クォータニオン逆運動学

http://www.tmps.org/index.php?%A5%AF%A5%A9%A1%BC%A5%BF%A5%CB%A5%AA%A5%F3%B5%D5%B1%BF%C6%B0%B3%D8

CGキャラクタアニメーションに関連するプログラム作成や研究に携わっていると，ほぼ間違いなくクォータニオン（Quaternion）という回転表現法を目にします．さらに，球面線形補間などクォータニオンの応用法を知る過程で，「クォータニオンを使ったインバースキネマティクスを実現できないだろうか？」という興味を抱くこともあると思います．

本記事ではその一つの解として，対数クォータニオンを用いた数値的逆運動学法（インバースキネマティクス，Inverse Kinematics，IK）を解説します．

ヤコビアンを用いた逆運動学 では，回転行列のオイラー角表現を用いた IK を紹介しました．その方法では，多リンク系の各関節の微小角変位量とエフェクタの変位量との関係を表すヤコビアン行列を導出し，その逆写像によってエフェクタを任意方向に微小移動させるための関節角変位量を算出しました．

一方，今回の方法では，本質的な処理手順はそのままに，回転行列のオイラー角表現（パラメータ数3）の替わりに対数クォータニオン（同じくパラメータ数3）を利用します．つまり，対数空間に写像されたクォータニオンのパラメータ変化とエフェクタの微小変位量との関係を表すヤコビアン行列を導出し，その逆写像によってエフェクタを任意方向に微小移動させるためのクォータニオンの変化量を算出します．

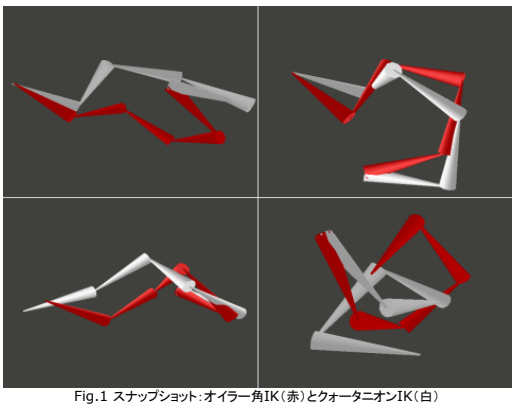

Fig.1 スナップショット：オイラー角IK（赤）とクォータニオンIK（白）

参考文献

- F Sebastian Grassia, "Practical Parameterization of Rotations Using the Exponential Map", Journal of Graphics Tools, Vol.3, No.3, pp.29-48, 1998.
  - 対数クォータニオンについてはこの論文を参照ください．SIGGRAPH 論文などで「exponential maps」「exp maps」といえば，間違いなくこの論文を指しています．
- 3D空間における回転の表現形式
  - あわせてこちらもご参照ください．対数クォータニオンの性質について簡単に言及しています．

サンプルプログラム

サンプルプログラムは，VisualC++ 2005 + DirectX SDK Update December 2006 の環境下で MFC と GSL1.9 を用いて作成し，WinVista(64bit)上でのみ実行テストしています．ただし，バイナリデータは32bit版です．

VC++2005プロジェクト(50kb)
サンプルバイナリ(289kb)

プログラムを実行すると，5 つのリンクを持つアームが表示されます．メニューの [テスト]→[View] を実行すると，エフェクタが適当な直線軌道上を往復するように，白と赤のリンクアームが運動するはずです．ここで，赤リンクはオイラー角度を利用した IK，白リンクがクォータニオン IK を利用した結果です．[テスト][View] を実行するたびに軌道がランダムに変化しますので，紅白のリンクの運動を比較してみてください．

リンクアームの作成とフォワードキネマティクス

インバースキネマティクスの説明に入る前に，リンクアームの作成，描画，そしてフォワードキネマティクスによるエフェクタ位置の算出方法をまとめておきます．

まず，リンクアームの姿勢情報を格納するメンバ変数を次のコードに示します．NUM_LINKS のリンク数を持つアームについて，各リンクの長さと初期方向を指定するオフセットベクトルを m_pOffset 配列に格納します．リンク間の関節回転行列を格納する配列は二種類用意し，それぞれオイラー角 IK とクォータニオン IK による計算姿勢を格納するため利用します．

```
  //! オフセットベクトル
  D3DXVECTOR3 m_pOffset[NUM_LINKS];
  //! 姿勢ベクトル
  D3DXMATRIX m_pEulerPose[NUM_LINKS];
  D3DXMATRIX m_pQuatPose[NUM_LINKS];
```

上記の変数はコンストラクタ内で初期化します．全ての関節回転行列は単位行列に，オフセットベクトルは Z 方向に伸びる単位ベクトルとしました．

```
// CScneView::CSceneView() コンストラクタ内
  ・・・・・・

  for (size_t i = 0;i < NUM_LINKS;++i) {
    D3DXMatrixIdentity(&m_pEulerPose[i]);
    D3DXMatrixIdentity(&m_pQuatPose[i]);
    m_pOffset[i] = D3DXVECTOR3(0, 0, 1.0f);
  }
```

