# MOCAPデータのフィルタリングと補間

http://www.tmps.org/index.php?MOCAP%A5%C7%A1%BC%A5%BF%A4%CE%A5%D5%A5%A3%A5%EB%A5%BF%A5%EA%A5%F3%A5%B0%A4%C8%CA%E4%B4%D6

デジタル信号処理の分野では，サウンドデータや画像データといったデジタルデータの加工，編集処理方法が広く研究されています．その研究成果の一部は，多次元デジタル時系列データであるモーションデータにも応用することができ，例えばデジタルフィルタを利用した動作の変形法や，複数のモーションデータの補間による動作制作方法が提案されています．本稿では，畳み込み（convolution）と補間（interpolation）を用いたモーションデータの加工方法について説明します．この2つの処理は，いずれもカーネル関数を用いた加重和計算によって実現されますが，3次元回転データに応用する場合には少し工夫が必要です．本稿ではそれらの注意点と実装方法を中心に説明します．なお，本稿は3D空間における回転の表現形式の記事を前提として作成していますので，そちらも合わせて参照ください．

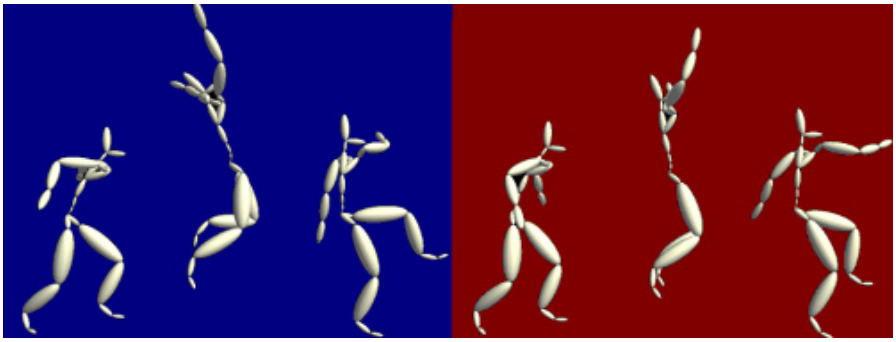

Fig.1 平滑化フィルタの適用例　左：オリジナル　右：平滑化結果



## サンプルプログラム

MOCAPデータファイルで作成したプログラムを拡張し，モーションデータに対するフィルタリング，補間処理を行うプログラムを作成します．サンプルプログラムは，VisualC++ 2005 + DirectX SDK Update February 2006 の環境下で，それぞれ MFC と DXUT を用いて作成しています．

DXUT 版(VC++2005, 283kb)
MFC 版(VC++2005, 66kb)

## 参考文献


1. Armin Bruderlin and Lance Williams, "Motion Signal Processing", SIGGRAPH 1995, pp.97-104, 1995.
2. Andrew Witkin and Zoran Popovic, "Motion Warping", SIGGRAPH 1995, pp.105-108, 1995.
3. Jehee Lee and Sung Yong Shin, "A Coordinate-Invariant Approach to Multiresolution Motion Analysis", Graphical Models, vol.63, no.2, pp.87-105, 2001.


デジタル信号処理を応用したモーションデータの加工方法は，まず 1.と 2. の論文で議論されました．ただし，これらの論文では，オイラー角表現された関節回転角度に対する処理方法を提案していますが，そのままでは3D空間における回転の表現形式で説明しているような不具合が生じます．この点については 3.の論文で詳しく議論されており，数学的に正しいモーションデータの処理方法も提案されています．

これらの研究成果を踏まえ，本稿ではクォータニオンと Exponential map を用いた，3次元回転データに対するデジタルフィルタ処理と補間方法を説明します．

# 畳み込みフィルタ処理 †

## カーネルを用いた畳み込みフィルタ †

デジタルフィルタの一種である畳み込みフィルタ(convolution filter)は，ある時系列データ $f(t), t \in \{1 \cdots N\}$ と畳み込みカーネル $h_k, k \in \{-K \cdots 0 \cdots K\}$ が与えられたとき，次式によって新しい時系列 $g(t), i \in \{1 \cdots N\}$ を計算するモデルです．

