CCD-IK and Particle-IK

http://www.tmps.org/index.php?CCD-IK%20and%20Particle-IK

Movie 1. CCD-IK & Particle-IK & Jacobian IKs

本記事では Cyclic-Coordinate-Descent（CCD）法と，Particle-IK（PIK）法と呼ばれる逆運動学（インバースキネマティクス，Inverse Kinematics，IK）の計算アルゴリズムを紹介します．いずれも反復計算に基づく最適化技法で，1ステップあたりの計算量が少なく，かつ収束までの反復回数が少ないため，ゲーム等のリアルタイムアニメーションに適しています．

ヤコビアンを用いた逆運動学，クォータニオン逆運動学のアルゴリズムは，全ての関節の角度を同時に少しずつ更新しながら近似解を探します．一方，CCD 法は各 1 回の計算ステップにおいては，ある 1 つの関節の最適角度のみを求めます．それを全ての関節に順番に繰り返すことで，スケルトン全体としての近似解を探します．また，PIK 法では関節角度ではなく関節位置を直接操作とします．具体的には，布地シミュレーション等で用いられるパーティクルシミュレーションの要領で，1 つ 1 つの関節位置を順番に繰り返し更新することで近似解を求めます．

CCD 法，PIK 法ともに計算が早く，アルゴリズムも単純なため実装も容易です．また計算パラメータの調整次第で様々な解を得られる点で，ヤコビ法などの数値計算法に対して多くの点で有利です．ただし，いずれも関節の可動域の調整に難点，もしくは可動域の設定自体が不可能だという欠点もあります．

記事の構成

- サンプルプログラム
- 参考文献
  - Cyclic-Coordinate-Descent（CCD）法
    - CCD 法の概要
    - 各計算ステップにおける各関節角度の最適化
    - 反復手順
    - サンプルコード
    - CCD 法の長所，短所
  - Particle Inverse Kinematics（PIK）
    - PIK のアルゴリズム概要
    - 仮想バネを利用したパーティクル位置の更新
    - サンプルコード
  - PIK 法の長所と短所
- まとめ

サンプルプログラム

サンプルプログラムは Visual Studio 2008 Professional と XNA Game Studio 3.0 を用いて作成しました．

Visual Studio 2005 + XNA GS 2.0プロジェクト（35KB），2009/02/17 更新

キーボードの左シフトキーを押しながらスクリーン上でマウスを移動させると，マウスポインタにスケルトンの先端＝エンド・エフェクタが一致するようにスケルトンがアニメートします．また，10キーパッドの「0」を押すと CCD 法，「1」で PIK 法，「2」でオイラー角のヤコビ法，「3」でクォータニオンのヤコビ法による IK アルゴリズムにそれぞれ切り替えられます．

参考文献

- Chris Welman, "Inverse kinematics and geometric constraints for articulated figure manipulation", M.Sc Thesis, Simon Fraser University, 1993
  - CCD 法といえばこちらの文献がよく引用されています．わりとサラッと触れられているので，他にオリジナルの論文があるのかもしれません．
- Chris Hecker, Bernd Raabe, Ryan W. Enslow, John DeWeese, Jordan Maynard, and Kees van Prooijen, "Real-time Motion Retargeting to Highly Varied User-Created Morphologies", ACM Transactions on Graphics, vol.27, no.3, Article-27, 2008.
  - PIK 法を初めて導入した論文です．この論文では効率的なモーションリターゲティング（体格の異なるキャラクタ間で同一のアニメーションデータを共有するための技術）を実現するための 1 要素技術として PIK 法を利用しています．なお，下記のプレゼンテーションでも PIK 法について触れられています．

- Chris Hecker, "How To Animate a Character You've Never Seen Before", Game Developers Conference 2007.

