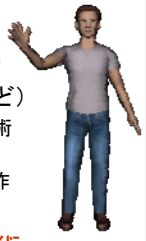
コンピュータグラフィックス特論 II

第11回　キャラクタアニメーション（2）

九州工業大学　尾下 真樹

キャラクタ・アニメーション

- CGにより表現されたキャラクタのアニメーションを実現するための技術
- キャラクタ・アニメーションの用途
  - オフライン・アニメーション（映画など）
  - オンライン・アニメーション（ゲームなど）
    - どちらの用途でも使われる基本的な技術は同じ（データ量や詳細度が異なる）
    - 後者の用途では、インタラクティブな動作実現のための工夫が必要になる
- 人体モデル・動作データの処理技術



全体の内容

- キャラクタ・アニメーションの基礎
- 骨格モデル・姿勢・動作の表現
- 動作データの作成
- 運動学
- 姿勢・動作ブレンディング
- 応用技術

今日の内容

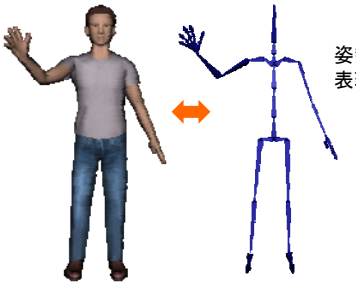
- 前回の復習
- サンプルプログラム
- 運動学
  - 順運動学
  - 逆運動学
- レポート課題

前回の復習

人体モデルの表現

形状モデル（ポリゴンモデル）　骨格モデル（多関節体）

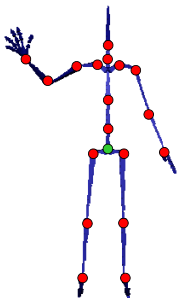
描画用



姿勢・動作の表現・処理用

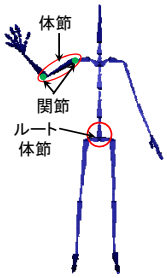
骨格モデルの表現

- 多関節体モデルによる表現
  - 複数の体節(リンク)が関節で接続されたモデル
  - 体節
    - 複数の関節が接続されており、体節の長さや体節内での関節の接続位置は固定
  - 関節
    - 2つの体節の間を接続
    - 関節の回転により姿勢が変化する



骨格・姿勢の表現方法

- 骨格情報と姿勢情報を分ける
- 骨格情報の中で、関節・体節を分ける
  - 体節
    - 複数の関節と接続
    - 各関節の接続位置
      - 体節のローカル座標系
  - 関節
    - 2つの体節の間を接続
      - ルート側・末端側の体節



骨格モデルの表現方法

- 骨格情報の中で、体節と関節を分ける

```
// 人体モデルの体節を表す構造体
struct  Segment
{
    // 接続関節
    vector< Joint * >      joints;
    // 各関節の接続位置(体節のローカル座標系)
    vector< Point3f >      joint_positions;
};

// 人体モデルの関節を表す構造体
struct  Joint
{
    // 接続体節
    Segment *              segments[ 2 ];
};
```

```
// 人体モデルの骨格を表す構造体
struct  Skeleton
{
    // 体節・関節の配列
    vector< Segment * >  segments;
    vector< Joint * >       joints;
};
```

姿勢の表現方法

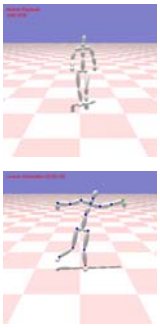
- 骨格情報と姿勢情報を分ける

```
// 人体モデルの姿勢を表す構造体
struct  Posture
{
    Skeleton *  body;
    Point3f     root_pos;        // ルートの位置
    Matrix3f    root_ori;        // ルートの向き(回転行列表現)
    Matrix3f *  joint_rotations; // 各関節の回転(回転行列表現)
                                 // [リンク番号] リンク数分の配列
};
```

サンプルプログラム

デモプログラム

- 複数のデモを含むプログラム
  - マウスの中ボタン or m キーで切り替え
- 動作再生
- キーフレーム動作再生
- 順運動学計算
- 逆運動学(CCD-IK)
- 姿勢補間
- 動作補間(2つの動作の補間)
- 動作接続・遷移



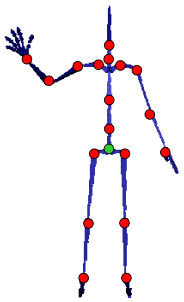
サンプルプログラム

- デモプログラムの一部（human_sample.cpp）
  - 骨格・姿勢のデータ構造定義（SimpleHumn.h/cpp）
  - BVH動作クラス（BVH.h/cpp）
  - アプリケーションの基底クラス（GLUTBaseApp.h/cpp）
    - 各イベント処理のためのメソッドの定義を含む
    - 本クラスを派生させて、各アプリケーションクラス定義
  - メイン処理、コールバック関数（GLUTBaseApp.cpp）
    - 全アプリケーションを管理、切替
    - 現在のアプリケーションのイベント処理を呼び出し
  - 各デモの主要な処理は各自で実装

骨格・姿勢・動作のデータ構造

- 骨格・姿勢の構造体定義（SimpleHumn.h/cpp）

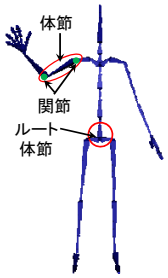
```
// 多関節体の体節を表す構造体
struct Segment

// 多関節体の関節を表す構造体
struct Joint

// 多関節体の骨格を表す構造体
struct Skeleton

// 多関節体の姿勢を表す構造体
struct Posture
```



- BVH動作クラス（BVH.h/cpp）

骨格・姿勢の表現方法

- 骨格情報と姿勢情報を分ける
- 骨格情報の中で、関節・体節を分ける
  - 体節
    - 複数の関節と接続
    - 各関節の接続位置
      - 体節のローカル座標系
  - 関節
    - 2つの体節の間を接続
      - ルート側・末端側の体節