$$g(\tau) = \sum_{k=-K}^{K} h_k f(\tau + k)$$

$K = 2$ のカーネルを用いた畳み込みの処理手順は，下図のように表されます．

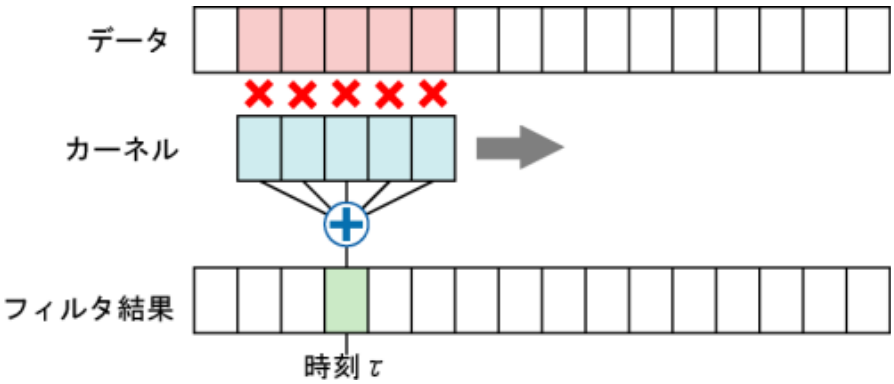


Fig.2 カーネルを用いた畳み込み計算

まず，時刻 $\tau$ にカーネルの中心を配置します．次に，カーネルの各成分と対応するデータとの積をそれぞれ計算し，その和によって $g(\tau)$ を計算します．この処理を全ての $\tau$ について行うことで，フィルタリングされた時系列 $g(t)$ が得られます．これら全体の処理は次のようなプログラムコードで表されます．

```
1     const int K = 2;
2     float f[N];
3     float g[N];
4     float h[2 * K + 1];
5
6     for (int i = K;i < N - K;++i)
7     {
8       float sum = 0;
9       for (int k = -K;k <= K;++k)
10        sum += kernel[k + K] * data[i + k];
11      result[i] = sum;
12    }
```

このコードでは，$t \leq K$ と $t > N - K$ なるデータが未計算のままです．これは，それらの時刻のデータを計算しようとすると，カーネルが元データの範囲を超えてしまう(つまり上記コード10行目の「data[i + k]」のインデクスが有効範囲を超えてしまう)ためです．対処方法はいくつかありますが，ここでは元データをそのままコピーするという最も単純な方法をとります．

カーネルの係数や幅を変更することで，低域通過フィルタや高域通過フィルタ，微分フィルタなどの様々なフィルタを設計できます．ただし，本稿はカーネル係数の総和が 1.0 であるようなフィルタのみを扱います(理由は後述します)．

## 3次元回転量の畳み込み †

モーションデータにカーネルフィルタを適用する場合は，各関節ごとに畳み込み計算を行うことになります．つまり，関節 $i$ の回転量(クォータニオンや回転行列)の時系列を $q_i(t)$ と表すとき，畳み込み計算は次の式で表されます．

$$q_i'(\tau) = \sum_{k=-K}^{K} h_k q_i(\tau + k)$$

しかし，クォータニオンや回転行列で表現された回転量は，本来は乗算計算によって合成するものです．そのため，上式に示す加重和計算では速度非一定性の問題から，畳み込み結果に大きな誤差を生じる可能性があります．本稿では文献 3.を参考に，Exponential map(以後，exp マップと略記) を用いた座標系非依存な演算によって，できる限り正確な値を計算する方法を説明します．もちろん，expマップも万能

ではないのでいくつかの不具合を生じますが，比較的最良の方法だと思われます．

計算方法を説明するために，$K=1$ のシンプルな畳み込みを考えます．なお，ここでは $q_i(\tau+k)$ を $q_k$，$p_i(\tau+k)$ を $p_k$ のように略記します．まず，exp マップ $p_i=\ln(q_i)$ を用いて上式を書き換えます．

$$p'_0=h_{-1}p_{-1}+h_0p_0+h_1p_1$$

exp マップは特異状態を持つため，この式でも正確な値は計算できません．そこで，$\sum_{K=-1}^{K} h_k=1$ の条件を利用し，次のように式を変形します．

$$\begin{aligned} p'_0 &= p_0-(h_{-1}+h_1)p_0+h_{-1}p_{-1}+h_1p_1 \\ &= p_0+h_{-1}(p_{-1}-p_0)+h_1(p_1-p_0) \end{aligned}$$

