ヤコビアンを用いた逆運動学

http://www.tmps.org/index.php?%A5%E4%A5%B3%A5%D3%A5%A2%A5%F3%A4%F2%CD%D1%A4%A4%A4%BF%B5%D5%B1%BF%C6%B0%B3%D8

ここでは，順運動学と対をなす技法である逆運動学について説明します． 逆運動学には様々な計算手法が提案されていますが，本稿ではヤコビアンを用いた数値解法を解説します

また，この記事の発展版として，クォータニオンを用いた逆運動学法についてもまとめています．ひとまずこの記事に目を通した後にクォータニオン逆運動学を参照ください．

逆運動学 [†]

前回解説したように，順運動学（FK）とは多関節リンク構造を構成する関節の回転角度から各リンクの位置や姿勢を求める計算です． これと逆に，特定のリンクの位置や方向を満たすような関節角度を求める計算を逆運動学計算（inverse kinematics. IK）といいます． 以下ではエフェクタの位置や方向からリンク構造を構成する全ての関節角度を計算する問題を考えることにします． エフェクタの位置ベクトル $p=(p_1\cdots p_M)$，リンク構造を構成する関節角度ベクトルを $\theta=(\theta_1\cdots\theta_N)$ とするとき， IK は FK の写像関数 f の逆写像関数によって定式化されます．

$$\text{順運動学} \quad : \quad p=f(\theta)$$
$$\text{逆運動学} \quad : \quad \theta=f^{-1}(p)$$

p は一般的に作業空間ベクトル（workspace vector）と呼ばれます． p にはエフェクタの位置ベクトルだけでなく，方向ベクトルを合わせた 6 次元ベクトルも与えられます．さらに，2 つのリンクの状態を同時に指定する場合などにはより次元数の大きなベクトルが与えられることもあり，操作の目的に応じて p の次元数 M は変化します．また，関節角度ベクトル θ の次元数はリンク構造の自由度 N に一致します．

ここで，作業空間の次元数 M とリンク構造の自由度 N の関係によって，IK の解の性質が変化することに注意する必要があります．

- $M=N$ の場合
  - 作業空間の次元数とリンク構造の自由度が一致する場合，FK，IK ともに解が一意に計算できます． つまり，ある関節角度ベクトルを与えるとエフェクタの位置が必ず 1 つだけ決定し， またある座標にエフェクタが位置するような関節角度ベクトルは必ず 1 つだけ存在します．したがって $M=N$ の場合の IK は解析的な手法によって解くことが可能です．
- $M<N$ の場合
  - 作業空間の次元数よりリンク構造の自由度が大きい場合，FK は解が一意に決定しますが， IK は解が無数に存在します． これを（作業空間に対する）リンク構造の冗長性（redundancy）と呼びます． 例えば，Fig.1 のようにルートとエフェクタの位置が固定されている場合でも， リンク構造は様々な姿勢をとることが可能です． リンク構造に冗長性が存在する場合，IK は解析的に計算できないので， ヤコビアンを用いた数値計算や，最適化計算，探索法等を用いて解くことになります．
- $M>N$ の場合
  - 作業空間の次元数がリンク構造の自由度より大きい場合，FK は解が一意に決定しますが作業空間より次元の低い空間しか扱えません． 例えば，1 リンクの向き（2 自由度）しか操作できない場合，エフェクタの位置は 3 次元で表されますが， 実際にはリンク長を半径とする球面上しか移動できません． また IK では解が存在しない場合があります．

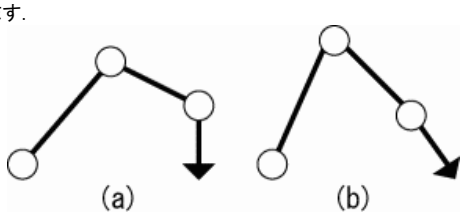


Fig.1 リンク構造の冗長性

[↑]

ヤコビアン [†]

リンク i のリンク長と初期方向を表す平行移動行列を T_i，関節 i の回転を表す回転行列を R_i とするとき， FKは次のように定式化されました．

$$p = f(\theta) = \oint T_N R(\theta_{N-1}) T_{N-1} \cdots R(\theta_1) T_1$$

上式のように，FKの写像関数 f は回転行列，すなわち非線形関数である三角関数により構成されています．作業空間の次元数 M とリンク構造の自由度 N が等しい場合は，連立三角方程式を解くことで解析的に逆写像関数が計算できますが，次元数が大きくなるに従って計算が指数的に複雑になります．また $M \neq N$ の場合には解が解析的に求まりません．