次に，指定された関節回転行列をもとにエフェクタ位置を計算する，フォワードキネマティクス関数を示します．リンクアームのルートから先端に向かって，トランスフォーム行列を加算するだけのコードです．ただし，オフセットベクトルからオフセット行列を作成，乗算する部分を，無理やりなキャストを使ってベクトルの加算に置き換えています．返り値についても同様に，ゼロベクトルとの乗算を無理やりキャストによって簡略化しています．

```
D3DXVECTOR3 CSceneView::EffectorPos(const D3DXMATRIX pose[])
{
```

```
    D3DXMATRIX m, om;
    D3DXMatrixIdentity(&m);
    for (int i = NUM_LINKS - 1;i >= 0;--i) {
      D3DXMatrixIdentity(&om);
      *reinterpret_cast(&om._41) = m_pOffset[i];
      m *= om * pose[i];
    }
    return *reinterpret_cast(&m._41);
}
```

最後に，リンクアームの描画関数を示します．オイラー角 IK，クォータニオン IK の結果を同時に描画します．

```
HRESULT CSceneView::DrawLinks(void)
{
    D3DXMATRIX m, mt;

    m_pEffect->SetTechnique(m_hTechPhong);
    m_pEffect->SetVector("g_vAmbiColor", &D3DXVECTOR4(1.0f, 1.0f, 1.0f, 1.0f));

    // オイラー角 IK による結果
    m_pEffect->SetVector("g_vDiffColor", &D3DXVECTOR4(0.6f, 0.0f, 0.0f, 1.0f));
    D3DXMatrixIdentity(&m);
    for (size_t i = 0;i < NUM_LINKS;++i) {
      D3DXMatrixTranslation(&mt, m_pOffset[i].x / 2.0f, m_pOffset[i].y / 2.0f, m_pOffset[i].z / 2.0f);

      m = mt * m_pEulerPose[i] * m;
      DrawMeshSub(m_pConeMesh, 0, m);
      m = mt * m;
    }

    // クォータニオン IK による結果
    m_pEffect->SetVector("g_vDiffColor", &D3DXVECTOR4(0.6f, 0.6f, 0.6f, 1.0f));
    D3DXMatrixIdentity(&m);
    for (size_t i = 0;i < NUM_LINKS;++i) {
      D3DXMatrixTranslation(&mt, m_pOffset[i].x / 2.0f, m_pOffset[i].y / 2.0f, m_pOffset[i].z / 2.0f);

      m = mt * m_pQuatPose[i] * m;
      DrawMeshSub(m_pConeMesh, 0, m);
      m = mt * m;
    }

    return S_OK;
}
```

ヤコビアン行列を用いたインバースキネマティクス

数値計算による逆運動学計算法では，エフェクタの位置や方向⇔関節回転量の関係を表すヤコビアン行列が用いられます．また，ヤコビアンを用いた逆運動学で解説したように，関節回転量を表すパラメータにはオイラー角表現が広く利用されています．これは，逆運動学計算法が開発されたロボット工学分野において，実ロボット制御と相性が良いオイラー角表現が広く利用されている点に起因していると思われます．しかし，CGキャラクタは各関節自由度をモータ駆動させる必要もないので，実際にはオイラー角以外の様々な3次元回転量表現を使えるはずです．そこで，まずヤコビアン行列の計算方法を任意の回転パラメータ表現について一般化します．

多リンクアームを構成する N 自由度の関節回転量を θ，そのときのアーム先端(エフェクタ)位置を p とし，微小回転量 $\triangle\theta$ を加えた時のエフェクタ位置の微小変位を $\triangle p$ と表します．ここで，前記事では，関節回転量 θ をオイラー角の各パラメータで構成しました．このとき，オイラー角の微小回転量 $\triangle\theta$ と エフェクタ微小変位 $\triangle p$ との関係は次の式で表されます．

$$\triangle p = \triangle\theta J$$

ここで行列 J は，$\triangle\theta$ から $\triangle p$ への写像を与えるヤコビアン行列と呼ばれる行列で，各オイラー角パラメータに関するエフェクタ変位ベクトル各成分の偏微分係数で構成されます．

$$J = \begin{bmatrix} \frac{\partial p_x}{\partial \theta_1} & \frac{\partial p_y}{\partial \theta_1} & \frac{\partial p_z}{\partial \theta_1} \\ \frac{\partial p_x}{\partial \theta_2} & \frac{\partial p_y}{\partial \theta_2} & \frac{\partial p_z}{\partial \theta_2} \\ \vdots & \vdots & \vdots \\ \frac{\partial p_x}{\partial \theta_N} & \frac{\partial p_y}{\partial \theta_N} & \frac{\partial p_z}{\partial \theta_N} \end{bmatrix}$$

繰り返しになりますが，関節の回転量 θ が特にオイラー角である必要はありません．つまり，ヤコビアン行列 J の計算に支障がなければ，θ の要素がクォータニオンの 4 つのパラメータでも，Axis-Angle 表現の 4 つのパラメータでも問題ないはずです．すなわち θ の各要素が回転量を表すパラメータである限り，上式は任意の回転量表現について一般化できます．ここで，「ヤコビアン行列 J の計算に支障がなければ」という条件は，「θ の各要素が互いに独立である」という制約に対応します．つまり，θ_1 の変化によって θ_2 や θ_3 の値が変化してはならないという制約です．これは，パラメータが従属関係にあると，ヤコビアン行列各要素の偏微分が計算できないことを意味します．例えば，関数 $F(u\ v)$の偏微分において u と v が互いに従属だと，合成関数の偏微分の定義から $\partial F/\partial u = \partial F/\partial v \cdot \partial v/\partial u = \cdots = \partial F/\partial u \cdot \partial u/\partial v \cdot \partial v/\partial u = \cdots$と堂々巡りになることから明らかです．