骨格モデルの表現方法

```
// 人体モデルの体節を表す構造体
struct Segment
{
   // 接続関節
   vector< Joint * >     joints;
   // 各関節の接続位置（体節のローカル座標系）
   vector< Point3f >     joint_positions;
};

// 人体モデルの関節を表す構造体
struct Joint
{
   // 接続体節
   Segment *             segments[ 2 ];
};

// 人体モデルの骨格を表す構造体
struct Skeleton
{
   // 体節・関節の配列
   int                   num_segments;
   Segment **            segments;
   int                   num_joints;
   Joint **              joints;
};
```

複数の関節と接続
ルート体節以外は、0 番目の関節が、ルート側の関節とする

2つの体節の間を接続
0 番目の体節が、ルート側の体節とする

※ IK計算などで、体節・関節を順番に辿りながら処理をするときのために、ルートがどちら側を判断するためのルールや情報があった方が良い

姿勢の表現方法

```
// 人体モデルの姿勢を表す構造体
struct Posture
{
   Skeleton *  body;
   Point3f     root_pos;       // ルートの位置
   Matrix3f    root_ori;       // ルートの向き（回転行列表現）
   Matrix3f *  link_rotations; // 各リンクの相対回転（回転行列表現）
                               // [リンク番号] リンク数分の配列
};
```

動作の表現方法

```
// 人体モデルの動作を表す構造体
struct Motion
{
   // 骨格モデル
   Skeleton *          body;
   // フレーム数
   int                 num_frames;
   // フレーム間の時間間隔
   float               interval;
   // 全フレームの姿勢 [フレーム番号]
   Posture *           frames;

   // 姿勢を取得
   void  GetPosture( float time, Posture & p ) const;
};
```

骨格・動作の読み込み

- 骨格モデルや動作データが必要
- BVH形式の動作データを読み込み、骨格モデル・動作データに変換する
  - BVH動作のクラス(BVH)や読み込み処理は、前回説明した通り
  - BHVクラスのままでは扱いづらいため、変換

```
// BVH動作から骨格モデルを生成
Skeleton *  CoustructBVHSkeleton( class BVH * bvh );

// BVH動作からデータ(＋骨格モデル)を生成
Motion *  CoustructBVHMotion( class BVH * bvh, Skeleton * b );
```

描画処理

- 姿勢描画

```
// 姿勢の描画(スティックフィギュアで描画)
void  DrawPosture( const Posture & posture );

// 姿勢の影の描画(スティックフィギュアで描画)
void  DrawPostureShadow( const Posture & posture,
        const Vector3f & light_dir, const Color4f & color );
```

  - 内部で順運動学計算を呼び出し

アプリケーションの基底クラス

- GLUTBaseApp

```
class  GLUTBaseApp
{
  protected:
    // 視点操作のための変数
    // マウス入力処理のための変数
    // アプリケーション状態の変数
  public:
    // イベント処理インターフェース
    virtual void  Initilize();
    virtual void  Start();
    virtual void  Display();
    ・・・
    virtual void  Animation( float delta );
};
```

GLUTメイン処理

- 複数のアプリケーションを管理・切り替え

```
// 全アプリケーションのリスト
vector< GLUTBaseApp * >    applications;

// 現在実行中のアプリケーション
GLUTBaseApp *    app = NULL;
```

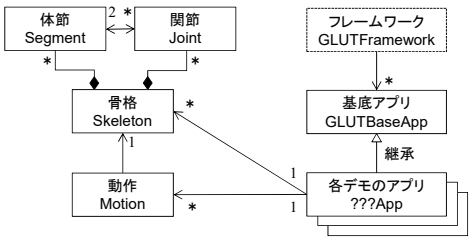
上のリストの中の、現在実行中のアプリケーションを表す

- GLUTコールバック関数から、現在のアプリケーションのイベント処理を呼び出し

```
//void  DisplayCallback( void )
{
   app->Display();
   ・・・
}
```

クラス図

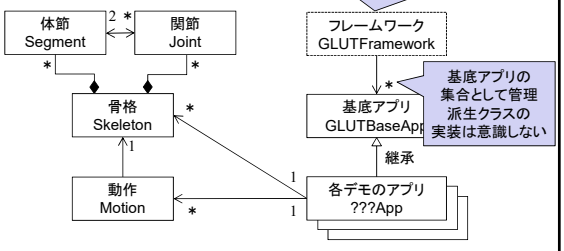
- クラス・構造体間の関係



クラス図

- クラス・構造体間の関係

グローバル関数の集まりで構成されるので、クラスではないが、ここでは一つのクラスと同様に記述



動作再生アプリケーション

- MotionPlaybackApp
  - BVH動作を読み込んで再生
    - Sキーで、一時停止・再生
    - 一時停止中にP・Nキーで、前・次のフレーム
    - Wキーで、再生速度を変更
    - Lキーを押すと、読み込むBVHファイルを選択
      - 最初に自動的に読み込むBVHファイルは、プログラム中で指定

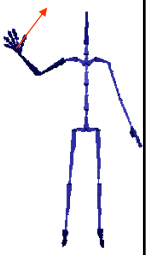
動作再生アプリケーション

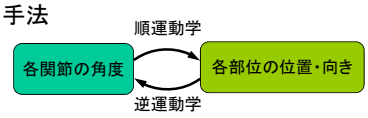
- MotionPlaybackApp（実装済み）
  - GLUTBaseAppを派生
  - 初期化処理（Initialize関数）で、骨格や動作を読み込み（BVH動作を読み込んで変換）
  - 開始処理（Start関数）で、再生時刻をリセット
  - アニメーション処理（Animation関数）
    - 経過時間 delta に応じてアニメーション時間を進める
    - 動作から現在時刻の姿勢を取得
  - 描画処理（Display関数）
    - 現在姿勢を描画

運動学（キネマティクス）

運動学（キネマティックス）

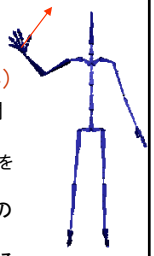
- 運動学（キネマティックス）
  - 多関節体の動きの表現の基礎となる考え方
  - 人間の姿勢は、全関節の関節角度により表現できる
  - 各関節の角度と、各部位の位置・向きの間の関係を計算するための手法