そこで数値計算によって解を求めることを考えます．まず，関節角度 θ_1 に微小角変位 $\triangle\theta_1$ を加えたときのエフェクタの微小変位 $\triangle p_1$ を求めます．

$$\begin{aligned}
p + \triangle p_1 &= f(\theta_1 + \triangle\theta_1, \theta_2, \cdots, \theta_N) \\
\triangle p_1 &= f(\theta_1 + \triangle\theta_1, \theta_2, \cdots, \theta_N) - f(\theta_1, \theta_2, \cdots, \theta_N) \\
&= \oint T_N R(\theta_{N-1}) T_{N-1} \cdots R(\theta_2) T_2 \left\{ R(\theta_1 + \triangle\theta_1) - R(\theta_1) \right\} T_1 \\
&\simeq \oint T_N R(\theta_{N-1}) T_{N-1} \cdots R(\theta_2) T_2 \left\{ \frac{dR(\theta_1)}{d\theta_1} \triangle\theta_1 \right\} T_1 \\
&= \frac{\partial f}{\partial \theta_1} \triangle\theta_1
\end{aligned}$$

上式のように，$\triangle p_1$ は(写像関数 f の θ_1 に関する偏微分 × 微小角変位 $\triangle\theta_1$ で近似されます．同様にして，関節 i の微小角変位 θ_i によるエフェクタ微小変位 $\triangle p_i$ は次のように近似されます．

$$\triangle p_i = \begin{bmatrix} \triangle p_{i,1} & \triangle p_{i,2} & \cdots & \triangle p_{i,M} \end{bmatrix} \simeq \begin{bmatrix} \left.\frac{\partial f}{\partial \theta_i}\right|_1 & \left.\frac{\partial f}{\partial \theta_i}\right|_2 & \cdots & \left.\frac{\partial f}{\partial \theta_i}\right|_M \end{bmatrix} \triangle\theta_i$$

さらに，以上の式を拡張することで，関節微小角変位ベクトル $\triangle\theta$ を加えたときのエフェクタ微小変位 $\triangle p$ を近似することができます．

$$\begin{aligned}
\triangle p &\simeq \frac{\partial f}{\partial \theta_1} \triangle\theta_1 + \frac{\partial f}{\partial \theta_2} \triangle\theta_2 + \cdots + \frac{\partial f}{\partial \theta_N} \triangle\theta_N \\
&= \begin{bmatrix} \triangle\theta_1 & \triangle\theta_2 & \cdots & \triangle\theta_N \end{bmatrix} \begin{bmatrix} \frac{\partial f}{\partial \theta_1} & \frac{\partial f}{\partial \theta_2} & \cdots & \frac{\partial f}{\partial \theta_N} \end{bmatrix}^T \\
&= \begin{bmatrix} \triangle\theta_1 & \triangle\theta_2 & \cdots & \triangle\theta_N \end{bmatrix} \begin{bmatrix} \left.\frac{\partial f}{\partial \theta_1}\right|_1 & \left.\frac{\partial f}{\partial \theta_1}\right|_2 & \cdots & \left.\frac{\partial f}{\partial \theta_1}\right|_M \\ \left.\frac{\partial f}{\partial \theta_2}\right|_1 & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \left.\frac{\partial f}{\partial \theta_N}\right|_1 & \cdots & \cdots & \left.\frac{\partial f}{\partial \theta_N}\right|_M \end{bmatrix} \\
&= \triangle\theta J
\end{aligned}$$

上式で導出された J がヤコビアン(Jacobian. ヤコビ行列)と呼ばれる行列で，関節微小角変位ベクトルからエフェクタ微小変位ベクトルへの写像を表します．このように，微小変位を扱う時の FK は線形演算によって計算されることがわかります．したがって，ヤコビアン J の逆変換となる行列 $J^\sharp$ を計算することによって，エフェクタ微小変位ベクトルから関節微小角変位ベクトルへの写像を計算することが可能です．

$$\begin{aligned}
\triangle\theta &= \triangle p J^\# \\
J^\# J &= I
\end{aligned}$$

$J^\sharp$ を用いた逆運動学計算の簡単な例として，与えられた目標位置に向かってエフェクタを直線的に移動させる場合の計算手順を以下に示します．

1. エフェクタの目標位置を p_G とする．
2. 現在のエフェクタの位置 p から p_G へ伸びる長さ $step$ のベクトルを計算し，目標エフェクタ変位ベクトル $\triangle p = (p_G - p)/|p_G - p| * step$ ($step$ は計算刻み幅)を計算する．
3. ヤコビアン J を計算し，その逆写像行列 $J^\sharp$ を計算する．
4. $\triangle\theta = \triangle p J^\sharp$ により関節角変位ベクトルを計算する．
5. $\theta \longleftarrow \theta + \triangle\theta$ により，関節角度ベクトルを更新する．
6. 更新後の $|p_G - p| \leq \epsilon$ (ϵ は許容誤差)ならば計算終了，それ以外の場合は 2. に戻る．