オイラー角表現は全てのパラメータが独立なので，ヤコビアン行列を計算できます．しかし，クォータニオンや Axis-Angle 表現はこの仮定を満たしません．これは，回転量を表すクォータニオンは単位クォータニオンでなければならない点や，Axis-Angle 表現における回転軸ベクトルが単位ベクトルでなければならない点に原因があります．例えば，回転量ゼロのクォータニオン $q = [q_x\ q_y\ q_z\ q_w] = [0\ 0\ 0\ 1.0]$ に微小変化 q_x が加わり，$[0.1\ 0\ 0\ 1.0]$になった場合を考えます．加算後のクォータニオンは $|q| = 1.0$ を満たさず，単位クォータニオンではないので3次元回転量を表しません．したがって，$|q| = 1.0$ となるように正規化する必要がありますが，その結果は $[0.099\cdots\ 0\ 0\ 0.99\cdots]$ となります．ここで，q_w の値が変化しているのはもちろん，操作した q_x の値そのものも減少してしまいます．この問題は Axis-Angle 表現の回転軸ベクトルについても同様に発生します．このように，クォータニオンや Axis-Angle 表現のパラメータは互いに従属であるため，ヤコビアン行列を計算できません．

したがって，「回転量を表し」つつ「各パラメータが独立」な回転表現形式である対数クォータニオンを用いることにします．

3次元回転表現の相互変換

アルゴリズムの解説に必要な事前知識として，回転行列 R，クォータニオン $q = [q_x\ q_y\ q_z\ q_w]$，対数クォータニオン $u = [u_x\ u_y\ u_z]$，ヨー・ピッチ・ロール角(オイラー角) $[r_{yaw}\ r_{pitch}\ r_{roll}]$ との相互変換方法と，その実装コードをまとめます．なお，ここで示す方法は DirectX Graphics の左手系表現に準拠しています．OpenGL などの右手系では計算法が若干異なるので注意してください．

回転行列⇔クォータニオン

クォータニオン⇒トランスフォーム行列

結果だけ示します．

$$\begin{bmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y+q_wq_y) & 2(q_xq_z-q_wq_y) & 0 \\ 2(q_xq_y-q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z+q_wq_x) & 0 \\ 2(q_xq_z+q_wq_y) & 2(q_yq_z-q_wq_x) & 1-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
D3DXMATRIX quat2mat(const D3DXQUATERNION &q)
{
    D3DXMATRIX m(1, 0, 0, 0,  0, 1, 0, 0,  0, 0, 1, 0,  0, 0, 0, 1);
    m._11 = 1.0f - 2.0f * (q.y * q.y + q.z * q.z);
    m._12 = 2.0f * (q.x * q.y + q.w * q.z);
    m._13 = 2.0f * (q.x * q.z - q.w * q.y);
    m._21 = 2.0f * (q.x * q.y - q.w * q.z);
    m._22 = 1.0f - 2.0f * (q.x * q.x + q.z * q.z);
    m._23 = 2.0f * (q.y * q.z + q.w * q.x);
    m._31 = 2.0f * (q.x * q.z + q.w * q.y);
    m._32 = 2.0f * (q.y * q.z - q.w * q.x);
    m._33 = 1.0f - 2.0f * (q.x * q.x + q.y * q.y);
    return m;
}
```

回転行列⇒クォータニオン

定式化は煩雑になるので，実装コードだけ示します．

```
D3DXQUATERNION mat2quat(const D3DXMATRIX &m)
{
  D3DXQUATERNION q(0, 0, 0, 0);
  float s, trace = m._11 + m._22 + m._33 + 1.0;
  if (trace >= 1.0f) {
    s = 0.5f / sqrtf(trace);
    q.w = 0.25f / s;
    q.x = (m._23 - m._32) * s;
    q.y = (m._31 - m._13) * s;
    q.z = (m._12 - m._21) * s;
  }
  else {
    float max = m._22 > m._33 ? m._22 : m._33;
    if (max < m._11) {
      s = sqrtf(m._11 - (m._22 + m._33) + 1.0f);
      q.x = s * 0.5f;
      s = 0.5f / s;
      q.y = (m._12 + m._21) * s;
      q.z = (m._31 + m._13) * s;
      q.w = (m._23 - m._32) * s;
    }
    else if (max == m._22) {
      s = sqrtf(m._22 - (m._33 + m._11) + 1.0f);
      q.y = s * 0.5f;
      s = 0.5f / s;
      q.x = (m._12 + m._21) * s;
      q.z = (m._23 + m._32) * s;
      q.w = (m._31 - m._13) * s;
    }
    else {
      s = sqrtf(m._33 - (m._11 + m._22) + 1.0f);
      q.z = s * 0.5f;
      s = 0.5 / s;
      q.x = (m._31 + m._13) * s;
      q.y = (m._23 + m._32) * s;
      q.w = (m._12 - m._21) * s;
    }
  }
  return q;
}
```

クォータニオン⇔Axis-Angle

本記事には直接関係ありませんが，対数クォータニオンの説明に便利なので掲載します．なお，回転軸ベクトルを v'，その軸周りの回転量を θ' と表します．

クォータニオン⇒Axis-Angle

結果だけ示します．

$$\theta' = 2\cos^{-1}(q_w)$$

$$v' = [q_x/\sin(\theta'/2)\ q_y/\sin(\theta'/2)\ q_z/\sin(\theta'/2)]$$