Cyclic-Coordinate-Descent(CCD)法

CCD 法の概要

CCD 法は，1 回の計算ステップで 1 つの関節角度のみを最適化し，それを全ての関節について反復することで全体としての近似解を求めます．

まず，数式を使った定義から示します．スケルトンを構成する N 個の関節の回転量をそれぞれ θ_i, $i=\{1,2,\cdots,N\}$ と表し，全てをまとめたベクトル $\Theta=[\theta_1\ \theta_2\cdots\theta_N]$ を定義します．このとき，スケルトンの先端＝エンド・エフェクタの位置 p_E は順運動学計算 $p_E=FK(\Theta)$ によって求められます．一方，逆運動計算ではエフェクタの位置 p_E と目標位置 p_T を最小化するような各関節の回転量を求めます．この最小化問題は，次のように定式化されます．

$$\Theta=\mathrm{argmin}_{\Theta}\left|p_T-FK(\Theta)\right|$$

ヤコビアンを用いた解法では，エフェクタを少しずつ目標位置に近づけるように，全関節回転量の微小変化 $\triangle\Theta$ を計算します．一方，CCD 法では全ての関節回転量 Θ を同時に更新するのではなく，各関節の回転量 θ_i を 1 つずつ最適化するアプローチをとります．つまり，1 回の計算ステップでは部分的に最小化問題を解き，それを全関節について反復する方法をとります．

$$\theta_i=\mathrm{argmin}_{\theta_i}\left|p_T-FK(\Theta)\right|$$
$$i=N,N-1,\cdots,1,N,\cdots$$

ここで，関節の添え字 i は，式に示すようにエフェクタからルートに向かう順にループさせます．この反復計算により，計算ステップを経るたびに解が必ず最適値に近づくこと，つまりエフェクタが目標地点に近づくことが保障されます．

各計算ステップにおける各関節角度の最適化

CCD 法の各計算ステップにおける最小化問題は，閉じた形式（Closed-Form）で一意に計算できます．この計算をおおざっぱに説明するために，Fig.1 に示すような始端点を中心に回転する 1 リンク構造を考えます．

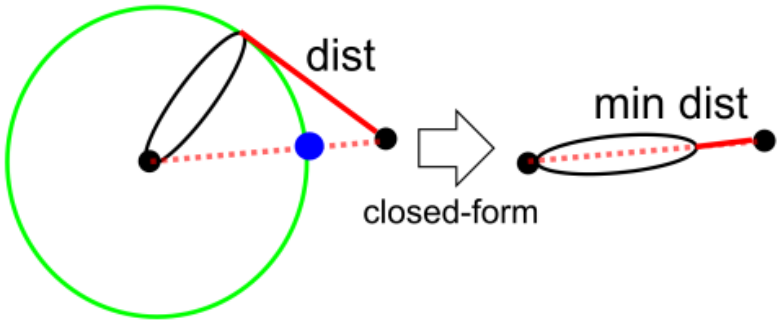

Fig.1 最適なリンク回転量に関する閉じた形式の解法

ここで，リンクの終端点が目標位置に最も近づくように回転量を最適化する問題を考えます．まず，リンクが 360 度回転したときの終端点の軌跡は緑線で表されます．この軌跡上で目標位置に最も近い点は，青丸に示す位置となります．つまり最適回転角は，リンクの終端点がリンク始端点と目標位置を結ぶ線分上に位置するような回転量となります．したがって，リンクの始端から終端へ向かうベクトルを，リンク始端から目標位置へ向かうベクトルに一致させる回転量を計算すれば良いことがわかります．さらにこの計算は 2 つのベクトルの外積と内積を用いて，下記の手順で一意に計算できます．

- 定義
  - v_1:リンクの始端から終端へ向かう単位ベクトル
  - v_2:リンクの始端から目標位置へ向かう単位ベクトル
- 解法:回転軸とその軸周りの回転角度を計算する．
  i. 回転軸 v_{axis} を v_1 と v_2 の外積 $v_{axis}=v_1\times v_2$ によって計算．
    - この回転軸は v_1 と v_2 が張る平面の法線なので，ベクトル v_1 はその平面上を最短経路で回転移動します．