しかし前述のように，リンク構造の自由度と作業空間の次元数の関係によって $J^\sharp$ の性質は異なります．具体的には，ヤコビアン J の階数 $rank(J)$ がM以上ならば $J^\sharp$ を計算することが可能です．特に $N = M$ かつ $rank(J) = M$ ならば，$J^\sharp = J^{-1}$ となり，単純にヤコビアンの逆行列を計算することで $J^\sharp$ が求まります．一方，$rank(J) > M$ の場合は $J^\sharp$ は無数に存在するため一意に計算できないので，次節で説明する擬似逆行列のように任意の拘束条件下で解を計算することになります．

擬似逆行列，加重擬似逆行列

前述のように，$rank(J) > M$ の場合にはヤコビアン J の逆写像行列 $J^\sharp$ は無数に存在します．つまり，リンク構造に冗長性が存在する場合は，1 つのエフェクタ位置に対応するリンク構造の姿勢は無数に存在することになります．例えば，人間の腕は肩 3 自由度，肘 2 自由度，手首 2 自由度の計 7 自由度を持

つため，肩の位置と手の位置と方向(6 自由度)を固定した状態でも肘を移動させることができます．こうした冗長性を解決して $J^{\sharp}$ を一意的に計算するためには，任意の拘束条件を導入して解を計算する必要があります．その代表的な方法が，擬似逆行列(pseudoinverse)を用いる方法です．詳細な説明は省略しますが，ヤコビアンの擬似逆行列 J^{+} は これによって計算される関節角変位ベクトル $\theta = \triangle p J^{+}$ の自乗和 $\triangle\theta\triangle\theta^{T}$ を最小化するように一意に計算されます．こうして得られる解を(ヤコビアンの)擬似逆行列解(pseudoinverse solution)といいます．

$$J^{+} = (J^{T}J)^{-1}J^{T}$$

さらに，各関節の剛性(stiffness)，いわば動かしやすさを指定する場合には，加重擬似逆行列(weighted pseudoinverse)を用いることで関節角変位ベクトルの加重自乗和 $\theta W \theta^{T}$ を最小化するような解を計算できます．ここで，加重行列 W は $N \times N$ の正定値対称行列である必要があります．

$$J^{+} = (J^{T}W^{-1}J)^{-1}J^{T}W^{-1}$$

冗長変数 [†]

前節の擬似逆行列解を一般化した一般化擬似逆行列解が次の式で与えられます．

$$\triangle\theta = \triangle p J^{\#} + \eta(I - JJ^{\#})$$

ここで，η は冗長変数(redundant coefficients)と呼ばれる N 次元のベクトルです．前節で示した擬似逆行列解は一般化擬似逆行列解において $\eta = \vec{0}$ の場合の特殊解です．この冗長変数は上式の両辺に右から J を乗算するとわかるように，エフェクタの状態には影響しません．

$$\begin{aligned}\triangle\theta J &= \triangle p J^{\#}J + \eta(I - JJ^{\#})J \\ \triangle\theta J &= \triangle p I + \eta(J - JI) \\ \triangle\theta J &= \triangle p\end{aligned}$$

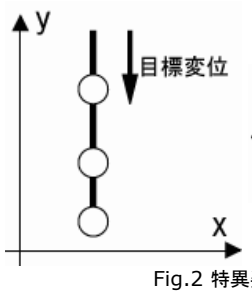
また，$rank(J) = M$ の場合には $JJ^{\sharp} = JJ^{-1} = I$ となり，冗長変数の項そのものが消失します．つまり冗長変数は，リンク構造の冗長性を利用して，エフェクタの状態を保ちながら関節角度を操作するための変数だといえます．

擬似逆行列解の問題点 [†]

擬似逆行列解は IK の代表的な数値解法ですが，ロボット工学分野で発展した経緯もあり，キャラクタアニメーションに応用する場合にはいくつかの問題もあります．

特異姿勢 [†]

$J^{\sharp}$ が存在する条件は前述の通り $rank(J) \leq M$ ですが，$rank(J) < M$ つまりランク落ちが発生する姿勢を特異姿勢(singular posture)と呼びます．例えば，Fig.2 に示すような 2 次元上の 3 自由度多関節リンク構造のヤコビアンを計算すると，第 1 列の成分はそれぞれ 0 以外の値をとります．一方，第 2 列の成分は全て 0 になりランク落ちとなります．直観的にみても Fig.2 の姿勢で各関節に微小角変位を与えても，エフェクタは x 軸方向には大きく移動しますが，y 軸方向にはほとんど移動しない状態にあることがわかると思います．特異姿勢では逆運動学の解が存在しないので，計算において特異姿勢を回避するような工夫が必要となります．



$$J = \begin{bmatrix} * & 0 \\ * & 0 \\ * & 0 \end{bmatrix}$$

Fig.2 特異姿勢

また，特異姿勢付近では J のいずれかの列の成分が小さくなり，$J^{\sharp}$ の対応する成分が非常に大きな値をとります．したがって Fig.2 に近い姿勢において同図のような目標変位を与えると，非常に大きな関節角変位ベクトルが出力されるため動作が不自然になるという問題もあります．そのため，ロボット工学の分野では特異点付近でも自然な関節角変位ベクトルを与えるような $J^{\sharp}$ を計算する方法も提案されています．