運動学（キネマティックス）

- フォワード・キネマティックス（順運動学）
  - 多関節体の関節角度から、各部位の位置・向きを計算
  - 回転・移動の変換行列の積により計算
- インバース・キネマティックス（逆運動学）
  - 指定部位の位置・向きから、多関節体の関節角度を計算
    - 手先などの移動・回転量が与えられた時、それを実現するための関節角度の変化を計算
  - 動きを指定する時、関節角度よりも、手先の位置・向きなどを使った方がやりやすい
  - ロボットアームの軌道計画等にも用いられる



運動学（キネマティックス）

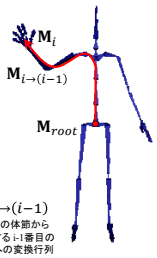
- 運動学の概要
- 順運動学
- 逆運動学

順運動学の計算方法

- フォワード・キネマティックス（順運動学）
  - 各体節（or 関節）の位置・向きを表す変換行列を計算
    - i 番目の体節のローカル座標系からワールド座標系への変換行列 $\mathbf{M}_i$
  - 変換行列の定義・計算方法
    - i 番目の体節からルート体節に向かって順番に隣接する体節への変換行列をかけることで計算できる

$$\mathbf{M}_i = \mathbf{M}_{root}\ \mathbf{M}_{1\to root} \cdots \mathbf{M}_{(i-1)\to(i-2)}\ \mathbf{M}_{i\to(i-1)}$$

（$\mathbf{M}_{root}$：ルート体節の位置・向き／$\mathbf{M}_{i\to(i-1)}$：i番目の体節から隣接するi-1番目の体節への変換行列）



順運動学の計算方法

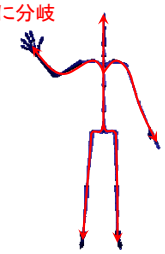
- フォワード・キネマティックス（順運動学）
  - 姿勢（腰の位置・向き、各関節の回転）から、各体節・関節の位置・向きを計算
  - 繰り返し計算
    - ルートから末端に向かって繰り返し
      - 複数の子関節がある場合は各方向に分岐
      - 再帰呼び出しを使うと実装しやすい
    - 前の体節の位置・向きを表す変換行列に、
      1. 次の関節への移動（・回転）
      2. 関節の回転
      3. 次の体節への移動（・回転）

      を順番、に右側にかける適用

順運動学の計算方法

- 繰り返し計算
  - ルートから末端に向かって繰り返し
    - 複数の子関節がある場合は各方向に分岐
    - 再帰呼び出しを使うと実装しやすい
      - 複数の末端に向かっての枝分かれにも対応できる
  - 前の体節の位置・向きに、
    1. 次の関節への移動・回転
    2. 関節の回転
    3. 次の体節への移動・回転

    を順番に適用

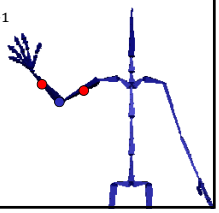


順運動学の計算方法

- 繰り返し計算
  - ルートから末端に向かって繰り返し
    - 複数の子関節がある場合は各方向に分岐
    - 再帰呼び出しを使うと実装しやすい
  - 前の体節の位置・向きに、$\mathbf{M}_{i-1}$
    1. 次の関節への移動（・回転）
    2. 関節の回転 $\mathbf{R}_j$ $\mathbf{T}_{(i-1)\to i}$
    3. 次の体節への移動（・回転）

    を順番に適用 $\mathbf{T}_{(i-1)\to i}$

$$\mathbf{M}_i = \mathbf{M}_{i-1}\ \underset{①}{\mathbf{T}_{(i-1)\to i}}\ \underset{②}{\mathbf{R}_j}\ \underset{③}{\mathbf{T}_{(i-1)\to i}}$$



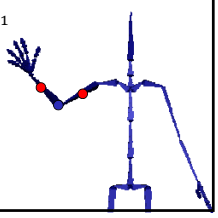
順運動学の計算方法

- 繰り返し計算
  - ルートから末端に向かって繰り返し
    - 複数の子関節がある場合は各方向に分岐
    - 再帰呼び出しを使うと実装しやすい
  - 前の体節の位置・向きに、$\mathbf{M}_{i-1}$
    1. 次の関節への移動（・回転）
    2. 関節の回転 $\mathbf{R}_j$ $\mathbf{T}_{(i-1)\to i}$
    3. 次の体節への移動（・回転）

    を順番に適用 $\mathbf{T}_{(i-1)\to i}$

骨格情報から取得
姿勢情報から取得
骨格情報から取得

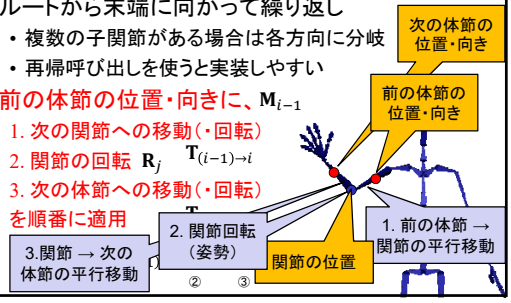
$$\mathbf{M}_i = \mathbf{M}_{i-1}\ \underset{①}{\mathbf{T}_{(i-1)\to i}}\ \underset{②}{\mathbf{R}_j}\ \underset{③}{\mathbf{T}_{(i-1)\to i}}$$



順運動学の計算方法

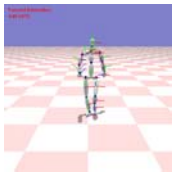
- 繰り返し計算
  - ルートから末端に向かって繰り返し
    - 複数の子関節がある場合は各方向に分岐
    - 再帰呼び出しを使うと実装しやすい
  - 前の体節の位置・向きに、$\mathbf{M}_{i-1}$
    1. 次の関節への移動（・回転）
    2. 関節の回転 $\mathbf{R}_j$ $\mathbf{T}_{(i-1)\to i}$
    3. 次の体節への移動（・回転）

