料理をする」など，いくらでも複雑で高度なものが考えられますが，DiMP が今のところ対象とするタスクは最も基本的なものです．つまり，DiMP におけるタスクとは「マルチボディの異なる二つの構成要素の状態量を所定の時区間で一致させる」ことを指します．例えば，ロボットアームの手先を表す剛体と，箱を表す剛体の位置を適当な時刻に一致させるタスクを考えることでロボットアームが箱へ向かって手先を伸ばす動作を計画することができます．

タスクを作成するには，それに関連付けるためのタイムスロットを作成する必要があります．タイムスロットとは，タスクを実行する時間区間を表します．先ほどの例で言うと，ロボットアームと箱の位置が一致すべき時間区間です．ここで，「ロボットアームが手先を箱へ向けて移動させる」といった，タスクを達成するための予備動作はタイムスロットに含まれません．このような動作は DiMP が自動的に生成します．

Graph

`TimeSlot*`	`CreateTimeSlot(string_t name, real_t ts = 0.0, real_t te = 0.0)`	タイムスロットを作成

タイムスロットを作成するには次のようにします．

```
TimeSlot* timeslot = graph->CreateTimeSlot("timeslot1", 1.0, 2.0);
```

`CreateTimeSlot` には作成するタイムスロットの名前を任意に指定し，タイムスロットの始点時刻と終点時刻を指定します．ここでタイムスロットの範囲は $[1.0, 2.0]$ となります．タイムスロットの関数を以下に示します．

TimeSlot

void	Set(real_t ts, real_t te)	時刻の初期値
void	Lock(bool on = true)	時刻を固定する
void	SetStartRange(real_t ts_min, real_t ts_max)	始点時刻の範囲拘束
void	SetEndRange(real_t te_min, real_t te_max)	終点時刻の範囲拘束
void	SetDurationRange(real_t T_min, real_t T_max)	時間の範囲拘束

デフォルトではタイムスロットの始点・終点時刻は固定されておらず，動作計画の結果によって異なる値を取り得る状態になっています．したがって CreateTimeSlot や Set で指定する値はあくまで計画開始時の初期値にすぎません．これは，言ってみれば「いつでも良いのでロボットアームの手先を箱に一致させてほしい」ということになります．タスクの達成時間をこちらで明示したい場合，タイムスロットの時刻を固定します．

```
timeslot->Lock();
```

これで，動作計画中にタイムスロットの範囲は定数として扱われます．固定を解除したい場合は Lock(false) とします．また，固定はしたくないがタイミングの範囲を拘束したい場合もあるでしょう．その場合は Set[Start|End|Duration]Range を使います．これらは，それぞれ始点時刻，終点時刻および時間（終点と始点の差分）に関して範囲拘束を課します．

次にいよいよタスクを作成します．DiMP でサポートしているタスクはマッチングタスクと干渉回避タスクの二種類です．

Graph

MatchingTask*	CreateMatchingTask(string_t name, Object* obj0, Object* obj1, TimeSlot* time = 0)	マッチングタスクを作成
AvoidTask*	CreateAvoidTask(string_t name, Object* obj0, Object* obj1, TimeSlot* time = 0)	干渉回避タスクを作成

いずれも，name はタスクの名前，obj0 と obj1 はタスクを課す剛体，time はタスクに

関連付けるタイムスロットです．time を 0 (null) とした場合はタイムライン全区間がタイムスロットとなります．

以下に例を示します．

```
MatchingTask* task;
task = graph->CreateMatchingTask("task1", hand, box, timeslot);
```

上の例で hand と box は事前に作成しておいた剛体を保持する Object*型変数とします．

マッチングタスク

マッチングタスクは指定した二つの剛体の位置を，タイムスロットが示す時区間において一致させる拘束を課します．

干渉回避タスク

干渉回避タスクは，指定した二つの剛体が，タイムスロットが示す時区間において干渉しないようにするタスクです．具体的には，それぞれの剛体に割り当てられた形状にもとづいて剛体の座標原点を中心とする外接球の半径を算出し，剛体間の距離が外接球半径の和よりも大きくなるように拘束します．

外接球にもとづく干渉判定ですので，細長い形状などに対してはかなり保守的な干渉回避となります．

3.10 動作計画の詳細設定と実行

前節までで，マルチボディを構成する剛体と関節を作成し，タイムラインとタスクによって動作計画の仕様を指定する方法を説明しました．本節では図 3.1 におけるそれ以降の手順について説明します．

3.10.1 グラフの初期化

まず，グラフの初期化を実行します．

Graph		
void	Init()	グラフを初期化

Init 関数は，ユーザによって作成されたマルチボディ，タイムラインおよびタスクをもとに，動作計画で用いられる内部構造を初期化します．一度初期化を行った後に，剛体

表 3.1　変数ラベル一覧

ラベル	変数の型	説明
ObjectTP	vec3_t	剛体の位置
ObjectRP	quat_t	剛体の向き
ObjectTV	vec3_t	剛体の速度
ObjectRV	vec3_t	剛体の角速度
JointP	real_t	関節の変位
JointV	real_t	関節の速度
ForceT	vec3_t	関節の並進力
ForceR	vec3_t	関節のモーメント
TimeStart	real_t	タイムスロット始点
TimeEnd	real_t	タイムスロット終点

の追加などの編集が行われたときは Init を再度呼ぶ必要があります.