```
void quat2axis(D3DXVECTOR3 &v, float &a, const D3DXQUATERNION &q)
{
  a = acosf(q.w) * 2.0f;
  v = D3DXVECTOR3(q.x, q.y, q.z) / sinf(a * 0.5f);
}
```

なお，DirectX Graphics の D3DXQuaternionToAxisAngle 関数は，クォータニオンの x-y-z 成分をそのまま 3 次元ベクトル化した $[q_x\ q_y\ q_z]$ を返すようです．つまり，回転軸ベクトルが単位ベクトルである保証はありません．ユーザ責任で正規化すればよいだけですが，この実装はちょっとどうかと思います．

Axis-Angle⇒クォータニオン

結果だけ示します．

$$q = [v'_x\sin(\theta'/2)\ v'_y\sin(\theta'/2)\ v'_z\sin(\theta'/2)\ \cos(\theta'/2)]$$

```
void axis2quat(D3DXQUATERNION &q, const D3DXVECTOR3 &v, const float &a)
{
  const float sina = sinf(a * 0.5f), cosa = cosf(a * 0.5f);
  q = D3DXQUATERNION(sina * v.x, sina * v.y, sina * v.z, cosa);
}
```

クォータニオン⇔対数クォータニオン

対数クォータニオンとは，一言で言うと「Axis-Angle 表現における 3 次元回転軸ベクトル v' の各成分に回転量 θ' を積算した 3 次元量」です．なぜそう定義されたのかについては，数学の偉い人に尋ねるしかありませんので，私達はあくまでも道具として扱います．

クォータニオン⇒対数クォータニオン

直前の定義に反しますが，DirectXでは，対数クォータニオンを次式のように定義しているようです(あくまでも処理結果からの予想です)．本記事ではこちらの定義に沿って実装しています．

$$u = \ln(q) = [\theta v_x\ \theta v_y\ \theta v_z\ 0]$$

この θ は Axis-Angle表現における軸周り回転量 θ' の半分の値 $\theta = \theta'/2$ です．

$$\theta = \cos^{-1}(q_w)$$

v はAxis-Angle 表現における回転軸 v' を回転量 θ でスケーリングしたベクトルです．

$$v = [q_x{}^*\theta/\sin(\theta)\ q_y{}^*\theta/\sin(\theta)\ q_z{}^*\theta/\sin(\theta)]$$

なお，参考文献 では Axis-Angle の回転軸 v' と回転量 θ' をそのまま用いて対数クォータニオンを定義しています．このことから察するに，回転軸ベクトルさえ同一であれば，対数クォータニオンの定義において軸周り回転量は大した問題ではないのかもしれません．

```
D3DXQUATERNION quat_ln(const D3DXQUATERNION &q)
{
  const float theta = acosf(q.w), isintheta = theta / sinf(theta);
  return D3DXQUATERNION(q.x * isintheta, q.y * isintheta, q.z * isintheta, 0);
}
```

対数クォータニオン⇒クォータニオン

結果だけ示します．

$$q = \exp(u) = [v_x \sin(\theta)/\theta \ v_y \sin(\theta)/\theta \ v_z \sin(\theta)/\theta \ \cos(\theta)]$$

ここで，$\theta = 0$ のときはゼロ除算のため計算できません．ただし，参考文献に示されている通り，実際には $\sin(\theta)$ のテイラー展開を利用して $\theta = 0$ のときの近似値を計算できます．

$$\frac{1}{\theta}\sin(\theta) = \frac{1}{\theta}\left(\theta + \frac{-1}{3!}\theta^3 + \frac{1}{5!}\theta^5 + \cdots\right) = 1 - \frac{\theta^3}{6} + \frac{\theta^5}{240} + \cdots$$

実装コードではテイラー展開を利用しない方法を示します．

```
D3DXQUATERNION quat_exp(const D3DXQUATERNION &u)
{
  const float theta = sqrtf(u.x * u.x + u.y * u.y + u.z * u.z), sintheta = sinf(theta) / theta;
  return D3DXQUATERNION(u.x * sintheta, u.y * sintheta, u.z * sintheta, cosf(theta));
}
```

回転行列⇔オイラー角

ヨー・ピッチ・ロール形式（$R_z R_x R_y$ オーダーのオイラー角）$[r_{yaw} \ r_{pitch} \ r_{roll}]$ について，回転行列との相互変換式を示します．

ヨー・ピッチ・ロール角⇒回転行列

順運動学の項を参照してください．

$$R = R_z(r_{roll}) R_x(r_{pitch}) R_y(r_{yaw})$$