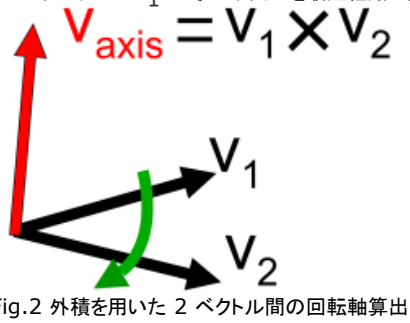

Fig.2 外積を用いた 2 ベクトル間の回転軸算出

  ii. 軸周りの回転角 θ_α を $Cos^{-1}(v_1\cdot v_2)$ により算出．

iii. v_{axis} と θ_a を用いて回転クォータニオンを合成.

さらに，この計算法を複数のリンクで構成されたスケルトンに拡張します．CCD 法では 1 回の計算ステップで回転させるのは 1 つの関節だけですので，Fig.3 のような仮想リンクの考え方を導入することで，1 リンクの場合と同一の計算手順を利用できます．具体的には，回転させる関節とエフェクタを結ぶような 1 つのリンクを仮想的に作成し，その仮想リンクについて上述の計算手順を適用します．さらに具体的にいうと，v_1 と v_2 をそれぞれ「回転させる関節からエフェクタへ向かう単位ベクトル」と，「回転させる関節から目標位置へ向かう単位ベクトル」に変更するだけです．

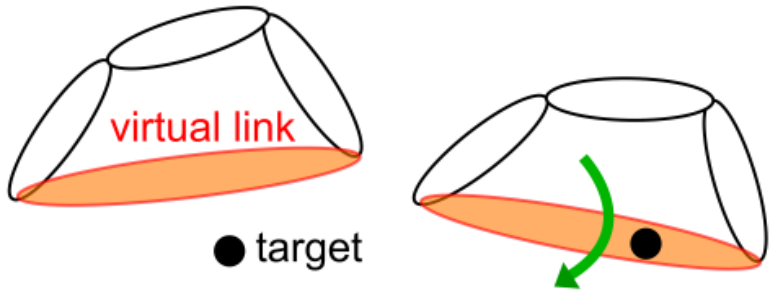

Fig.3 仮想リンクを用いたスケルトンの回転

反復手順

各計算ステップにおける各関節の最適角度は，上述の閉じた形式の解法で求められることがわかりました．CCD 法では，この計算ステップをエフェクタに近い関節からルート関節に向かう順に繰り返し適用することで，スケルトン全体としての最適解を求めます．この繰り返し手順と，それに伴うスケルトン姿勢の変化の例を Fig.4 に示します．ここで，青丸がスケルトンのエフェクタの到達目標位置，緑丸が各計算ステップでの注目関節を示します．