3.10.2　詳細設定

以下では，動作計画の前に行う詳細設定の方法について項目別に説明します.

変数や拘束の参照方法

DiMP は動作計画に関係する変数や拘束を内部で保持しており，詳細設定を行う際にはそれらの変数や拘束を指定する必要があります．変数や拘束は，「ラベル」，「ティック」，「ノード」の 3 つの組み合わせで一意に特定することができます．これを ID と呼び，ID クラスで表されます．ここでノードとは剛体，関節およびタスクの総称です．変数のラベルは VarTag 列挙型，拘束のラベルは ConTag 列挙型にそれぞれ保持されます．また，変数ラベルと拘束ラベルの一覧をそれぞれ表 3.1，表 3.2 に示します.

ID を作成するには ID クラスのコンストラクタを利用します.

ID

```
ID(int tag, Node* node = 0, Tick* tick = 0)
```

tag には上で述べたラベルのひとつを指定します．node には指定したい変数や拘束を保有するノードを与えます．tick には変数や拘束が対応する時刻を与えます．node や

表 3.2 拘束ラベル一覧

ラベル	拘束変数の型	説明
ObjectC1T	vec3_t	剛体軌道の C^1 連続性 (並進成分)
ObjectC1R	quat_t	剛体軌道の C^1 連続性 (回転成分)
JointC1	real_t	関節軌道の C^1 連続性
JointRangeP	real_t	関節変位の範囲拘束
JointRangeV	real_t	関節速度の範囲拘束
JointRangeF	real_t	関節トルクの範囲拘束
JointTP	vec3_t	関節の機構的拘束 (位置・並進成分)
JointRP	quat_t	関節の機構的拘束 (位置・回転成分)
JointTV	vec3_t	関節の機構的拘束 (速度・並進成分)
JointRV	vec3_t	関節の機構的拘束 (速度・回転成分)
ForceT	real_t	剛体の力のつり合い (並進成分)
ForceR	real_t	剛体の力のつり合い (回転成分)
TimeStartRange	real_t	タイムスロット始点の範囲拘束
TimeEndRange	real_t	タイムスロット終点の範囲拘束
TimeDurationRange	real_t	タイムスロット区間長の範囲拘束
MatchingTP	vec3_t	マッチングタスク（位置・並進成分）
MatchingRP	quat_t	マッチングタスク（位置・回転成分）
MatchingTV	vec3_t	マッチングタスク（速度・並進成分）
MatchingRV	vec3_t	マッチングタスク（速度・回転成分）
AvoidP	vec3_t	干渉回避タスク（位置）
AvoidV	vec3_t	干渉回避タスク（速度）

tick に 0 を指定すると，ノードや時刻に関する条件づけは行われません．例えば

```
ID id(VarTag::ObjectTP);
```

とすると，全剛体の全時刻に関する位置変数から成る変数の集合を指定する ID が得られます．

変数のロック

変数をロックすることでその変数を定数として扱うことができます．

Graph

```
void    Lock(ID id, bool lock = true)
```

id は変数 ID，lock はロックするか，あるいはロック解除するかを指定するフラグです．

拘束の無効化

拘束を無効化すると，動作計画においてその拘束が考慮されなくなります．

Graph

```
void    Enable(ID id, bool enable = true)
```

id は拘束 ID です．enable が true の場合は id に合致する拘束を有効化し，enable が false の場合はこれを無効化します．

優先順位の設定

拘束には優先順位を設定できます．もし変数の自由度が足らずにすべての拘束を満足する解が見つからない場合，より優先度の高い拘束のみを満足し，より優先度の低い拘束に関してはその誤差を最小化するような解が選ばれます．

Graph

```
void    SetPriority(ID id, int level)
```

id は拘束 ID，level は優先度レベルを指定します．優先度レベルは 0 が最高優先度で，値が増えるにつれて優先度が低くなります．ただしレベルの絶対的な値には意味はありません．例えば，レベルを $0, 1, 2$ と分けても $0, 3, 10$ と分けても解は変わりません．