```
D3DXMATRIX ypr2mat(float yaw, float pitch, float roll)
{
  D3DXMATRIX m, mt;
  D3DXMatrixRotationZ(&m, roll);
  m *= *D3DXMatrixRotationX(&mt, pitch);
  return m * *D3DXMatrixRotationY(&mt, yaw);
}
```

回転行列⇒ヨー・ピッチ・ロール角

こちらも結果だけ示します．Yaw-Pitch-Roll角から回転行列への変換式 $R = R_z(r_{roll}) R_x(r_{pitch}) R_y(r_{yaw})$ を展開してじっくり眺めれば，わりと簡単に導出できるはずです．

$$r_{pitch} = \sin^{-1}(-R_{32})$$

$$r_{roll} = \begin{cases} \tan^{-1}(R_{12}/R_{22}) + \pi & if \ |\cos(r_{pitch})| == 0 \& R_{12} \geq 0 \\ \tan^{-1}(R_{12}/R_{22}) - \pi & else \ if \ |\cos(r_{pitch})| == 0 \& R_{12} < 0 \\ \tan^{-1}(R_{12}/R_{22}) & otherwise \end{cases}$$

$$r_{yaw} = \begin{cases} \tan^{-1}(R_{31}/R_{33}) + \pi & if \ |\cos(r_{pitch})| == 0 \& R_{31} \geq 0 \\ \tan^{-1}(R_{31}/R_{33}) - \pi & else \ if \ |\cos(r_{pitch})| == 0 \& R_{31} < 0 \\ \tan^{-1}(R_{31}/R_{33}) & otherwise \end{cases}$$

```
void mat2ypr(float &yaw, float &pitch, float &roll, const D3DXMATRIX &m)
{
  roll = atan2f(m._12, m._22);
  pitch = asinf(-m._32);
  yaw = atan2f(m._31, m._33);

  if (fabsf(cosf(pitch)) < 1.0e-6f) {
    roll += m._12 > 0.0f ? D3DX_PI : -D3DX_PI;
    yaw += m._31 > 0.0f ? D3DX_PI : -D3DX_PI;
  }
}
```

対数クォータニオンからのヤコビアン行列導出

対数クォータニオンの各パラメータに関する回転行列の偏微分値を算出し，ヤコビアン行列を計算する手順を説明します．まず，上述の変換式を用いることで，対数クォータニオン→クォータニオン→回転行列の順で変換できることが示されました．すなわち，回転行列は対数クォータニオンの 3 つのパラメータについての関数として表されます．

$$R_i = R(u_i) = R([u_{i.x} \ u_{i.y} \ u_{i.z}])$$

また，ヤコビアン行列の $3i+j$ 行目は，順運動学計算式における関節 i の回転行列を，関節 i の回転量を表す対数クォータニオンの j 番目のパラメータに関する偏微分値で置き換えることで計算されます．例として，関節 i の対数クォータニオンパラメータ $u_{i.x}$ に関するヤコビアン行列の計算式を次に示します．

$$\frac{\partial p}{\partial \theta_{i.x}} = \vec{0} T_N R(\theta_{N-1}) T_{N-1} \cdots \left\{\frac{\partial R(\theta_i)}{\partial \theta_{i.x}}\right\} \cdots R(\theta_1) T_1$$

ただし，対数クォータニオンから回転行列への変換関数は，対数クォータニオン→クォータニオンと，クォータニオン→回転行列への変換という 2 つの関数で構成される合成関数です．したがって，対数クォータニオンの各パラメータに関する回転行列の偏微分は，合成関数の偏微分法に沿って次式のように書き換えられます．

$$\frac{\partial R}{\partial u_i} = \frac{\partial R}{\partial q_x}\frac{\partial q_x}{\partial u_i} + \frac{\partial R}{\partial q_y}\frac{\partial q_y}{\partial u_i} + \frac{\partial R}{\partial q_z}\frac{\partial q_z}{\partial u_i} + \frac{\partial R}{\partial q_w}\frac{\partial q_w}{\partial u_i}$$

さらに展開すると，次式が得られます．なお，ここでは $\partial q_j / \partial u_i$ を $\dot{q_j}$ と略記します．

$$\frac{\partial R}{\partial u_i} = \begin{bmatrix} -4(q_y\dot{q_y} + q_z\dot{q_z}) & 2(q_x\dot{q_y} + q_y\dot{q_x} + q_w\dot{q_z} + q_z\dot{q_w}) & 2(q_x\dot{q_z} + q_z\dot{q_x} - q_w\dot{q_y} - q_y\dot{q_w}) & 0 \\ 2(q_x\dot{q_y} + q_y\dot{q_x} - q_w\dot{q_z} - q_z\dot{q_w}) & -4(q_x\dot{q_x} + q_z\dot{q_z}) & 2(q_y\dot{q_z} + q_z\dot{q_y} + q_w\dot{q_x} + q_x\dot{q_w}) & 0 \\ 2(q_x\dot{q_z} + q_z\dot{q_x} + q_w\dot{q_y} + q_y\dot{q_w}) & 2(q_y\dot{q_z} + q_z\dot{q_y} - q_w\dot{q_x} - q_x\dot{q_w}) & -4(q_x\dot{q_x} + q_y\dot{q_y}) & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

この実装コードを示します．引数 q は対数クォータニオンから求められたクォータニオン，dq は対数クォータニオンパラメータ u_i に関するクォータニオンの偏微分値 $\partial q_j / \partial u_i$ です．