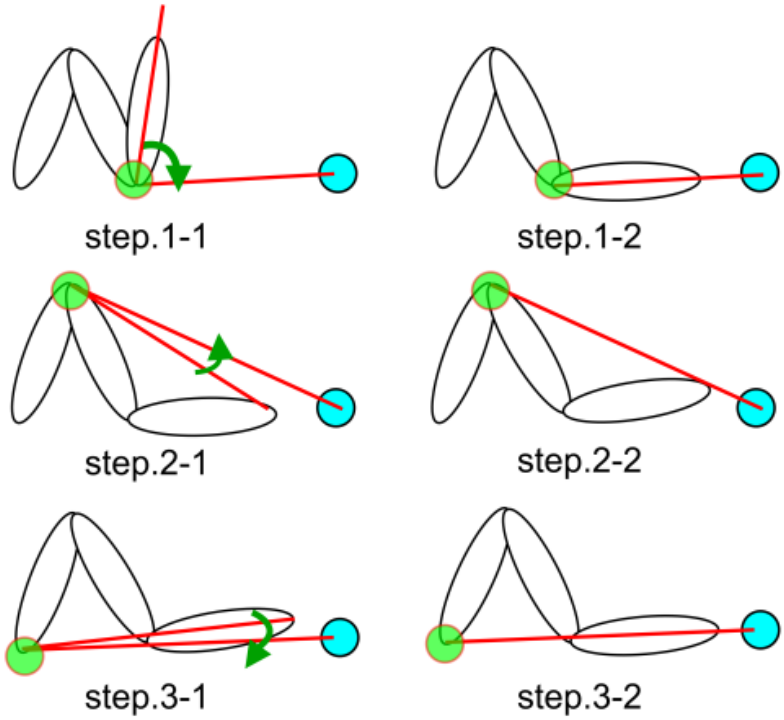

Fig.4 CCD-IK の計算アルゴリズム

1. step.1-1：エフェクタに最も近い関節から，エフェクタに向かうベクトルと目標位置に向かうベクトルをそれぞれ計算
2. step.1-2：2 つのベクトルを利用して，エフェクタに最も近い関節の回転量を修正
3. step.2-1：中間の関節から，エフェクタに向かうベクトルと目標位置に向かうベクトルをそれぞれ計算
4. step.2-2：2 つのベクトルを利用して，中間の関節の回転量を修正
5. step.3-1：ルート関節から，エフェクタに向かうベクトルと目標位置に向かうベクトルをそれぞれ計算
6. step.3-2：2 つのベクトルを利用して，ルート関節の回転量を修正
7. step.4：step.1-1 から再度繰り返し

計算の打ち切りは，エフェクタと目標位置の誤差がある閾値以下になる時点か，反復回数に上限値を設定する方法のいずれかが採られます．なお，リアルタイム性を重視する場合は後者の方法が適しているでしょう．

サンプルコード

CCD 法の計算アルゴリズムのサンプルコードを示します．実装にあたっては，ベクトル v_1, v_2, v_{axis} のそれぞれを注目関節の局所座標系で計算する必要があります．これは，各関節に加えられる修正回転量が，それぞれの局所座標系における値でなければならないためです．そのため，ワールド座標系における v_1, v_2 を計算した後，それらを局所座標系に変換する処理を追加しています．また，内積を用いて算出される回転角度が一定閾値以下(1.0e-5f 以下)のときは，外積の計算が発散しがちなので，回転量の修正処理自体を省略しています．反復計算は誤差の上限値，もしくは最大反復回数を満足したときに終了します．

```
1 /// <summary>Cyclic-Coordinate Descent IK ソルバ</summary>
```

```
/// <param name="skeleton">スケルトン</param>
/// <param name="effector">エフェクタ関節</param>
/// <param name="targetPos">目標位置</param>
/// <param name="numMaxIteration">最大反復回数</param>
/// <param name="errToleranceSq">誤差許容値の二乗</param>
static public void SolveCcd(Skeleton skeleton, Joint effector, Vector3 targetPos, int numMaxIteration, float errToleranceSq)
{
  Vector3 localTargetPos = Vector3.Zero;
  Vector3 localEffectorPos = Vector3.Zero;

  for (int it = 0; it < numMaxIteration; ++it) {
    for (Joint joint = effector.Parent; joint != null; joint = joint.Parent) {
      // エフェクタの位置の取得
      Vector3 effectorPos = skeleton.GetGlobalPosition(effector.Index);
      // 注目ノードの位置の取得
      Vector3 jointPos = skeleton.GetGlobalPosition(joint.Index);

      // ワールド座標系から注目ノードの局所座標系への変換
      Matrix invCoord = Matrix.Invert(skeleton.GetGlobalCoordinate(joint.Index));
      // エフェクタ，到達目標のローカル位置
      localEffectorPos = Vector3.Transform(effectorPos, invCoord);
      localTargetPos = Vector3.Transform(targetPos, invCoord);

      // (1) 基準関節→エフェクタ位置への方向ベクトル
      Vector3 basis2Effector = Vector3.Normalize(localEffectorPos);
      // (2) 基準関節→目標位置への方向ベクトル
      Vector3 basis2Target = Vector3.Normalize(localTargetPos);

      // ベクトル (1) を (2) に一致させるための最短回転量 (Axis-Angle)
      //
      // 回転角
      float rotationDotProduct = Vector3.Dot(basis2Effector, basis2Target);
      float rotationAngle = (float)Math.Acos(rotationDotProduct);
      if (rotationAngle > 1.0e-5f) {
        // 回転軸
        Vector3 rotationAxis = Vector3.Cross(basis2Effector, basis2Target);
        rotationAxis.Normalize();
        // 関節回転量の補正
        joint.Rotation = Quaternion.CreateFromAxisAngle(rotationAxis, rotationAngle) * joint.Rotation;
      }
    }

    if ((localEffectorPos - localTargetPos).LengthSquared() < errToleranceSq)
      return;
  }
}
```