関節可動域 [†]

擬似逆行列解の計算手順を見るとわかるように，計算はあくまで幾何学的な数値計算であり，キャラクタやロボットの関節可動域を全く考慮したものではありません．したがって，キャラクタの姿勢によっては肘が逆に曲がるなど不自然な動作がしばしば発生します．そのため，冗長変数を制御することで関節可動域を満足する解を計算する手法も提案されています．

計算量 [†]

擬似逆行列解はあくまでも数値計算なので，解析的な方法と比べて当然計算量が大きくなりがちです．特にヤコビアンの計算には $O(N^2)$ の行列乗算演算が必要であり，大きなボトルネックとなります．計算刻み幅 $step$ を調整することである程度の高速化は可能ですが，これは計算誤差とのトレードオフになります．

DirectX Graphicsにおける実装例

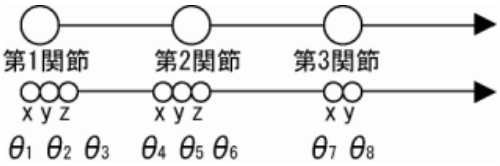
今回のサンプルは，多関節リンク構造のエフェクタが指定された目標点に向かって直線的に移動するようなプログラムです．なおサンプルでは一般化加重擬似逆行列解を実装していますが，加重行列 W は常に単位行列 I，冗長変数 η はゼロベクトル $\vec{0}$ としています．

サンプルではヤコビアンを計算するために GNU Scientific Library を用いています． 実際にはそれほど難しい計算ではありませんが，慣れたライブラリに頼って手抜きしました...）GSL が導入されていない環境ではコンパイルできませんが，関数名やコメントを参照しつつ，各自のライブラリに置き換えていただければ，と思います． また，アルゴリズムとは直接関係のない部分は説明を割愛させてもらいますので， 詳細はソースコード中のコメント等を参照してください．

サンプルプロジェクトは以下になります（プロジェクト名：JacIK）．ビルドするためには Summer2004 以降の DirectX9.0 SDK と GSL がインストールされている必要になります．実行後 [Test] メニューから [IK] を選択するとダイアログが出現し，目標位置を入力するとその点に向かってエフェクタが直線的に移動します．エフェクタが原点に移動するアニメーションをキャプチャした MPEG ムービーも置いておきます（途中画面が数回ブレれます）．

VC++.NET2003プロジェクトファイル(35Kb)
サンプルMPEGムービー(55Kb)

リンクモデル



サンプルで用いたリンク構造モデルは Fig.3 に示すように 3 関節 8 自由度を持ちます．全てのリンクは初期状態で Z 軸沿いの長さ 1 のリンク $p_l = (0\ 0\ 1)$ です．よって，エフェクタの位置を求める FK の式は次のように表されます．

$$p_e = \vec{0} T_3 R_y(\theta_8) R_x(\theta_7) T_2 R_z(\theta_6) R_y(\theta_5) R_x(\theta_4) T_1 R_z(\theta_3) R_y(\theta_2) R_x(\theta_1)$$

上式に対応した C++ コードは次の通りです．

```
"MainDoc.cpp"

D3DXVECTOR3 GetEffectorPosition(gsl_vector *pvTheta)
{
  D3DXMATRIX mTmp;
  // FKによりエフェクタ位置を計算
  mTmp = OffsetZ(1.0) * RotY( gsl_vector_get(pvTheta, 7) ) * RotX( gsl_vector_get(pvTheta, 6) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 5) ) * RotY( gsl_vector_get(pvTheta, 4) ) * RotX( gsl_vector_get(pvTheta, 3) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 2) ) * RotY( gsl_vector_get(pvTheta, 1) ) * RotX( gsl_vector_get(pvTheta, 0) );

  // ゼロベクトルとの乗算を省略し，平行移動項を取得
  return D3DXVECTOR3(mTmp(3, 0), mTmp(3, 1), mTmp(3, 2));
}
```

pvTheta が関節角度ベクトル θ に対応し，OffsetZ 関数は Z 軸沿いの平行移動移動行列 T_i を作成する関数，RorX，RotY，RotZ はそれぞれ XYZ 軸まわりの回転行列を作成する関数になります． また，gsl_vector_get 関数はベクトルの成分を取得する GSL の関数です．コードに示すように，ゼロベクトルとの乗算は合成トランスフォームの平行移動項を取得することに等しいです． ベクトル×行列演算のほうが計算量は有利ですが，ここではコードの見通しを優先しています）．