    を順番に適用 $\mathbf{T}_{(i-1)\to i}$



順運動学計算アプリケーション

- ForwardKinematicsApp（一部未実装）
  - MotionPlaybackApp から派生
    - 動作再生処理は、基底クラスを利用
  - 動作再生中に順運動学計算を呼び出して、現在姿勢での全体節の位置・向き（座標系）、全関節の位置を計算して描画
  - 順運動学計算（各自実装）
    - MyForwardKinematicsIteration 関数の一部を作成



順運動学計算のプログラミング

- 順運動学計算
  - 入力：姿勢（各関節の回転＋ルートの位置・向き）
  - 出力：各体節の位置・向き ＋ 各関節の位置
    - STLの可変長配列を使用
    - 関節は向きを持たないと考えて、位置のみを求める

```
// 順運動学計算
void MyForwardKinematics( const Posture & posture,
   vector< Matrix4f > & seg_frame_array,
   vector< Point3f > & joi_pos_array );
```

順運動学計算のプログラミング

```
// 順運動学計算
void MyForwardKinematics( const Posture & posture,
   vector< Matrix4f > & seg_frame_array,
   vector< Point3f > & joi_pos_array )
{
   // 配列初期化
   seg_frame_array.resize( posture.body->num_segments );
   joi_pos_array.resize( posture.body->num_joints );

   // ルート体節の位置・向きを設定
   seg_frame_array[ 0 ].set( posture.root_ori, posture.root_pos, 1 );

   // Forward Kinematics 計算のための反復計算
   ForwardKinematicsIteration(
      posture.body->segments[ 0 ], NULL, posture,
      &seg_frame_array.front(), &joi_pos_array.front() );
}
```

順運動学計算のプログラミング

```
// 順運動学計算
void MyForwardKinematics( const Posture & posture,
   vector< Matrix4f > & seg_frame_array,
   vector< Point3f > & joi_pos_array )
{
   // 配列初期化
   seg_frame_array.resize( posture.body->num_segments );
   joi_pos_array.resize( posture.body->num_joints );

   // ルート体節の位置・向きを設定
   seg_frame_array[ 0 ].set( posture.root_ori, posture.root_pos, 1 );

   // Forward Kinematics 計算のための反復計算
   ForwardKinematicsIteration(
      posture.body->segments[ 0 ], NULL, posture,
      &seg_frame_array.front(), &joi_pos_array.front() );
}
```

最初の体節（ルート体節）と前の体節（なし）を引数として呼び出し、再帰呼び出しによる反復計算を開始

順運動学計算のプログラミング

```
// 順運動学計算のための反復計算
// （ルート体節から末端体節に向かって繰り返し再帰呼び出し）
void MyForwardKinematicsIteration( const Segment * segment,
   const Segment * prev_segment, const Posture & posture,
   Matrix4f * seg_frame_array, Point3f * joi_pos_array)
{
   // 現在の体節に隣接する各関節に対して繰り返し
   for ( int i=0; i<segment->num_joints; i++ )
   {
      // 次の体節・関節を取得、前の体節側（ルート側）の関節はスキップ

      // 次の体節の変換行列＋次の関節の位置を計算
      // 前の体節の変換行列と関節の回転（姿勢より取得）から計算

      // 次の体節に対して繰り返し（再帰呼び出し）
      ForwardKinematicsIteration( ・・・ );
   }
```

順運動学計算のプログラミング

```
// 順運動学計算
// （ルート体節か
void MyForwardKinematicsIteration( const Segment * segment,
   const Segment * prev_segment, const Posture & posture,
   Matrix4f * seg_frame_array, Point3f * joi_pos_a
{
   // 現在の体節に隣接する各関節に対して繰り返
   for ( int i=0; i<segment->num_joints; i++ )
   {
      // 次の体節・関節を取得、前の体節側（ルート側）の関節はスキップ

      // 次の体節の変換行列＋次の関節の位置を計算
      // 前の体節の変換行列と関節の回転（姿勢より取得）から計算

      // 次の体節
      ForwardKin                    ・・・ );
   }
```

現在の体節と一つ前の体節＋現在姿勢を入力
全体節の位置・向きの配列と全関節の位置の配列も
計算結果を格納するための引数として渡す

末端側の隣接する関節・体節に向かって繰り返し

再帰呼び出しによる繰り返し

末端体節に到達したら戻る

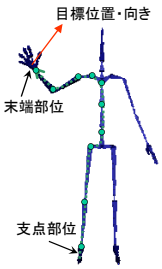
前のスライドの計算方法に従って計算

運動学（キネマティックス）

- 運動学の概要
- 順運動学
- 逆運動学

逆運動学

- インバース・キネマティックス（逆運動学）
  - 入力
    - 現在の姿勢（腰の位置・向き、各関節の回転）
    - 任意の末端部位（エンドエフェクタ）とその目標位置・向き
      - 複数の部位となったり、位置・向きの一部のみを指定する場合がある
    - 支点部位
  - 出力
    - 入力を満たすような姿勢変化を計算

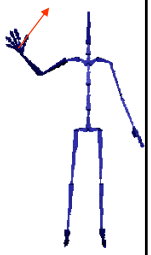


逆運動学計算における問題

- 逆運動学での制約条件に関する問題
  - アンダー・コンストレインツ（少な過ぎる制約）
    - ある部位の位置・向きを指定しただけでは姿勢が一意に決まらない
      - 例：手の位置がきまっても、その手の位置を実現するような全身の関節角度の組は無数に存在する
    - 何らかの評価基準を入れて適当な解を決定する
      - 例：前ステップの姿勢との差がなるべく小さくなる解を選択
  - オーバー・コンストレインツ（多過ぎる制約）
    - 逆に制約が多すぎて、与えられた制約を満足する姿勢がない