最後に，両辺の指数を計算することで次式を得ます．

$$\begin{aligned} \exp\{p'_0\} &= \exp\{p_0+h_{-1}(p_{-1}-p_0)+h_1(p_1-p_0)\} \\ &= \exp(p_0)\exp\{h_{-1}(p_{-1}-p_0)+h_1(p_1-p_0)\} \\ q'_0 &= q_0\exp\{h_{-1}\ln(q_{-1}^{-1}q_0)+h_1\ln(q_1^{-1}q_0)\} \end{aligned}$$

こうして得られた計算式は，座標系非依存な演算であることに注意してください．つまり，最後の式中の $\exp$ の中括弧内はすべて 2 つのクォータニオン間の回転量を表す exp マップなので，基準座標系のとり方に関係なく，スケーリングや加算ができます．そのため，特異状態の発生しない座標系非依存な畳み込み計算が行えます．

ただし，exp マップは距離非最小性による誤差を含むので，必ずしも理論値と一致しません．文献 3.ではその誤差量を軽減するためのフィルタリング法を提案されていますので，興味のある方は参照してみてください．

## 畳み込み計算の実装

3次元回転量の畳み込み計算の実装方法を示します．まず，2つのクォータニオンの差分の exp マップ $\ln(q_k^{-1}q_0)$ を計算する関数を作成します．

```
 1 //2つのクォータニオンの差分の exp マップの計算
 2 D3DXQUATERNION calc_differential_expmap(const D3DXQUATERNION &q0, const D3DXQUATERNION &q1)
 3 {
 4   D3DXQUATERNION q0i;
 5   D3DXQuaternionInverse(&q0i, &q0);
 6   D3DXQUATERNION qd = q0i * q1;
 7
 8   if (qd.w < 0)
 9     qd *= -1.0f;
10   D3DXQUATERNION expm;
11   D3DXQuaternionLn(&expm, &qd);
12   return expm;
13 }
```

6, 7 行目で，クォータニオンの w 成分を正値にするような正規化処理があります．理論的な理由はよくわかっていませんが，この正規化処理がないと正しい畳み込み計算が行われませんでした．クォータニオンは正負を反転しても回転行列に変換すると同じ値を示すようなので，数値的には問題ないとは思いますが...(調査中)

次に，カーネルを用いた畳み込み計算の関数を作成します．なお，カーネル係数の和は必ず 1.0 とすることから，関数名は"interpolate: 補間"としました．

```
 1 // カーネルを用いた畳み込み計算
 2 D3DXQUATERNION interpolate_rotation(const std::vector &rots, const std::vector<float> &kernel)
 3 {
 4   float ksum = 0;
 5   for (size_t k = 0;k < kernel.size();++k)
 6     ksum += kernel[k];
 7
 8   D3DXQUATERNION qd, exd(0, 0, 0, 0);
 9   for (size_t k = 1;k < kernel.size();++k)
```

```
10 |      exd += kernel[k] * calc_differential_expmap(rots[0], rots[k]);
11 |    exd /= ksum;
12 |    D3DXQuaternionExp(&qd, &exd);
13 |    return rots[0] * qd;
14 ! }
```

一応，カーネルの総和を 1.0 に正規化する処理も含めています．

さらに，比較のために実装した QLERP(Quaternion Linear intERPolation)と，exp マップの加重和による畳み込み計算のコードも示します．

```
 1 // QLERP による畳み込み
 2 D3DXQUATERNION interpolate_rotation_qlerp(const std::vector &rots, const std::vector<float> &kernel)
 3 {
 4   float ksum = 0;
 5   for (size_t k = 0;k < kernel.size();++k)
 6     ksum += kernel[k];
 7
 8   D3DXQUATERNION res, qi(0, 0, 0, 0);
 9   for (size_t k = 0;k < kernel.size();++k)
10     qi += (kernel[k] / ksum) * rots[k];
11   D3DXQuaternionNormalize(&res, &qi);
12   return res;
13 }
```

```
 1 // exp マップの加重和による畳み込み
 2 D3DXQUATERNION interpolate_rotation_exp(const std::vector &rots, const std::vector<float> &kernel)
 3 {
 4   float ksum = 0;
 5   for (size_t k = 0;k < kernel.size();++k)
 6     ksum += kernel[k];
 7
 8   D3DXQUATERNION qd, exd(0, 0, 0, 0);
 9   for (size_t k = 0;k < kernel.size();++k)
10   {
11     D3DXQUATERNION ext;
12     qd = rots[k]; //qd = rots[k].w < 0 ? rots[k] * -1.0f : rots[k];
13     D3DXQuaternionLn(&ext, &qd);
14     exd += kernel[k] * ext;
15   }
16   exd /= ksum;
17   D3DXQuaternionExp(&qd, &exd);
18   return qd;
```

