モーションのパラメータ化

http://www.tmps.org/index.php?%A5%E2%A1%BC%A5%B7%A5%E7%A5%F3%A4%CE%A5%D1%A5%E9%A5%E1%A1%BC%A5%BF%B2%BD

今回はモーションデータのパラメータ化技術について解説します．これはモーションブレンディングを拡張することで開発された技術で，複数の類似動作の加算合成によって新しい動作を合成する技法です．ただし，モーションブレンディングのように補間の重み付けを直接操作するのではなく，動作パラメータと呼ばれる少数のパラメータによって合成計算を制御します．例えば，Fig.1 ではパンチターゲット位置によってパンチ動作データをパラメータ化しています．その結果，黄色の球をパンチするサンプル動作から，任意のターゲット（赤色の球）をパンチする動作を合成できます．この技術により，他にも歩行動作における足跡や歩行到達位置，ジャンプ動作のジャンプ高さや落下地点を操作することで，モーションデータを直感的に編集するようなインターフェースを作成できます．

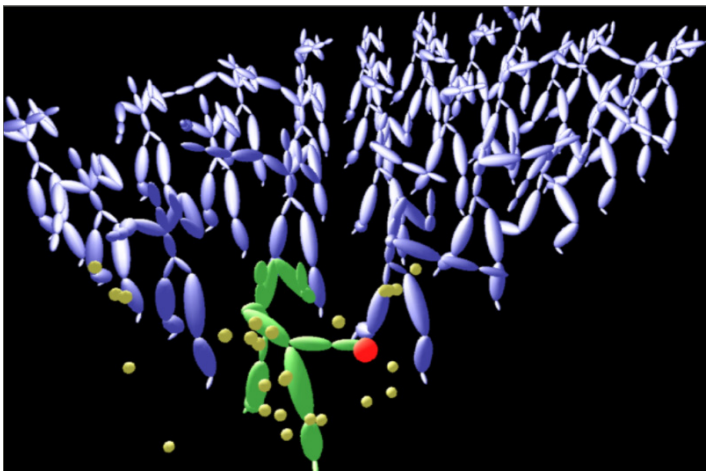

Fig.1 実行スナップショット

サンプルプログラム

サンプルプログラムは，VisualC++ 2005 + DirectX SDK Update August 2006 の環境下で，MFC を用いて作成しており，コンパイルには GNU Scientific Library が必要です．アーカイブにはパンチ動作の BVH モーションキャプチャデータも含めてあります．今回のバイナリデータの実行には，DirectX SDK (August 2006) もしくは DirectX End User Runtime (August 2006) のインストールが必要です．

サンプルソースコード(VC++2005, MFC, 444kb)
サンプルバイナリ(694kb)

遊び方

サンプルプログラムはパンチ動作をパラメータ化します．具体的には，パンチのターゲット位置（動作の真ん中の時刻における右手先位置）を操作することで新しいパンチ動作を合成できます．アプリケーションの操作方法は次の通りです．

1. 実行形式ファイル（Parametrization.exe）とエフェクトファイル（Viewer.fx）を同じディレクトリに置きます
2. プログラムを起動して [ファイル]→[開く]から，サンプル BVH モーションデータファイルを一括ロードします．
3. [Motion]→[Spatial align]で，全てのサンプル動作の向き・位置を正規化します．
4. [Motion]→[Temporal align]で，全てのサンプル動作を最初のサンプル動作に同期させます
5. [Motion]→[Calc parameters]で，サンプル動作のパンチターゲット位置を計算，可視化します．

6. [Motion]→[Analyze]で，パラメータ化の前処理を行います．
7. 方向キーの↑↓でパンチ位置の高さを，←→で正面からの回転角度を操作します．そして[Motion]→[Synthesis]で，指定したターゲットをパンチする動作を生成します．

- 合成結果は [ファイル]→[名前を付けて保存]で BVH 形式で保存できます

参考文献

1. Douglas J. Wiley and James K. Hahn, Interpolation Synthesis of Articulated Figure Motion, IEEE Computer Graphics and Applications, vol.17, no.6, pp.39-22745, 1997.
2. Charles F. Rose and Peter-Pike J. Sloan and Michael F. Cohen, Artist Directed Inverse-Kinematics Using Radial Basis Function Interpolation", Computer Graphics Forum, vol.20, no.3, pp.239-250, 2001.
3. Lucas Kovar and Michael Gleicher, "Automated Extraction and Parameterization of Motions in Large Data Sets", ACM Transactions on Graphics, vol.23, no.3, pp.559-568, 2004.
4. Tomohiko Mukai and Shigeru Kuriyama, "Geostatistical Motion Interpolation", ACM Transactions on Graphics, vol.24, no.3, pp.1062-1070, 2005.