逆運動学（IK）の計算方法

- いくつかの計算方法がある
  - 数値的解法（Numerical）
    - 擬似逆行列を使った計算方法
    - Cyclic Coordinate Descent（CCD）法
    - 粒子法（Particle-IK）
    - 非線形最適化問題による計算方法
  - 解析的解法（Analytical）
  - サンプルデータを用いる方法



IKの数値的解法（1）

- ヤコビ行列による計算
  - 各関節の角度を微小変化させたときの、末端部位の微小変化を計算

    $\mathbf{J}\Delta\boldsymbol{\theta} = \Delta\mathbf{p}$
    - $\Delta\boldsymbol{\theta}$ 支点から末端までの各関節の角度変化（n次元）（オイラー角表現が基本）
    - $\Delta\mathbf{p}$ 末端部位の位置・向き変化（3or6 次元）向きにはオイラー角や部位の軸などを使用
  - $\Delta\mathbf{p}$ を満たすような $\Delta\boldsymbol{\theta}$ を計算
    - 通常 n>3or6 なので解は一意に決まらない

IKの数値的解法（2）

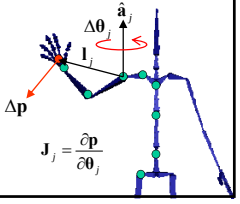
- ヤコビ行列の計算
  - 各関節の角度を微小変化させたときの、末端部位の微小変化

$$\begin{pmatrix} & \\ \mathbf{J}_j & \end{pmatrix} \begin{pmatrix} \\ \Delta\boldsymbol{\theta} \end{pmatrix} = \begin{pmatrix} \\ \Delta\mathbf{p} \end{pmatrix}$$

$\mathbf{J}_j$：関節から末端へのベクトル

$$\mathbf{J}_j = \hat{\mathbf{a}}_j \times \mathbf{l}_j$$

j番目の関節の回転に応じた末端の変化　回転軸の向きを表す単位ベクトル



IKの数値的解法(3)

- 擬似逆行列による計算
  - Jは正則ではなので逆行列は計算できない
  - 擬似逆行列を使用(|Δθ|が最小となる解を計算)

$$\mathbf{J}\Delta\boldsymbol{\theta} = \Delta\mathbf{p}$$
$$\Delta\boldsymbol{\theta} = \mathbf{J}^{+}\Delta\mathbf{p} \qquad \mathbf{J}^{+} = \mathbf{J}^{T}\left(\mathbf{J}\mathbf{J}^{T}\right)^{-1}$$

  - 逆行列が計算できれば擬似逆行列は計算可能
  - $\mathbf{J}$ には、関係のない関節は含まないようにする必要がある(逆行列が計算できなくなる)

IKの数値的解法(4)

- 零空間写像による他の制約の導入

$$\Delta\boldsymbol{\theta} = \mathbf{J}^{+}\Delta\mathbf{p} + \left(\mathbf{I} - \mathbf{J}^{+}\mathbf{J}\right)\Delta\boldsymbol{\theta}'$$

  - 第2項は末端部位の位置・向きには影響しない
    - 影響しない範囲で、任意の修正を適用できる
    - 関節回転可能範囲の制約、重心の位置の制約、等
- 重み付き擬似逆行列

$$\Delta\boldsymbol{\theta} = \mathbf{J}^{+}\Delta\mathbf{p} \qquad \mathbf{J}^{+} = \mathbf{W}^{-1}\mathbf{J}^{T}\left(\mathbf{J}\mathbf{W}^{-1}\mathbf{J}^{T}\right)^{-1}$$

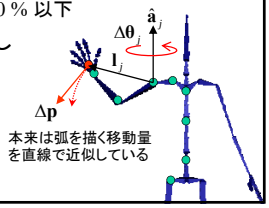
  - $\mathbf{W}$ は各回転軸の重みを表す n×n 行列

IKの数値的解法(5)

- ヤコビ行列による計算方法の注意

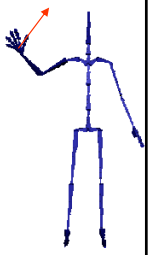
$$\Delta\boldsymbol{\theta} = \mathbf{J}^{+}\Delta\mathbf{p} + \left(\mathbf{I} - \mathbf{J}^{+}\mathbf{J}\right)\Delta\boldsymbol{\theta}'$$

  - $\Delta\mathbf{p}$ の長さは十分小さく取る必要がある
    - リンク全体の長さの 2～10 % 以下
  - 繰り返し計算によって少しずつ姿勢を修正
  - 姿勢が変わる度に $\mathbf{J}$ を計算し直す必要がある



逆運動学(IK)の計算方法

- いくつかの計算方法がある
  - 数値的解法(Numerical)
    - 擬似逆行列を使った計算方法
    - Cyclic Coordinate Descent(CCD)法
    - 粒子法(Particle-IK)
    - 非線形最適化問題による計算方法
  - 解析的解法(Analytical)
  - サンプルデータを用いる方法



CCD法

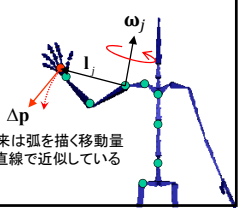
- Cyclic Coordinate Descent(CCD)法
  - 疑似逆行列を使う方法と同様の繰り返し計算で姿勢変化を計算
    - 疑似逆行列を使う方法よりも簡単な方法
  - 各関節の回転を独立に計算
    - 末端関節の位置が目標位置に来るように、各関節の回転軸・回転角度を計算

CCD法

- Cyclic Coordinate Descent(CCD)法
  - 各関節の回転を独立に計算
    - 末端関節の位置が目標位置に来るように、各関節の回転軸・回転角度を計算

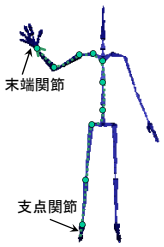
$$\boldsymbol{\omega}_j = \mathbf{l}_j \times \Delta\mathbf{p}$$