## 移動平均によるモーションデータの平滑化

カーネルを用いた畳み込みフィルタの応用例として，移動平均法によるモーションデータの平滑化処理の実装例を示します．なお移動平均法とは，畳み込み計算における全てのカーネル係数を 1.0 として，ある時間区間内の平均値を計算する方法です．

まず，MOCAPデータファイルで作成した CMotionData クラスを利用し，モーションデータ全体をフィルタリングする Filter 関数を作成します．

```
 1 bool NMotionSignal::Filter(CMotionData &dest, const std::vector<float> &kernel)
 2 {
 3   if (!dest.IsActive() || kernel.empty())
 4     return false;
 5   if (kernel.size() % 2 == 0 || dest.NumFrames() < kernel.size())
 6     return false;
 7
 8   float ksum = 0;
 9   for (size_t k = 0;k < kernel.size();++k)
10     ksum += kernel[k];
11   size_t kernel_width = kernel.size() / 2;
12
13   std::vector rots(kernel.size());
14   CMotionData tmp = dest;
15   for (size_t f = kernel_width;f < dest.NumFrames() - kernel_width;++f)
16   {
17     D3DXVECTOR3 pos(0, 0, 0);
18     for (size_t k = 0;k < kernel.size();++k)
19       pos += tmp.GetPosition(f + k - kernel_width) * kernel[k];
20     dest.SetPosition(f, pos / ksum);
21
22     for (size_t j = 0;j < dest.NumJoints();++j)
```

```
23      {
24        for (size_t k = 0;k < kernel.size();++k)
25          rots[k] = tmp.GetRotation(f + k - kernel_width, j);
26        dest.SetRotation(f, j, interpolate_rotation(rots, kernel));
27      }
28    }
29    return true;
30  }
```

17～20 行目ではルート位置の時系列に対するフィルタリングを行いますが，単純なカーネル和によって新しい位置を算出しています．一方，22～27 行目では各関節回転量の時系列に対し，interpolation_rotation 関数を利用したフィルタリングを行っています．

次に，$K = 25$ の移動平均フィルタを利用するための，Filter 関数の呼び出し側のコードを示します．

```
1     std::vector<float> kernel(51, 1.0f);
2     NMotionSignal::Filter(*m_pMotion, kernel);
```

1 行目で全ての要素が 1.0 に初期化された要素数 51 のカーネルを作成し，2 行目でモーションデータとの畳み込み計算を行っています．

### 平滑化処理結果例

ダイナミックな動作に対し，移動平均による平滑化を施した処理結果のムービーを示します．QLERP や exp マップの加重和では，予期しない不具合が生じていますが，本稿で説明した方法では（少なくとも見た目には）正しい処理が行われていることが確認できると思います．

- オリジナル動作（151kb）
- 平滑化処理結果（127kb）
- QLERPによる平滑化処理結果（132kb）
- exp マップの加重和による処理結果（133kb）

## モーションデータの補間

次に，モーションブレンディング法（Motion Blending）や動作補間法（Motion Interpolation）と呼ばれる手法の基礎について説明します．モーションブレンディング法は，複数のモーションの足し合わせによって，それらの中間的な動作を計算する方法です．モーションブレンディング法は，2つのキャラクタアニメーションシーケンスを接続する際に，つなぎ目にあたる動作を滑らかに補間することで切り替わりの不連続さを目立たなくしたり，多数のサンプル動作から新しい動作を合成するための基礎技法として利用されています．本稿では，その基本となるモーションデータのカーネル補間について説明します．