ヤコビアンの計算

今回のサンプルの要となるのがヤコビアンの計算部分です． ここで改めてヤコビアンの定義式を示します．

$$J = \begin{bmatrix} \frac{\partial f}{\partial \theta_1} \\ \frac{\partial f}{\partial \theta_2} \\ \vdots \\ \frac{\partial f}{\partial \theta_N} \end{bmatrix} = \begin{bmatrix} \left.\frac{\partial f}{\partial \theta_1}\right|_1 & \left.\frac{\partial f}{\partial \theta_1}\right|_2 & \cdots & \left.\frac{\partial f}{\partial \theta_1}\right|_M \\ \left.\frac{\partial f}{\partial \theta_2}\right|_1 & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \left.\frac{\partial f}{\partial \theta_N}\right|_1 & \cdots & \cdots & \left.\frac{\partial f}{\partial \theta_N}\right|_M \end{bmatrix}$$

$$\frac{\partial f}{\partial \theta_i} = \vec{0} T_N R(\theta_{N-1}) T_{N-1} \cdots \left\{ \frac{dR(\theta_i)}{d\theta_i} \right\} \cdots R(\theta_1) T_1$$

ヤコビアンは上段の式に示すように，FK の写像関数 f の関節角に関する偏微分を行ベクトルとした行列です．そして下段の式に示すように，関節角度 θ_i に関する偏微分は，f の計算式における回転行列 $R(\theta_i)$ をその微分行列に置き換えることで計算されます．XYZ 各軸周りの回転行列を微分した行列を以下に示します．

$$\frac{d}{d\theta}R_x(\theta) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & -\cos(\theta) & -\sin(\theta) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\frac{d}{d\theta}R_y(\theta) = \begin{bmatrix} -\sin(\theta) & 0 & -\cos(\theta) & 0 \\ 0 & 0 & 0 & 0 \\ \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\frac{d}{d\theta}R_z(\theta) = \begin{bmatrix} -\sin(\theta) & \cos(\theta) & 0 & 0 \\ -\cos(\theta) & -\sin(\theta) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