モーションブレンディングを応用したモーションパラメータ化法は，まず論文[1]で提案されました．これは最近傍法＋線形補間＋ランダムサンプリング法を応用したものです．次に，動径基底関数補間法（Radial Basis Function Interpolation，RBF補間法）を応用した技法である論文[2]が発表されました（同一のアルゴリズムはShape by Exampleにも利用されています）．さらに，パラメータ化処理を完全に自動化するシステムが論文[3]で発表されました．このシステムをベースとし，空間統計学を導入することで補間精度の向上を図ったのが論文[4]です．今回の記事では，論文[4]で導入された空間統計学の計算を簡略化したアルゴリズムを紹介します．

これらの技法は「Motion Parameterization（動作のパラメータ化）」や「Motion Interpolation（動作補間法）」と呼ばれ，ブレンディングとは区別して扱われます．

アルゴリズムと実装

モーションブレンディングのアルゴリズムとの主な差異は，動作パラメータをもとに各サンプル動作への補間カーネルを計算する処理が追加されている点です．本サンプルプログラムでは単純化したクリギングによって補間カーネルを算出します（OnMotionAnalyze関数とOnMotionSynthesis関数）．また，サンプル動作を同期させるための時間的正規化処理（OnMotionTemporalAlign関数）に加え，全てのサンプル動作の初期姿勢を揃えるための空間的正規化（OnMotionSpatialAlign関数）を前処理に追加しています．

なお，動作データの空間的，時間的正規化処理では，最初にロードされたサンプル動作データを基準サンプルとして扱います．そして，2 番目以降にロードされた動作データを基準サンプルに合わせて変換します．

ファイル読込み

個人的な備忘録のため，MFCのファイルダイアログを用いた複数 BVH ファイルの一括読込み処理を示します．まず，CFileDialog のコンストラクタに "OFN_ALLOWMULTISELECT" フラグを指定します．次に，ファイルパスを格納するためのメモリ領域（dlg.m_ofn.lpstrFile）を確保し，そのサイズを dlg.m_ofn.nMaxFile に渡します．そして，ダイアログ呼び出しが成功した後，ファイルパスを取得したリストの先頭を dlg.GetStartPosition() で取得し，順次 dlg.GetNextPathName(pos) によってファイルパスを 1 つずつ取得します．もちろん，OnFileOpen 関数のスコープを外れる前にファイルパス格納のためのメモリ領域は解放しなければなりません．

なお"OFN_NOCHANGEDIR" フラグは，読込みファイルの格納先にカレントディレクトリを移動させないためのオプションです．

```
1  void CSceneView::OnFileOpen()
2  {
3    CFileDialog dlg(TRUE, "bvh", NULL, OFN_ALLOWMULTISELECT | OFN_NOCHANGEDIR,
4                    "BVH Motion File (*.bvh)|*.bvh|||", this);
5    dlg.m_ofn.lpstrFile = new char[512*1024-1];
6    memset(dlg.m_ofn.lpstrFile, 0, 512*1024-1);
7    dlg.m_ofn.nMaxFile = 512 * 1024 - 1;
8    if (dlg.DoModal() != IDOK) {
9      delete[] dlg.m_ofn.lpstrFile;
10   return;
11   }
12   ReleaseMotions();
13
14   POSITION pos = dlg.GetStartPosition();
15   while (pos != NULL) {
16     m_vecSampleFigures.push_back(new CFigure);
17     m_vecSampleMotions.push_back(new CMotionData);
18     NMotionFile::LoadBvhFile(dlg.GetNextPathName(pos),
19                              m_vecSampleFigures.back(), m_vecSampleMotions.back());
20     m_vecSampleFigures.back()->SetPose(m_vecSampleMotions.back(), 0);
21   }
22   delete[] dlg.m_ofn.lpstrFile;
23   SendMessage(WM_USER_RENDER_SCENE);
```

```
24 ! }
```