### 2つのモーションデータの線形補間

最も単純な例として，2つのモーションデータを線形補間する場合について考えます．これは，2つのルート位置の線形補間と，対応する関節で回転量の線形補間つまりクォータニオンの球面線形補間：SLERP を，各時刻で独立に計算することで求められます．この実装コードは次のようになります．

```
1   bool NMotionSignal::Interpolate(CMotionData &dest, const CMotionData &src1, const CMotionData &src2, float rate)
2   {
3     if (!src1.IsActive() || !src2.IsActive())
4       return false;
5     if (src1.NumJoints() != src2.NumJoints())
6       return false;
7
8     dest.Initialize(src1.NumFrames() < src2.NumFrames() ? src1.NumFrames() : src2.NumFrames(), src1.NumJoints());
9     for (size_t f = 0;f < dest.NumFrames();++f)
10    {
11      dest.SetPosition(f, (rate - 1.0f) * src1.GetPosition(f) + rate * src2.GetPosition(f));
12      for (size_t j = 0;j < dest.NumJoints();++j)
13      {
14        D3DXQUATERNION q1 = src1.GetRotation(f, j);
15        D3DXQUATERNION q2 = src2.GetRotation(f, j);
16        D3DXQUATERNION qc;
17        D3DXQuaternionSlerp(&qc, &q1, &q2, rate);
18        dest.SetRotation(f, j, qc);
19      }
20    }
21    return true;
22  }
```

補間対象の動作の時間長が異なる場合，合成動作は時間長が短いほうに合わせられます．

## 任意数のモーションデータのカーネル補間

2つ以上の任意数のモーションデータを補間する場合，時系列に対する畳み込みフィルタと同様に，カーネルを用いた加重和を計算することになります．ここで，畳み込みフィルタでは，ある時系列データにおいて，複数の時刻からのサンプリングデータに対するカーネル補間を計算していました．一方，複数の時系列データを補間する場合には，ある時刻において，複数の時系列データからのサンプリングデータに対するカーネル補間を計算します．すなわち，畳み込みフィルタが時間領域に対してカーネルフィルタを適用するのに対し，ここでは空間領域に対してカーネルフィルタを適用することになります．

少々回りくどい説明になりましたが，モーションブレンディングでも結局，interpolation_rotation 関数を利用することになります．ただし，カーネルは時間サンプルではなく，動作サンプルに対して適用されます．実装コード例は次の通りです．

```
 1  bool NMotionSignal::Interpolate(CMotionData &dest, std::vector &srcs, std::vector<float> &kernel)
 2  {
 3    if (kernel.empty() || kernel.size() != srcs.size())
 4      return false;
 5
 6    float ksum = 0;
 7    for (size_t k = 0;k < kernel.size();++k)
 8      ksum += kernel[k];
 9
10    size_t frames = srcs[0]->NumFrames(), joints = srcs[0]->NumJoints();
11    for (size_t m = 1;m < srcs.size();++m)
12    {
13      if (joints != srcs[m]->NumJoints())
14        return false;
15      if (frames > srcs[m]->NumFrames())
16        frames = srcs[m]->NumFrames();
17    }
18    dest.Initialize(frames, joints);
19
20    std::vector rots(kernel.size());
21    for (size_t f = 0;f < frames;++f)
22    {
23      D3DXVECTOR3 pos(0, 0, 0);
24      for (size_t k = 0;k < kernel.size();++k)
25        pos += kernel[k] * srcs[k]->GetPosition(f);
26      dest.SetPosition(f, pos / ksum);
27
28      for (size_t j = 0;j < joints;++j)
29      {
30        for (size_t k = 0;k < kernel.size();++k)
31          rots[k] = srcs[k]->GetRotation(f, j);
32        dest.SetRotation(f, j, interpolate_rotation(rots, kernel));
33      }
34    }
35    return true;
36  }
```

NMotionSignal::Filter 関数と多数の類似点があることがわかると思います．また，ここでも合成動作の時間長は，最も短いサンプル動作に合わせられます．

## 補間結果例

単純な 2 つの手伸ばし動作を補間する例を示します．上方へ手を伸ばす動作と，下後方へ手を伸ばす動作をブレンディングした結果，中段側方へ手を伸ばす動作が合成されることが確認できると思います．

- 上方への手伸ばし動作(24kb)
- 下後方への手伸ばし動作(45kb)
- 補間動作(31kb)

# まとめ

モーションデータに対するカーネルフィルタの適用方法と，複数のモーションデータの補間法について説明しました．理論的にはやや難解かもしれませんが，実装コードは比較的シンプルなので，合わせて理解していただければと思います．また，今回説明した方法は，現時点では最良の方法だと思われますが，まだまだ多くの課題が残されています(特に計算量の問題)．今後の研究成果に期待したいところです．

---

Last-modified: 2006-04-30 (日) 10:17:53 (2920d)

Site admin: cherub