以上に示したヤコビアン計算式の実装例は次の通りです.

```
"MainDoc.cpp"

gsl_matrix* CMainDoc::ComputeJacobian(const gsl_vector *pvTheta) const
{
  gsl_matrix *pmJac;

  // ヤコビアンは8×3（関節角度ベクトル長×3次元位置ベクトル長）の行列
  pmJac = gsl_matrix_alloc(pvTheta->size, 3);
  ASSERT(pmJac);

  D3DXMATRIX mTmp, mBack;

  // 第3リンクのオフセット行列
  mBack = OffsetZ(1.0);

  // 第8関節に関する偏微分
  mTmp = mBack * DiffY( gsl_vector_get(pvTheta, 7) ) * RotX( gsl_vector_get(pvTheta, 6) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 5) ) * RotY( gsl_vector_get(pvTheta, 4) ) * RotX( gsl_vector_get(pvTheta, 3) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 2) ) * RotY( gsl_vector_get(pvTheta, 1) ) * RotX( gsl_vector_get(pvTheta, 0) );
  mBack *= RotY( gsl_vector_get(pvTheta, 7) );
  gsl_matrix_set(pmJac, 7, 0, mTmp(3, 0));
  gsl_matrix_set(pmJac, 7, 1, mTmp(3, 1));
  gsl_matrix_set(pmJac, 7, 2, mTmp(3, 2));

  // 第7関節に関する偏微分
  mTmp = mBack * DiffX( gsl_vector_get(pvTheta, 6) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 5) ) * RotY( gsl_vector_get(pvTheta, 4) ) * RotX( gsl_vector_get(pvTheta, 3) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 2) ) * RotY( gsl_vector_get(pvTheta, 1) ) * RotX( gsl_vector_get(pvTheta, 0) );
  mBack *= RotX( gsl_vector_get(pvTheta, 6) ) * OffsetZ(1.0);
  gsl_matrix_set(pmJac, 6, 0, mTmp(3, 0));
  gsl_matrix_set(pmJac, 6, 1, mTmp(3, 1));
  gsl_matrix_set(pmJac, 6, 2, mTmp(3, 2));

  // 第6関節に関する偏微分
  mTmp = mBack * DiffZ( gsl_vector_get(pvTheta, 5) ) * RotY( gsl_vector_get(pvTheta, 4) ) * RotX( gsl_vector_get(pvTheta, 3) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 2) ) * RotY( gsl_vector_get(pvTheta, 1) ) * RotX( gsl_vector_get(pvTheta, 0) );
  mBack *= RotZ( gsl_vector_get(pvTheta, 5) );
  gsl_matrix_set(pmJac, 5, 0, mTmp(3, 0));
  gsl_matrix_set(pmJac, 5, 1, mTmp(3, 1));
  gsl_matrix_set(pmJac, 5, 2, mTmp(3, 2));

  // 第5関節に関する偏微分
  mTmp = mBack * DiffY( gsl_vector_get(pvTheta, 4) ) * RotX( gsl_vector_get(pvTheta, 3) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 2) ) * RotY( gsl_vector_get(pvTheta, 1) ) * RotX( gsl_vector_get(pvTheta, 0) );
  mBack *= RotY( gsl_vector_get(pvTheta, 4) );
  gsl_matrix_set(pmJac, 4, 0, mTmp(3, 0));
  gsl_matrix_set(pmJac, 4, 1, mTmp(3, 1));
  gsl_matrix_set(pmJac, 4, 2, mTmp(3, 2));

  // 第4関節に関する偏微分
  mTmp = mBack * DiffX( gsl_vector_get(pvTheta, 3) )
    * OffsetZ(1.0) * RotZ( gsl_vector_get(pvTheta, 2) ) * RotY( gsl_vector_get(pvTheta, 1) ) * RotX( gsl_vector_get(pvTheta, 0) );
  mBack *= RotX( gsl_vector_get(pvTheta, 3) ) * OffsetZ(1.0);
  gsl_matrix_set(pmJac, 3, 0, mTmp(3, 0));
  gsl_matrix_set(pmJac, 3, 1, mTmp(3, 1));
  gsl_matrix_set(pmJac, 3, 2, mTmp(3, 2));

  // 第3関節に関する偏微分
  mTmp = mBack * DiffZ( gsl_vector_get(pvTheta, 2) ) * RotY( gsl_vector_get(pvTheta, 1) ) * RotX( gsl_vector_get(pvTheta, 0) );
  mBack *= RotZ( gsl_vector_get(pvTheta, 2) );
  gsl_matrix_set(pmJac, 2, 0, mTmp(3, 0));
  gsl_matrix_set(pmJac, 2, 1, mTmp(3, 1));
  gsl_matrix_set(pmJac, 2, 2, mTmp(3, 2));

  // 第2関節に関する偏微分
```

```
66 !    mTmp = mBack * DiffY( gsl_vector_get(pvTheta, 1) ) * RotX( gsl_vector_get(pvTheta, 0) );
67 |    mBack *= RotY( gsl_vector_get(pvTheta, 1) );
68 |    gsl_matrix_set(pmJac, 1, 0, mTmp(3, 0));
69 |    gsl_matrix_set(pmJac, 1, 1, mTmp(3, 1));
70 |    gsl_matrix_set(pmJac, 1, 2, mTmp(3, 2));
71 |
72 |    // 第1関節に関する偏微分
73 !    mTmp = mBack * DiffX( gsl_vector_get(pvTheta, 0) );
74 |    gsl_matrix_set(pmJac, 0, 0, mTmp(3, 0));
75 |    gsl_matrix_set(pmJac, 0, 1, mTmp(3, 1));
76 |    gsl_matrix_set(pmJac, 0, 2, mTmp(3, 2));
77 |
78 |    return pmJac;
79 ! }
```

ここで, DiffX, DiffY, DiffZ 関数はそれぞれ XYZ 軸周りの回転行列の微分行列を取得する関数です. 余分な乗算演算を省くために作業用行列 mBack 行列を導入していますが, 基本的には FK の式から導出されていることが確認できると思います.

加重擬似逆行列, 冗長項の計算 [†]