CCD 法の長所，短所 [t]

CCD 法は比較的計算が早いです．各計算ステップは順運動学計算や Acosなどの三角関数，行列-ベクトル演算を含むのでさほど高速ではありませんが，反復計算の収束が早いので，ヤコビ法などと比較すると全体の計算時間は短くなります．また，アルゴリズムの単純さも重要でしょう．

一方，CCD 法には関節の回転可動域の設定と，位置と方向を同時に指定した計算が苦手だという問題があります．まず前者ですが，CCD 法は"各計算ステップで必ず最適解を求める"ことが，スケルトン全体で最適解に収束する条件です．しかし可動域を設定すると，可動域から外れた角度を修正する必要が生じ，解の最適性を保証できなくなります．後者の問題については，CCD 法は基本的に目標"位置"を指定するアルゴリズムであることに起因します．すなわち，エフェクタの方向を指定するためには，エフェクタの周辺の関節などに追加の目標位置を指定しなければなりません．しかし，2 つの目標位置を同時に満足する解を見つけるためには，1 回の計算ステップで 2 つの"最適性"を同時に重み付けて最適化するか，繰り返し交互に最適化しなければなりませんが，いずれの方法でも反復計算が発散しやすく，あまり頑健な方法ではありません．

Particle Inverse Kinematics(PIK) [t]

PIK のアルゴリズム概要 [t]

PIK 法はこれまでに説明した IK 法と大きく異なり，関節角度を直接操作するのではなく，関節位置を操作することで逆運動学計算を行います．そのとき，関節角度は各関節位置を満たすように計算される，あくまでも副次的，間接的な値として扱われます．