    - 回転軸に回転角度をかけたベクトルの形式で求まる
    - 必要に応じて、回転行列などの別の表現に変換する
    - 必要に応じて、各関節の回転量にウェイトを適用することも可能



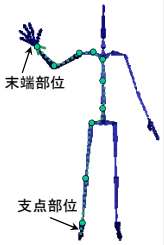
CCD法の計算手順

1. 指定された支点関節・末端関節にもとづき、支点関節から末端関節までの経路を決定
2. 繰り返し計算により、少しずつ姿勢を変化
3. 各繰り返し処理の中で、末端関節から支点関節に向かって順番に処理を適用
   - 末端関節の目標位置を満たすように各関節の回転を計算



CCD法の計算方法(1)

- 末端→支点関節のパス(関節順番)の決定
  - 各体節のどちら側(何番目の関節側)に末端・支点関節があるかが分かれば、パスを決定可能
    - 事前にインデックスを作成しておく
    - あるいは、ルートがどちらにあるかは分かるので、末端・支点関節からそれぞれルートに向かって辿り、合流した体節でパスを結合する
      - 後の説明では、こちらの方法を使用
    - 実装が難しければ、最初は常にルート体節が支点と仮定して作成すると、簡単になる

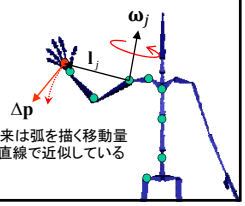


CCD法の計算方法(2)

- 各関節の回転を計算
  - 収束するまで繰り返し計算
    - 末端関節から支点関節に向かって順番に繰り返し
      - 各関節のヤコビ行列にもとづき、各関節の回転を独立に計算
      - 目標を満たすための各関節の回転軸・回転角度を計算

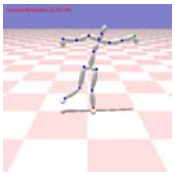
$$\boldsymbol{\omega}_j = \mathbf{l}_j \times \Delta\mathbf{p}$$

      - 末端関節との間に腰(ルート)がある場合は、ルートの移動・回転が必要
      - 各関節の回転量に重みをかけて調整することも可能



逆運動学計算アプリケーション

- InverseKinematicsCCDApp (一部未実装)
  - CCD法による逆運動学計算のアプリケーション
  - マウス操作による関節の選択・移動(実装済み)
    - 以前の授業のピッキング技術(スクリーン座標で判定)により末端・支点関節を選択
    - マウスドラッグに応じて、視線に垂直な超平面上で、末端関節の目標位置を移動
  - CCD法による逆運動学計算(各自実装)



CCD法のプログラミング

- CCD法による逆運動学計算
  - 入力:現在姿勢、支点関節番号、末端関節番号、末端関節の目標位置
  - 出力:逆運動学計算後の姿勢
    - 現在姿勢の入力と計算後の姿勢の出力に、同じ変数を使用

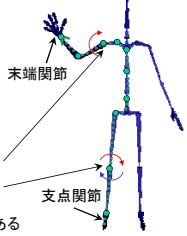
```
// 逆運動学計算(CCD法)
void  ApplyInverseKinematicsCCD( Posture & posture,
  int base_joint_no, int ee_joint_no, Point3f ee_joint_position );
```

CCD法のプログラミング(1)

- 支点関節から末端関節への経路(パス)探索
  - 入力:骨格情報、支点関節番号、末端関節番号
  - 出力:経路に含まれる関節番号+方向のリスト
    - 出力には可変長配列を使用
    - 関節番号+方向(1 or −1)
      - ルートよりも支点側(1)にあるか末端側(−1)にあるかで、関節周りの回転の方向が逆転するため、方向も合わせて出力

末端側:関節回転により末端側が回転

支点側:関節回転により支点側が回転
関節の回転を表すときには、回転方向を反対にする必要がある



CCD法のプログラミング（1）

- 支点関節から末端関節への経路（パス）探索
  - 入力：骨格情報、支点関節番号、末端関節番号
  - 出力：経路に含まれる関節番号＋方向のリスト
    - ルート体節が支点の場合は、支点番号を －1 とする
    - 出力には可変長配列を使用
    - 経路に含まれる関節番号＋方向（1 or －1）のリスト
      - ルートよりも支点側（1）にあるか末端側（－1）にあるかで、関節周りの回転の方向が逆転するため、方向も合わせて出力

```
// 末端関節から支点関節へのパスを探索
void  FindJointPath( const Skeleton * body,
   int base_joint_no, int ee_joint_no,
   vector< int > & joint_path, vector< int > & joint_path_signs );
```

CCD法のプログラミング（1）

```
// 末端関節から支点関節への経路（パス）を探索
void  FindJointPath( const Skeleton * body,
   int base_joint_no, int ee_joint_no,
   vector< int > & joint_path, vector< int > & joint_path_signs )
{
   // 末端関節から探索を開始

   // 末端関節からルートに向かうパスを探索
   while（・・・）
      // ルート体節か支点関節に到達したら終了
      // 現在の関節をパスの配列に追加、関節の符号は全て 1 とする

   // 支点関節に到達したら終了（支点がルート体節 or 現在の関節）

   // 支点関節からルートに向かうパスを探索（上と同様の処理）

   // 2つのパスを統合
}
```

CCD法のプログラミング（1）

```
// 末端関節から支点関節への経路（パス）を探索
void  FindJointPath( const Skeleton * body,
   int base_joint_no, int ee_joint_no,
   vector< int > & joint_path, vector< int > & joint_path_signs )
{
   // 末端関節から探索を開始

   // 末端関節からルートに向かうパスを探索
   while（・・・）
      // ルート体節か支点関節に到達したら終了
      // 現在の関節をパスの配列に追加、関節の符号は全て 1 とする

   // 支点関節に到達したら終了（支点がルート体節 or 現在の関節）

   // 支点関節からルートに向かうパスを探索（上と同様の処理）

   // 2つのパスを統合
}
```