擬似逆行列や冗長変数の節で示した計算式にしたがい, GSL の関数によって実装しています.

```
1    "MainDoc.cpp"
2
3  // 擬似逆行列
4  ! gsl_matrix* CMainDoc::ComputePseudoInverse(const gsl_matrix *pmJac) const
5  ! {
6  !   // J^T * J
7  !   gsl_matrix *pmJtJ;
8  |   pmJtJ = gsl_matrix_alloc(pmJac->size2, pmJac->size2);
9  |   ASSERT(pmJtJ);
10 |   gsl_blas_dgemm(CblasTrans, CblasNoTrans, 1.0, pmJac, pmJac, 0.0, pmJtJ);
11 |
12 |   // (J^T * J)^-1
13 !   gsl_matrix *pmJtJi;
14 |   pmJtJi = MatrixInverse(pmJtJ);
15 |
16 |   // (J^T * J)^-1 * J^T
17 !   gsl_matrix *pmPI;
18 |   pmPI = gsl_matrix_alloc(pmJtJi->size1, pmJac->size1);
19 |   ASSERT(pmPI);
20 |   gsl_blas_dgemm(CblasNoTrans, CblasTrans, 1.0, pmJtJi, pmJac, 0.0, pmPI);
21 |
22 |   gsl_matrix_free(pmJtJ);
23 |   gsl_matrix_free(pmJtJi);
24 |
25 |   return pmPI;
26 ! }
27
28 // 加重擬似逆行列
29 ! gsl_matrix* CMainDoc::ComputeWeightedPseudoInverse(const gsl_matrix *pmJac, const gsl_matrix *pmWeight) const
30 ! {
31 !   // W^-1
32 !   gsl_matrix *pmWI;
33 |   pmWI = MatrixInverse(pmWeight);
34 |
35 |   // J^T * W^-1
36 !   gsl_matrix *pmJtWi;
37 |   pmJtWi = gsl_matrix_alloc(pmJac->size2, pmWI->size2);
38 |   ASSERT(pmJtWi);
39 |   gsl_blas_dgemm(CblasTrans, CblasNoTrans, 1.0, pmJac, pmWI, 0.0, pmJtWi);
40 |
41 |   // J^T * W^-1 * J
42 !   gsl_matrix *pmJtWiJ;
43 |   pmJtWiJ = gsl_matrix_alloc(pmJtWi->size1, pmJac->size2);
44 |   ASSERT(pmJtWiJ);
45 |   gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, pmJtWi, pmJac, 0.0, pmJtWiJ);
46 |
47 |   // (J^T * W^-1 * J)^-1
48 !   gsl_matrix *pmJtWiJi;
49 |   pmJtWiJi = MatrixInverse(pmJtWiJ);
50 |
51 |   // (J^T * W^-1 * J)^-1 * J^T
52 !   gsl_matrix *pmJtWiJiJt;
53 |   pmJtWiJiJt = gsl_matrix_alloc(pmJtWiJi->size1, pmJac->size1);
54 |   ASSERT(pmJtWiJiJt);
55 |   gsl_blas_dgemm(CblasNoTrans, CblasTrans, 1.0, pmJtWiJi, pmJac, 0.0, pmJtWiJiJt);
56 |
57 |   // (J^T * W^-1 * J)^-1 * J^T * W^-1
58 !   gsl_matrix *pmWPI;
59 |   pmWPI = gsl_matrix_alloc(pmJtWiJiJt->size1, pmWI->size2);
60 |   ASSERT(pmWPI);
61 |   gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, pmJtWiJiJt, pmWI, 0.0, pmWPI);
62 |
63 |   gsl_matrix_free(pmJtWi);
64 |   gsl_matrix_free(pmJtWiJ);
65 |   gsl_matrix_free(pmJtWiJi);
66 |   gsl_matrix_free(pmJtWiJiJt);
67 |
```

```
68 |    return pmWPI;
69 ! }
70 |
71 // 冗長項
72 ! gsl_vector* CMainDoc::ComputeRedundantCoefficients(const gsl_vector *pvEta, const gsl_matrix *pmJac,
73 |      const gsl_matrix *pmJacPI) const
74 {
75 |   // η * J
76 !   gsl_vector *pvTmp;
77 |   pvTmp = gsl_vector_alloc(pmJac->size2);
78 |   ASSERT(pvTmp);
79 |   gsl_blas_dgemv(CblasTrans, 1.0, pmJac, pvEta, 0.0, pvTmp);
80 |
81 |   // vRC = η - η * J * J^+
82 !   gsl_vector *pvRC;
83 |   pvRC = gsl_vector_alloc(pmJacPI->size2);
84 |   ASSERT(pvRC);
85 |   gsl_vector_memcpy(pvRC, pvEta);
86 |   gsl_blas_dgemv(CblasTrans, -1.0, pmJacPI, pvTmp, 1.0, pvRC);
87 |
88 |   gsl_vector_free(pvTmp);
89 |
90 |   return pvRC;
91 ! }
```

関節角度ベクトルの更新処理 [†]