```
D3DXMATRIX dr_dq(const D3DXQUATERNION &q, const D3DXQUATERNION &dq)
{
  D3DXMATRIX m(0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0,  0, 0, 0, 0);
  m._11 = -4.0f * (q.y * dq.y + q.z * dq.z);
  m._12 =  2.0f * (q.x * dq.y + q.y * dq.x + q.w * dq.z + q.z * dq.w);
  m._13 =  2.0f * (q.x * dq.z + q.z * dq.x - q.w * dq.y - q.y * dq.w);
```

```
    m._21 =  2.0f * (q.x * dq.y + q.y * dq.x - q.w * dq.z - q.z * dq.w);
    m._22 = -4.0f * (q.x * dq.x + q.z * dq.z);
    m._23 =  2.0f * (q.y * dq.z + q.z * dq.y + q.w * dq.x + q.x * dq.w);
    m._31 =  2.0f * (q.x * dq.z + q.z * dq.x + q.w * dq.y + q.y * dq.w);
    m._32 =  2.0f * (q.y * dq.z + q.z * dq.y - q.w * dq.x - q.x * dq.w);
    m._33 = -4.0f * (q.x * dq.x + q.y * dq.y);
    return m;
}
```

次に，対数クォータニオン u_i に関するクォータニオンの偏微分値 $\partial q_j/\partial u_i$ の算出式を示します．導出の過程は無駄に長くなるので省略します(いずれ付録としてまとめるかもしれません)．

$$\frac{\partial q}{\partial u_x} = \left[-\frac{u_x}{\theta}\sin(\theta) \quad \frac{u_x^2}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right)+\frac{1}{\theta}\sin(\theta) \quad \frac{u_x u_y}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right) \quad \frac{u_x u_z}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right) \right]$$

$$\frac{\partial q}{\partial u_y} = \left[-\frac{u_y}{\theta}\sin(\theta) \quad \frac{u_x u_y}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right) \quad \frac{u_y^2}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right)+\frac{1}{\theta}\sin(\theta) \quad \frac{u_y u_z}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right) \right]$$

$$\frac{\partial q}{\partial u_z} = \left[-\frac{u_z}{\theta}\sin(\theta) \quad \frac{u_x u_z}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right) \quad \frac{u_y u_z}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right) \quad \frac{u_z^2}{\theta^2}\left(\cos(\theta)-\frac{1}{\theta}\sin(\theta)\right)+\frac{1}{\theta}\sin(\theta) \right]$$

以上の 3 式には共通項が多いので，これらは全てまとめて計算します．

```
void dq_ln_domain(D3DXQUATERNION &dqx, D3DXQUATERNION &dqy, D3DXQUATERNION &dqz, const D3DXQUATERNION &lnq)
{
    const double a = lnq.x, b = lnq.y, c = lnq.z;
    const double theta = sqrt(a * a + b * b + c * c);
    const double sint = sin(theta), cost = cos(theta);
    const double itheta = 1.0 / theta;
    const double it2c_it3s = cost / (theta * theta) - sint / (theta * theta * theta);

    dqx.w = -a * itheta * sint;
    dqx.x = a * a * it2c_it3s + itheta * sint;
    dqx.y = a * b * it2c_it3s;
    dqx.z = a * c * it2c_it3s;

    dqy.w = -b * itheta * sint;
    dqy.x = b * a * it2c_it3s;
    dqy.y = b * b * it2c_it3s + itheta * sint;
    dqy.z = b * c * it2c_it3s;

    dqz.w = -c * itheta * sint;
    dqz.x = c * a * it2c_it3s;
    dqz.y = c * b * it2c_it3s;
    dqz.z = c * c * it2c_it3s + itheta * sint;
}
```

次に，ヤコビアン行列 J の算出コードを示します．対数クォータニオンに関するクォータニオンの偏微分値を求め，クォータニオンに関する回転行列の偏微分値計算ルーチンに渡すという処理を，対数クォータニオンの 3 つのパラメータについて繰り返しています．

```
CMatrix CSceneView::ComputeQuatJacobian()
{
    CMatrix jm(3 * NUM_LINKS, 3);

    // 各関節についての繰り返し
    for (size_t jid = 0;jid < NUM_LINKS;++jid) {
      D3DXMATRIX dpose[NUM_LINKS];
      for (size_t i = 0;i < NUM_LINKS;++i)
        dpose[i] = m_pQuatPose[i];

      D3DXQUATERNION q, lnq, dq[3];
      D3DXQuaternionRotationMatrix(&q, &m_pQuatPose[jid]);
      D3DXQuaternionLn(&lnq, &q);
      dq_ln_domain(dq[0], dq[1], dq[2], lnq);

      for (size_t i = 0;i < 3;++i) {
        dpose[jid] = dr_dq(q, dq[i]);
        D3DXVECTOR3 dp = EffectorPos(dpose);
        for (size_t j = 0;j < 3;++j)
          jm(3 * jid + i, j) = dp[j];
      }
    }
    return jm;
}
```

最後に，ヤコビアン行列を用いたインバースキネマティクス関数を示します．①エフェクタ変位ベクトル $\triangle p$ の計算，②ヤコビアン行列の擬似逆行列 J^+ の計算と対数クォータニオン変位量 $\triangle\theta = J^+\triangle p$ の算出，③対数クォータニオンの修正 $\theta = \theta + \triangle\theta$ と関節回転行列への変換，という流れで処理を進めます．