最初はルート体節が支点であると仮定して、ここまでの処理を作成すると良い
うまくいったら、任意の支点関節が指定されても対応できるように、残りの処理も作成する

CCD法のプログラミング（2）

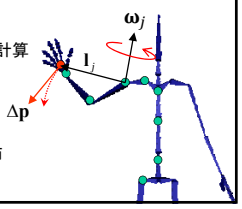
- 各関節の回転を計算
  - 収束するまで繰り返し計算
    - 末端関節から支点関節に向かって順番に繰り返し
      - 目標を満たすための各関節の回転軸・回転角度を計算

$$\boldsymbol{\omega}_j = \mathbf{l}_j \times \Delta\mathbf{p}$$

        - » 関節のローカル座標系で計算
      - 現在の関節の回転に、求めた回転を適用して、姿勢を更新

$$\mathbf{R}_j' = \mathbf{R}(\boldsymbol{\omega}_j)\,\mathbf{R}_j$$

        - » ルートよりも支点側の関節では、順番は逆になる



CCD法のプログラミング（2）

```
// 逆運動学計算（CCD法）
void  ApplyInverseKinematicsCCD( Posture & posture,
  int base_joint_no, int ee_joint_no, Point3f ee_joint_position )
{
   // 末端関節から支点関節へのパスを探索
   FindJointPath( ・・・);

   // CCD法の繰り返し計算（収束するか、一定回数繰り返したら終了）
   for ( int i=0; i<max_iteration; i++ )
   {
      // 末端関節から支点関節に向かって繰り返し
      for ( int j=0; j<joint_path.size(); j++ )
      {
         // 末端関節を目標位置に近づける、現在の関節の回転を計算
         // 回転を適用する際には、回転の方向を考慮

         // 順運動学計算を再計算
```

CCD法のプログラミング（2）

```
// 逆運動学計算（CCD法）
void  ApplyInverseKinematicsCCD( Posture & posture,
  int base_joint_no, int ee_joint_no, Point3f ee_joint_position )
{
   // 末端関節から支点関節へのパスを探索
   FindJointPath( ・・・);

   // CCD法の繰り返し計算（収束するか、一定回数繰り返したら終了）
   for ( int i=0; i<max_iteration; i++ )
   {
      // 末端関節から支点関節に向かって繰り返し
      for ( int j=0; j<joint_path.size(); j++ )
      {
         // 末端関節を目標位置に近づける、現在の関節の回転を計算
         // 回転を適用する
         // 順運動学計算を
```

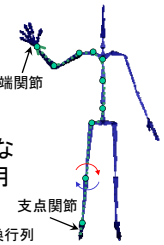
末端関節と現在の関節の間にルート体節がある場合は、ルート体節に移動・回転を適用

CCD法のプログラミング（3）

- ルートよりも支点側にある関節の回転
  - 前のスライドと同じ方法で関節回転を計算
    - 右図の赤矢印の回転が求まる
  - 逆の回転を適用
    - 右図の青矢印の回転を適用
  - そのままでは、ルートが固定されて、支点関節が移動する
  - 支点関節の位置・向きを保つような回転・移動を求めて、ルートに適用

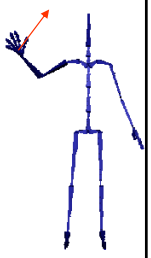
$$\mathbf{M} = \mathbf{M}_{base}\ \mathbf{M}_{base}{}'^{-1}$$

関節回転の適用前・後の支点関節の変換行列



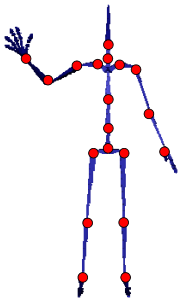
逆運動学（IK）の計算方法

- いくつかの計算方法がある
  - 数値的解法（Numerical）
    - 擬似逆行列を使った計算方法
    - Cyclic Coordinate Descent（CCD）法
    - 粒子法（Particle-IK）
    - 非線形最適化問題による計算方法
  - 解析的解法（Analytical）
  - サンプルデータを用いる方法



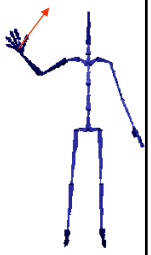
粒子法による解法

- 関節点を粒子とみなして、粒子の位置を計算
  - 粒子シミュレーションの手法により、末端関節の目標位置や粒子間の距離の条件を満たすような粒子の位置を計算
- 関節点（粒子）の位置の制約を満たすように、姿勢（全関節の回転）を計算



逆運動学（IK）の計算方法

- いくつかの計算方法がある
  - 数値的解法（Numerical）
    - 擬似逆行列を使った計算方法
    - Cyclic Coordinate Descent（CCD）法
    - 粒子法（Particle-IK）
    - 非線形最適化問題による計算方法
  - 解析的解法（Analytical）
  - サンプルデータを用いる方法

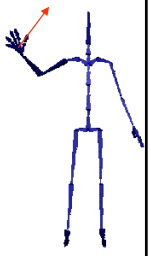


非線形最適化問題による解法

- 最適化問題と考えて、指定部位の位置・向きの条件を満たすような姿勢（関節角度の組み合わせ）を探索
- 疑似逆行列による解法（線形問題）を、より一般化した解法
- 目標関数の定義や最適化問題の解法にさまざまな方法がある
  - 手法によっては計算時間がかかる

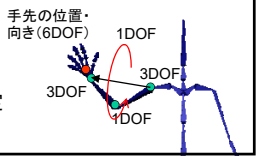
逆運動学（IK）の計算方法

- いくつかの計算方法がある
  - 数値的解法（Numerical）
    - 擬似逆行列を使った計算方法
    - Cyclic Coordinate Descent（CCD）法
    - 粒子法（Particle-IK）
    - 非線形最適化問題による計算方法
  - 解析的解法（Analytical）
  - サンプルデータを用いる方法



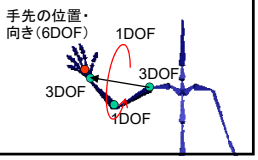
IKの解析的解法(1)