PIK 法の基本的な計算手順について，Fig.5 を例に説明します．

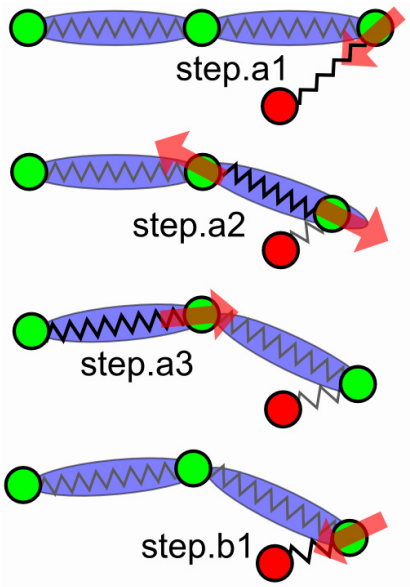

Fig.5 Particle-IK の反復計算手順

1. スケルトンの全ての関節とエンド・エフェクタの位置にパーティクルを置く
2. スケルトンのリンクに対応するように，隣接するパーティクルの間に接続関係と距離の制約条件を与える．
   - 仮想的なバネで 2 つのパーティクルを接続するようなものです．
   - ルートのパーティクルは最初の位置に固定します
3. エンド・エフェクタのパーティクルと目標位置の間に，距離 0 の制約条件を与えます
   - 長さ 0 のときに力が発生しないような仮想的なバネで接続します
4. 以下のループを反復する
   i. エフェクタのパーティクルを目標位置に一致させる．
   ii. エフェクタとその親関節(A)のパーティクルを，2 つ同時に，その間の制約距離を満たすように移動
   iii. 親関節(A)とその親関節(B)のパーティクルを，2 つ同時に，その間の制約距離を満たすように移動
   iv. 以下同文
   v. ルート直下の子関節のパーティクルを，ルート関節との制約距離を満たすように移動
   vi. 最初に戻る．

仮想バネを利用したパーティクル位置の更新

ここで，「パーティクルを，2 つ同時に，その間の制約距離を満たすように移動」の部分についてもう少し詳しく説明します．パーティクル間に張られた仮想バネは，1 回の計算ステップで必ず制約距離に戻ります．つまり，エフェクタと目標位置間のバネは必ず長さ 0 に戻り，各関節間のバネは対応するボーンの長さに戻ります．そのときバネの伸縮にあわせて，両端のパーティクルが同じ距離だけ，バネの張られた直線上を移動します．図に示すと Fig.6 のようになります．

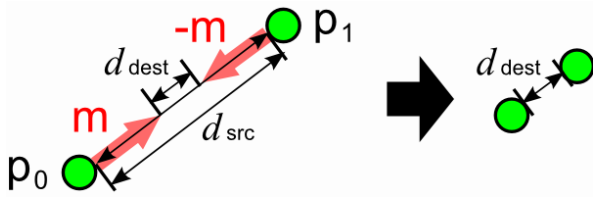

Fig.6 パーティクル位置の更新処理

2 つのパーティクルの位置をそれぞれ p_0, p_1 とすると，バネの長さは$d_{src} = |p_1 - p_0|$ と表されます．ここで，バネの目標長さを d_{dest} とすると，2 つのパーティクルの移動量 L_m はいずれも $L_m = (d_{src} - d_{dest})/2.0$ となります．また，移動方向を表す単位ベクトルは p_0 については $(p_1 - p_0)/d_{src}$, p_1 については $(p_0 - p_1)/d_{src}$ となります．したがって，移動ベクトル m は $m = (p_1 - p_0)L_m/d_{src}$ となります．さらに L_m を展開してまとめると，次の式が得られます．

$$m = (p_1 - p_0)*(0.5 - 0.5*d_{dest}/d_{src})$$

サンプルコード

PIK 法のアルゴリズムを実装したサンプルコードを示します．まず，引数として渡されたスケルトンの情報をもとに，初期パーティクル位置（particles）と仮想バネの目標長さ（constraints）を計算します．その後，目標位置に近いパーティクルから順に，上述の手順に沿って位置の更新処理を反復します．指定回数の反復計算が終わったら新しいパーティクル位置を満たすようにスケルトンの姿勢を更新します．

```
1 /// <summary>パーティクル IK ソルバ</summary>
2 /// <param name="skeleton">スケルトン</param>
3 /// <param name="effector">エフェクタ関節</param>
4 /// <param name="targetPos">目標位置</param>
```

```
  /// <param name="numMaxIteration">最大反復回数</param>
  /// <returns>パーティクル位置</returns>
  static public Vector3[] SolveParticle(Skeleton skeleton, Joint effector, Vector3 targetPos, int numMaxIteration)
  {
    List particles = new List();
    List<float> constraints = new List<float>();
    List joints = new List();

    // パーティクルの作成
    for (Joint j = effector; j.Parent != null; j = j.Parent) {
      particles.Add(skeleton.GetGlobalPosition(j.Index));
      constraints.Add(j.Offset.Length());
      joints.Add(j);
    }
    joints.Add(joints[joints.Count-1].Parent);
    Vector3 rootPos = skeleton.RootPosition;

    // パーティクル位置の更新
    for (int it = 0; it < numMaxIteration; ++it) {
      Vector3 dv = targetPos - particles[0];
      particles[0] += dv;

      for (int pid = 0; pid < particles.Count - 1; ++pid) {
        dv = particles[pid + 1] - particles[pid];
        dv *= 0.5f - constraints[pid] / dv.Length() * 0.5f;
        particles[pid] += dv;
        particles[pid + 1] -= dv;
      }

      dv = rootPos - particles[particles.Count - 1];
      dv *= 1.0f - constraints[constraints.Count - 1] / dv.Length();
      particles[particles.Count - 1] += dv;
    }

    // パーティクルへのスケルトンの当てはめ
    FitFigureToParticle(skeleton, particles, joints);
    return particles.ToArray();
  }
```