アルゴリズムの調整

Graph

```
void    SetNumIteration(int n)
```

Graph::Step の内部処理では変数の変化方法を決定するために大規模な連立方程式が計算されます．このために DiMP では Gauss-Seidel 法が採用されています．SetNumIteration 関数は Gauss-Seidel 法の反復回数を設定します．一般的に，反復回数を大きくするほど動作計画を完了するために実行すべき Step の回数は少なくて済み

ますが，逆に一度の Step にかかる計算時間は反復回数に比例して増大します．

3.10.3 動作計画の実行

DiMP が提供する動作計画アルゴリズムは反復解法です．したがって，アルゴリズムを一度に実行して最終的な解を出力するのではなく，軌道の更新を繰り返すことによって次第に望ましい軌道に近づけていきます．この一度の更新を実行するのが Graph::Step 関数です．

Graph		
void	Step()	一度の更新処理

DiMP では，最終的に満足のいく軌道を得るまでに何回 Step を呼び出すかの決定をユーザにゆだねています．リアルタイム性が要求される状況では Step を定数回呼び出すのが良いでしょう．そうすれば概ね定数時間で計算が終了します．逆に精度を優先したい場合は，単に Step の呼び出し回数を増やしてもいいですが，変数値や拘束誤差の変化をモニタして適当な打ち切り条件で終了することも可能です．

Graph		
real_t	CalcError(ID mask, bool sum_or_max)	拘束誤差を取得
real_t	CalcVariable(ID mask, bool ave_or_max)	変数値を取得

CalcError 関数は動作計画中の拘束誤差値を取得します．ID は拘束 ID，sum_or_max は true ならば誤差の総和を，false ならば誤差の最大値を返します．拘束に対して変数の自由度が不足している状況では，平衡状態でも拘束誤差が 0 になるとは限らないので注意が必要です．

CalcVariable 関数は動作計画中の変数値を取得します．ID は変数 ID，ave_or_max は true ならば変数の絶対値の平均を，false ならば絶対値の最大値を返します．これらの関数はあくまで全変数，全拘束の統計情報を取得するためのものですので，個別の変数や拘束の情報が取得したい場合は後述の方法を用いてください．

3.11 結果の取得

3.11.1 軌道の取得

動作計画後のマルチボディの軌道を取得するための簡易機能がいくつか用意されています．以下の一連の関数は，タイムライン上のある時刻における剛体の状態を取得するのに用います．

表 3.3　補間方法一覧

ラベル	説明
Constant	区間始点の位置で固定，速度，加速度は 0
LinearDiff	区間始点と終点の位置を線形補間
LinearInt	区間始点と位置と速度で補間
Quadratic	区間始点の位置と速度，終点の位置で二次補間
Cubic	区間始点・終点の位置・速度で三次補間
SlerpDiff	区間始点と終点の回転で球面線形補間 (quat_t 用)
SlerpInt	区間始点の回転と角速度で球面線形補間 (quat_t 用)

Object

vec3_t	Pos(real_t t, int type)	位置（並進成分）を取得
vec3_t	Vel(real_t t, int type)	速度（並進成分）を取得
vec3_t	Acc(real_t t, int type)	加速度（並進成分）を取得
quat_t	Ori(real_t t, int type)	位置（回転成分）を取得
vec3_t	Angvel(real_t t, int type)	速度（回転成分）を取得
vec3_t	Angacc(real_t t, int type)	加速度（回転成分）を取得

引数 t には時刻を指定します．また，type には軌道の補間方法を指定します．DiMP は剛体や関節の軌道を，ティックの配置された時刻における状態の系列として保持します．したがって，ティックの置かれていない中間時刻の状態は，何らかの補間計算によって求める必要があります．type に指定できる補間方法は Interpolate 列挙型にまとめられています（表 3.3）．quat_t 型とそれ以外で指定可能な補間ラベルが異なるので注意して下さい．なお，type は省略可能で，その場合 quat_t は SlerpDiff，それ以外は Quadratic がデフォルトとなります．

また，以下の関数は同様に関節の状態を取得するのに用います．

Joint

real_t	Pos(uint i, real_t t, int type)	変位を取得
real_t	Vel(uint i, real_t t, int type)	速度を取得
real_t	Acc(uint i, real_t t, int type)	加速度を取得

引数 i には取得する関節自由度のインデックスを指定します．ヒンジなどの 1 自由度関節であれば 0 を指定します．他の引数については剛体の場合と同様です．

3.11.2 軌道の可視化

DiMP には計画された軌道を可視化するための簡易描画機能が用意されています．描画には Springhead ライブラリの Graphics モジュールが使われますので，詳細については Springhead のマニュアルも同時に参照して下さい．

T.B.D.

索引

参考文献

[1] 田崎, 鈴木, 鈴木: “マルチボディシステムの拘束ベース多目的軌道計画”, 日本ロボット学会誌, Vol.31, No.5.