- 人間の骨格に基づく効率的な計算方法
- 手足の解析的なIK計算
  - 肩 or 股関節を支点として、手 or 足の位置・向きを満たす、腕・脚の姿勢を計算
  - 手先・足先の位置・向きは6自由度、
  - 腕・脚は全体で7自由度
  - 残りの自由度は1(ひじ・ひざの回転)のみ
    - 何らかの方法でこれを決定すれば解は一意に決まる



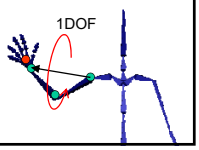
IKの解析的解法(2)

1. 入力:手先の目標位置・向き、肩(胴体)の位置・向き、腕の2本の体節の長さ
2. 目標位置と肩の距離から、ひじの屈伸回転を計算(1DOF)
3. 手先が目標位置に来るように、肩の回転を計算(2DOF)
4. 何らかの方法で、ひじの旋回回転を決定(1DOF)
5. 手先の目標向きに応じて、手首の回転を計算(3DOF)



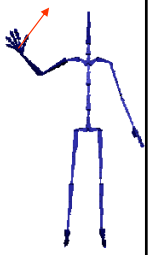
IKの解析的解法(3)

- ひじ(ひざ)の旋回角度の計算方法の例
  - 現在の姿勢の旋回角度を保持
    - 姿勢を少しだけ変更するような場合であれば、有効
  - 標準的な旋回角度を適用
    - どのような目標位置・向きのときに、どのような旋回角度になるかのデータを用意しておき、そのデータにもとづいて決定



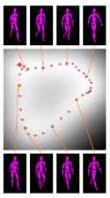
逆運動学(IK)の計算方法

- いくつかの計算方法がある
  - 数値的解法(Numerical)
    - 擬似逆行列を使った計算方法
    - Cyclic Coordinate Descent(CCD)法
    - 粒子法(Particle-IK)
    - 非線形最適化問題による計算方法
  - 解析的解法(Analytical)
  - サンプルデータを用いる方法



サンプルデータを用いる方法

- Style-based IK [Grochow et al. 2004]
  - あらかじめ多数の姿勢データを学習し、制約条件が与えられると、制約条件を満たしつつ、学習データに近いような、適切な姿勢を計算
    - 潜在空間(低次元空間)に学習姿勢を写像
    - 潜在空間の中で姿勢を探索・合成
  - サンプルデータにもとづき冗長性の問題を解決
  - 一つの学習モデルで複数の動作は扱えない
    - 同じ制約条件でも動作によって適切な姿勢は異なる



サンプルデータを用いる方法

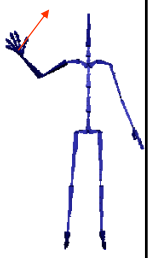
- 姿勢補間を用いる方法もある
  - 少ないパラメタ(決められた動作や末端部位)に対して、十分多数のサンプル姿勢が用意できる場合
  - 複数の姿勢の補間が必要になる
- 姿勢補間については次回説明

逆運動学計算の利用時の注意

- いずれの方法を用いても、入力姿勢から大きく離れた姿勢を生成することは難しい
  - 広い範囲の自然な姿勢の生成は難しい
- 姿勢や動作によっては、適切な末端関節の目標位置を与えることが難しい
  - 動作生成のための軌道を与えることは難しい
- 微小な姿勢の変化であれば対応可能
  - 手先や足先の位置を周囲の環境に応じて少し修正する程度の姿勢変形

逆運動学の計算方法のまとめ

- いくつかの計算方法がある
  - 数値的解法（Numerical）
    - 擬似逆行列を使った計算方法
    - Cyclic Coordinate Descent（CCD）法
    - 粒子法（Particle-IK）
    - 非線形最適化問題による計算方法
  - 解析的解法（Analytical）
  - サンプルデータを用いる方法



レポート課題

レポート課題（予定）

- キャラクタ・アニメーション基礎技術
  - サンプルプログラム（human_sample.cpp）の未実装部分を作成

1. 順運動学計算
2. 姿勢補間
3. キーフレーム動作再生
4. 動作接続・遷移
5. 動作補間
6. 逆運動学計算（CCD法）
   1. ルート体節を支点とする場合
   2. 任意の関節を支点とする場合

順運動学計算アプリケーション

- ForwardKinematicsApp（一部未実装）
  - MotionPlaybackApp から派生
    - 動作再生処理は、基底クラスを利用
  - 動作再生中に順運動学計算を呼び出して、現在姿勢での全体節の位置・向き（座標系）、全関節の位置を計算して描画
  - 順運動学計算（各自実装）
    - MyForwardKinematicsIteration関数の一部を作成

逆運動学計算アプリケーション

- InverseKinematicsCCDApp（一部未実装）
  - CCD法による逆運動学計算のアプリケーション
  - マウス操作による関節の選択・移動（実装済み）
    - 以前の授業のピッキング技術（スクリーン座標で判定）により末端・支点関節を選択
    - マウスドラッグに応じて、視線に垂直な超平面上で、末端関節の目標位置を移動
  - CCD法による逆運動学計算（各自実装）
    - ルート体節を支点とする → 任意の関節を支点とする
    - FindJointPath関数の一部を作成
    - ApplyInverseKinematicsCCD関数の一部を作成

レポート課題（予定）

- キャラクタ・アニメーション基礎技術
  - サンプルプログラム（human_sample.cpp）の未実装部分を作成

1. 順運動学計算
2. 姿勢補間
3. キーフレーム動作再生
4. 動作接続・遷移
5. 動作補間
6. 逆運動学計算（CCD法）
   1. ルート体節を支点とする場合
   2. 任意の関節を支点とする場合

残りの課題は次回説明

まとめ

- 前回の復習
- サンプルプログラム
- 運動学
  - 順運動学
  - 逆運動学
- レポート課題

次回予告

- キャラクタ・アニメーションの基礎
- 骨格モデル・姿勢・動作の表現
- 動作データの作成
- 運動学
- 姿勢・動作ブレンディング
- 応用技術