パーティクル位置へのスケルトンの当てはめ処理は，CCD 法と同様の Closed-form な関節角度更新処理を，エフェクタからルートに向かって適用します．なお，この当てはめ処理は私が適当に実装したもので，オリジナルの論文とはまったく異なるものです．この実装では色々と不具合が生じますが，その改善は本題と外れるので省略します．

```
  /// <summary>パーティクル群へのスケルトンの当てはめ</summary>
  /// <param name="skeleton">スケルトン</param>
  /// <param name="particles">パーティクル位置</param>
  /// <param name="joints">関節リスト</param>
  static private void FitFigureToParticle(Skeleton skeleton, List particles, List joints)
  {
    for (int jid = joints.Count - 1; jid > 0; --jid) {
      // 注目関節の局所座標系行列の逆行列
      Matrix invCoord = Matrix.Invert(skeleton.GetGlobalCoordinate(joints[jid].Index));

      Vector3 p1 = Vector3.Transform(particles[jid - 1], invCoord);
      Vector3 p0 = Vector3.Transform(skeleton.GetGlobalPosition(joints[jid].Index), invCoord);
      Vector3 dv = Vector3.Normalize(p1 - p0);
      Vector3 offset = Vector3.Normalize(joints[jid - 1].Offset);

      float rotationAngle = (float)Math.Acos(Vector3.Dot(offset, dv));
      if (rotationAngle > 1.0e-5f) {
        Vector3 rotationAxis = Vector3.Normalize(Vector3.Cross(offset, dv));
        joints[jid].Rotation = Quaternion.CreateFromAxisAngle(rotationAxis, rotationAngle);
      }
    }
  }
```

PIK 法の長所と短所 [†]

PIK 法はとにかく計算が早いです．SolveParticle メソッドは 3 次元ベクトルの単純な算術演算のみで実装されており，順運動各計算はもちろん，三角関数等も 1 度も使われません．最終的なスケルトンの当てはめ処理では簡易な IK 計算が必要となりますが，全体の計算コストは非常に低くなります．また，反復計算の途中で適当にパーティクルを動かしてもよいので，任意の関節位置に「緩やかな」制約を与えることも可能です．

一方，PIK 法も CCD 法と同様に，回転可動域の設定と，位置と方向を同時に指定した計算が苦手です．特に前者は，関節の位置＝パーティクルの位置を操作するアルゴリズムであることから，とりあえず PIK 法で姿勢を計算して可動域を外れた関節回転量を適当に修正して再度 PIK 法で…という，発見的な方法となってしまいます．

まとめ [†]

反復計算を利用したインバースキネマティクスとして，代表的な Cyclic coordinate descent 法と比較的新しい Particle inverse kinematics 法を紹介しました．いずれも実装も簡単で処理も高速なので，利便性の高いアルゴリズムだと思います．いくつかの問題点もありますが，アプリケーションによっては問題とならない／実用的な解決策がある場合がほとんどでしょう．技術的にもまだまだ改善の余地がありますので，研究対象としても面白いと思います．

Last-modified: 2009-02-24 (火) 13:53:37 (1889d)

Site admin: cherub