```
bool CSceneView::UpdateQuatPose(const D3DXVECTOR3 &dest, double step)
{
    D3DXVECTOR3 pv = dest - EffectorPos(m_pQuatPose);
    float length = D3DXVec3Length(&pv);
    if (length < step)
      return false;
    pv *= step / length;

    CVector dv(pv);
    dv = dv * ComputeQuatJacobian().PseudoInverse();

    for (size_t i = 0;i < NUM_LINKS;++i) {
      D3DXQUATERNION q, eq;
      D3DXQuaternionRotationMatrix(&q, &m_pQuatPose[i]);
      D3DXQuaternionLn(&eq, &q);
      for (size_t j = 0;j < 3;++j)
        eq[j] += dv[i * 3 + j];
      D3DXQuaternionExp(&q, &eq);
      D3DXMatrixRotationQuaternion(&m_pQuatPose[i], &q);
    }
    return true;
}
```

オイラー角 IK との比較 [†]

クォータニオン IK との比較に作成した，オイラー角 IK の計算コードを示します．ヤコビアンを用いた逆運動学で解説した技術をもとに，クォータニオン IK のコーディングにそって作成しました．以前に比べ，ソースコードの見通しや簡潔さを改善できたと思います．特に説明すべき部分もないでしょうから，解説は省略します．

```
void dmat_euler_domain(D3DXMATRIX &dmr, D3DXMATRIX &dmp, D3DXMATRIX &dmy, float roll, float pitch, float yaw)
{
    D3DXMatrixIdentity(&dmr);  D3DXMatrixIdentity(&dmp);  D3DXMatrixIdentity(&dmy);
    dmr._11 = -sinf(roll);     dmp._11 = 0;               dmy._11 = -sinf(yaw);
    dmr._12 = cosf(roll);      dmp._22 = -sinf(pitch);    dmy._13 = -cosf(yaw);
    dmr._21 = -dmr._12;        dmp._23 = cosf(pitch);     dmy._22 = 0;
    dmr._22 = dmr._11;         dmp._32 = -dmp._23;        dmy._31 = -dmy._13;
    dmr._33 = 0;               dmp._33 = dmp._22;         dmy._33 = dmy._11;
    dmr._44 = 0;               dmp._44 = 0;               dmy._44 = 0;
}

CMatrix CSceneView::ComputeEulerJacobian()
{
    CMatrix jm(3 * NUM_LINKS, 3);
    for (size_t jid = 0;jid < NUM_LINKS;++jid) {
      D3DXMATRIX dpose[NUM_LINKS];
```

```
    for (size_t i = 0;i < NUM_LINKS;++i)
      dpose[i] = m_pEulerPose[i];

    D3DXMATRIX dm[3], m[3];
    float yaw, pitch, roll;
    mat2ypr(yaw, pitch, roll, m_pEulerPose[jid]);

    dmat_euler_domain(dm[0], dm[1], dm[2], roll, pitch, yaw);

    D3DXMatrixRotationZ(&m[0], roll);
    D3DXMatrixRotationX(&m[1], pitch);
    D3DXMatrixRotationY(&m[2], yaw);

    for (size_t i = 0;i < 3;++i) {
      switch (i) {
        case 0: dpose[jid] = dm[0] * m[1] * m[2]; break;
        case 1: dpose[jid] = m[0] * dm[1] * m[2]; break;
        case 2: dpose[jid] = m[0] * m[1] * dm[2]; break;
      }
      D3DXVECTOR3 dp = EffectorPos(dpose);
      for (size_t j = 0;j < 3;++j)
        jm(3 * jid + i, j) = dp[j];
    }
  }
  return jm;
}

bool CSceneView::UpdateEulerPose(const D3DXVECTOR3 &dest, double step)
{
  D3DXVECTOR3 pv = dest - EffectorPos(m_pEulerPose);
  float length = D3DXVec3Length(&pv);
  if (length < step)
    return false;
  pv *= step / length;

  CVector dv(pv);
  dv = dv * ComputeEulerJacobian().PseudoInverse();

  for (size_t i = 0;i < NUM_LINKS;++i) {
    float ra[3];
    mat2ypr(ra[0], ra[1], ra[2], m_pEulerPose[i]);
    for (size_t j = 0;j < 3;++j)
      ra[2 - j] += dv[i * 3 + j];
    m_pEulerPose[i] = ypr2mat(ra[0], ra[1], ra[2]);
  }
  return true;
}
```

結果

アニメーションの評価についてはデモソフトをご覧いただくとして，ここでは IK 計算の精度を評価します．すなわち，指定したエフェクタ変位量と，算出された関節回転量によって発生するエフェクタ変位量の差を評価します．ただしここでは多少手抜きをし，理論的な計算繰り返し回数と，エフェクタが目標位置に到達するまでの実際の計算繰り返し回数の誤差を算出しました．つまり，各ステップのエフェクタ変位量の誤差は考えず，全体としてのエフェクタ変位量の精度を評価します．

評価テストの実装コードを示します．計算繰り返し回数の理論値は，(|目標位置 - 初期エフェクタ位置|/エフェクタ位置ステップ幅)で算出できます．そして，オイラー角 IK，クォータニオン IK のそれぞれの計算繰り返し回数について，理論値からの平方根平均自乗誤差(Root Mean Squared Error:RMS誤差)を 100 回の試行によって求めます．