目標エフェクタ変位ベクトルとヤコビアンの加重擬似逆行列の乗算結果と冗長項を，元の関節角度ベクトルに加えて更新する処理です．

```
1   "MainDoc.cpp"
2
3   bool CMainDoc::UpdateAngles(const gsl_vector *pvDir, double dStep) const
4   {
5   |   gsl_matrix *pmJac, *pmJacPI, *pmJacWPI;
6   |
7   |   // 加重行列を単位行列で初期化
8   !   gsl_matrix *pmWeight;
9   |   pmWeight = gsl_matrix_alloc(m_pvTheta->size, m_pvTheta->size);
10  |   gsl_matrix_set_identity(pmWeight);
11  |
12  |   // ヤコビアンの計算
13  !   pmJac = ComputeJacobian(m_pvTheta);
14  |   // ヤコビアンの擬似逆行列
15  !   pmJacPI = ComputePseudoInverse(pmJac);
16  |   // ヤコビアンの加重擬似逆行列
17  !   pmJacWPI = ComputeWeightedPseudoInverse(pmJac, pmWeight);
18  |   // 目標エフェクタ変位×加重擬似逆行列
19  !   gsl_blas_dgemv(CblasTrans, 1.0, pmJacWPI, pvDir, 1.0, m_pvTheta);
20  |
21  |   gsl_vector *pvEta, *pvRC;
22  |   pvEta = gsl_vector_alloc(m_pvTheta->size);
23  |   // 冗長変数etaをゼロクリア
24  !   gsl_vector_set_all(pvEta, 0.0);
25  |   // 冗長項の計算
26  !   pvRC = ComputeRedundantCoefficients(pvEta, pmJac, pmJacPI);
27  |   // 冗長項を関節角度ベクトルに加える
28  !   gsl_vector_add(m_pvTheta, pvRC);
29  |
30  |   gsl_vector_free(pvRC);
31  |   gsl_vector_free(pvEta);
32  |   gsl_matrix_free(pmJacPI);
33  |   gsl_matrix_free(pmJacWPI);
34  |   gsl_matrix_free(pmJac);
35  |   gsl_matrix_free(pmWeight);
36  |
37  |   return true;
38  ! }
```

エフェクタの目標位置への到達アニメーション生成 [†]

エフェクタの目標位置への直線的な到達アルゴリズムに従い，ダイアログから入力された目標位置へエフェクタが一致するまで関節角度ベクトルを繰り返し更新します．ただ，ループ中の特異姿勢はチェックしていませんので，計算が発散して無限ループになる場合もあるかも知れません...

```
1   "MainDoc.cpp"
2
3   void CMainDoc::OnTestIK()
4   {
5   |   D3DXVECTOR3 vTarget, vPos, vDisp;
6   |   const double dStep = 0.01;
7   |   float fDist;
8   |
9   |   // ダイアログ入力から目標位置を取得
10  !   CTargetDlg dlg;
11  |   if (dlg.DoModal() != IDOK)
12  |     return;
13  |   vTarget.x = dlg.m_fX;
14  |   vTarget.y = dlg.m_fY;
```

```
15 |    vTarget.z = dlg.m_fZ;
16 |
17 |    // リンク構造の全長より遠い位置は到達不可能
18 !    if (D3DXVec3Length(&vTarget) >= 3.0f)
19 |    {
20 |      MessageBox(NULL, "Out of workspace", NULL, 0);
21 |      return;
22 !    }
23 |
24 |    // D3DXVECTOR3から値を受け取るGSLベクトル構造体
25 !    gsl_vector *pvDir;
26 |    pvDir = gsl_vector_alloc(3);
27 |    ASSERT(pvDir);
28 |
29 |    while (1)
30 |    {
31 |      // 現在のエフェクタの位置を取得
32 !      vPos = GetEffectorPosition(m_pvTheta);
33 |      // 目標位置とエフェクタ位置の差分の計算
34 !      vDisp = vTarget - vPos;
35 |      // 差分が(step / 2)より小さければ計算終了
36 !      if (fDist = D3DXVec3Length(&vDisp), fDist < dStep / 2.0)
37 |        break;
38 |
39 |      // 目標エフェクタ変位 = (差分ベクトル / 差分ベクトル長) * step
40 !      vDisp *= static_cast<float>(dStep) / fDist;
41 |      gsl_vector_set(pvDir, 0, vDisp.x);
42 |      gsl_vector_set(pvDir, 1, vDisp.y);
43 |      gsl_vector_set(pvDir, 2, vDisp.z);
44 |
45 |      // 目標エフェクタ変位にしたがって関節角度ベクトルを更新
46 !      UpdateAngles(pvDir, dStep);
47 |      // 再描画
48 !      UpdateAllViews(NULL);
49 !    }
50 |
51 |    gsl_vector_free(pvDir);
52 ! }
```

まとめ

今回は IK の導入偏として，リンク構造の冗長性に起因する IK の問題点と，IK の数値解法の 1 つであるヤコビアンの擬似逆行列解を解説してみました．これらのさらに詳しい解説は，運動計画(motion planning)をキーワードにロボット工学の書籍等を参照してみてください．特に，さらに踏み込んだIKに興味のある方には，東京大学の山根先生のメカトロ演習の講義ノートをお勧めします．SEGA アニマニウムのIKエンジンであるUT-Poserの片鱗に触れることができます．

ただ，擬似逆行列解はロボットアームのような機械的なキャラクタのアニメーション生成には効果的ですが，人体など有機的なキャラクタへの適用は非常に難しいと思います．加重行列や冗長変数である程度の調整は可能ですが直観的とは言いがたく，そんなややこしいことをするなら FK でモーション付けしたほうが早い場合が多いと思います．実際の制作ではあらかじめ FK で大まかにモーション付けし，IK でエフェクタの位置を補正する，といった使い方になるのではないでしょうか？