```
void CSceneView::OnTestView()
{
  CWaitCursor cur;
  float sum0 = 0, sum1 = 0;
  for (size_t trial = 0;trial < 100;++trial) {
    for (size_t i = 0;i < NUM_LINKS;++i) {
      D3DXMatrixRotationYawPitchRoll(&m_pEulerPose[i], D3DX_PI * rand() / RAND_MAX, D3DX_PI * rand() / RAND_MAX, D3DX_PI * rand() / RAND_MAX);
      m_pQuatPose[i] = m_pEulerPose[i];
    }

    D3DXVECTOR3 dest = -EffectorPos(m_pEulerPose);

    int loop0, loop1;
    for (loop0 = 0;loop0 < 10000 && UpdateQuatPose(dest, 0.001f);++loop0);
    for (loop1 = 1;loop1 < 10000 && UpdateEulerPose(dest, 0.001f);++loop1);

    int ideal = static_cast<int>(D3DXVec3Length(&dest) * 2.0f / 0.001f) + 1;
    sum0 += (loop0 - ideal) * (loop0 - ideal);
    sum1 += (loop1 - ideal) * (loop1 - ideal);
  }

  CString str;
  str.Format("%f %f", sqrtf(sum0 / 100.0f), sqrtf(sum1 / 100.0f));
  MessageBox(str);
}
```

適当に3回ほどテストした結果を下表にまとめます．

試行	クォータニオンIK	オイラー角IK
A	0.678	52.341
B	0.854	99.646
C	0.656	98.148

クォータニオン IK の誤差は無視できる程度という，非常に優良な結果を出しています．一方，オイラー角 IK は非常に大きな誤差を生じています．誤差の内容を詳しく見ると，オイラー角 IK は指定した回数以上の計算繰り返しを必要とすることがわかりました．これは，指定した距離以下しかエフェクタが移動しなかったり，指定した方向以外へ移動していることを示唆しています．この誤差は，回転行列に対するオイラー角の非線形性によって生じていると考えられます．一方，対数クォータニオンも回転行列とは非線形な関係にあるのですが，これはほとんど無視できるレベルであるため，上記のような結果になったのだと思われます．

次に計算速度を評価します．姿勢更新処理1000回分の計算時間をミリ秒単位で計測し，1回あたりの計算時間を算出しました(Athlon64X2 5000+, Vista 64bit, 2GB-RAM, 32bitアプリ)．テストコードを以下に示します．

```
void CSceneView::OnTestView()
{
  for (size_t i = 0;i < NUM_LINKS;++i) {
    D3DXMatrixRotationYawPitchRoll(&m_pEulerPose[i], D3DX_PI * rand() / RAND_MAX, D3DX_PI * rand() / RAND_MAX, D3DX_PI * rand() / RAND_MAX);
    m_pQuatPose[i] = m_pEulerPose[i];
  }

  D3DXVECTOR3 dest = -EffectorPos(m_pEulerPose);

  timeBeginPeriod(1);
  DWORD t0, t1;

  t0 = timeGetTime();
  for (int i = 0;i < 1000;++i)
    UpdateQuatPose(dest, 0.001f);
  t0 = timeGetTime() - t0;

  t1 = timeGetTime();
  for (int i = 0;i < 1000;++i)
    UpdateEulerPose(dest, 0.001f);
  t1 = timeGetTime() - t1;
  timeEndPeriod(1);

  CString str;
  str.Format("%f %f", t0 / 1000.0f, t1 / 1000.0f);
  MessageBox(str);
}
```

結果を下表に示します．

試行	クォータニオンIK [msec]	オイラー角IK [msec]

A	0.022	0.025
B	0.020	0.022
C	0.019	0.024

ほぼ同等の結果ですが，クォータニオンIKのほうが若干高速でした．コード的にもアルゴリズム的にもオイラー角IKのほうがシンプルなので，少し意外な結果です．考えられる原因としては，行列⇒オイラー角への分解のために，atan2，asin関数等を使っているためでしょうか．もちろん，最適化や実装の方法によっては結果が逆転する可能性もありますので，この速度比較の結果はあくまでも参考程度としてください．本質的な計算量の差とは言い切れないと思います．

まとめ

クォータニオンを用いたインバースキネマティクス法を解説しました．対数クォータニオンを導入することで，エフェクタ位置追従性についてはオイラー角 IK よりも優れた結果が得られました．したがって，指定した速度でエフェクタを正確に動かしたい，という用途には最適だと思います．また，計算量もオイラー角IKに遜色なく，実装方法によってはより効率的に計算できそうです．

ただし，特異姿勢や「自然さ」の問題は解決できないままです．さらに，オイラー角表現と比べると対数クォータニオンは直観性に欠ける（各パラメータが何を意味するのかわかりずらい）ので，関節可動域の設定はさらに難しくなります．ヤコビアン行列の加重擬似逆行列や，冗長変数の計算パラメータ設定についても同様ですね．このように様々な問題もありますが，オイラー角IKの代替として十分使えそうです．

今回紹介した技術は，クォータニオンを使ったIKの一つのアルゴリズムであり，もっとベストな解決法がありそうです．どなたかアイデアがあれば，ぜひ挑戦してください．