空間的正規化 [†]

パラメータ化の前処理として，全てのサンプルの初期姿勢がほぼ同じ座標に位置し，ほぼ同じ方向を向くように正規化します．正規化の処理は時刻 0 における初期姿勢について，①全サンプルのルート位置を原点上に移動（高さはそのまま保持），②基準サンプルの腰横方向（+X方向）が，ワールド座標系の+X軸方向を一致するように補正，③他のサンプルは基準サンプルとの姿勢距離測度が最小になるように補正，という 3 つの手順で行います．なお，時刻 0 以降の姿勢に対しては，初期姿勢に対する補正トランスフォームを適用します．

実装したコードは次の通りです．まず，基準サンプルに対する①②の補正トランスフォームを計算し，基準サンプル全体をトランスフォームします．次に，他のサンプルに対する③の補正トランスフォームを計算し，各サンプル動作をそれぞれトランスフォームします．

```
 1   void transform_entire_motion(CMotionData &mot, D3DXMATRIX adjust)
 2 ┼ {
 3 │   D3DXVECTOR3 ps, pt;
 4 │   D3DXQUATERNION qs, qt;
 5 │   D3DXMATRIX ms, mt;
 6 ┼   for (size_t f = 0;f < mot.NumFrames();++f) {
 7 │     D3DXVECTOR3 ps = mot.GetPosition(f);
 8 │     D3DXVec3TransformCoord(&pt, &ps, &adjust);
 9 │     mot.SetPosition(f, pt);
10 │
11 │     qs = mot.GetRotation(f, 0);
12 │     D3DXMatrixRotationQuaternion(&ms, &qs);
13 │     mt = ms * adjust;
14 │     mt._41 = 0; mt._42 = 0; mt._43 = 0;
15 │     D3DXQuaternionRotationMatrix(&qt, &mt);
16 │     mot.SetRotation(f, 0, qt);
17 !   }
18 ! }
19
20   void CSceneView::OnMotionSpatialAlign()
21 ┼ {
22 │   D3DXMATRIX mAdjust, ms, mt;
23 │   D3DXVECTOR3 vAdjust, vs, vt;
24 │   D3DXQUATERNION qs, qt;
25 │
26 ┼   // 基準動作の初期ルート位置を原点上に一致させる平行移動トランスフォーム
27 !   vAdjust = m_vecSampleMotions[0]->GetPosition(0);
28 │   D3DXMatrixTranslation(&mAdjust, -vAdjust.x, 0.0f, -vAdjust.z);
29 ┼   // 基準動作のローカル+X軸をワールド+X軸に一致させる回転トランスフォーム
30 !   vs = D3DXVECTOR3(1.0f, 0.0f, 0.0f);
31 │   qs = m_vecSampleMotions[0]->GetRotation(0, 0);
32 │   D3DXMatrixRotationQuaternion(&ms, &qs);
33 │   D3DXVec3TransformCoord(&vt, &vs, &ms);
34 │   D3DXMatrixRotationY(&mt, atan2f(vt.z, vt.x));
35 │   mAdjust *= mt;
36 ┼   // 全フレームを同一のトランスフォームで補正
37 !   transform_entire_motion(*m_vecSampleMotions[0], mAdjust);
38 │
39 ┼   // 基準動作に合わせて他のサンプル動作も位置，方向を変換
40 !   m_vecSampleFigures[0]->SetPose(m_vecSampleMotions[0], 0);
41 │   std::vector pc0 = m_vecSampleFigures[0]->GetPointCloud();
42 ┼   for (size_t s = 1;s < m_vecSampleFigures.size();++s) {
43 │     m_vecSampleFigures[s]->SetPose(m_vecSampleMotions[s], 0);
44 │     std::vector pc1 = m_vecSampleFigures[s]->GetPointCloud();
45 │     mAdjust = NFigureUtil::CalcTransformBetweenFigures(pc0, pc1);
46 │     transform_entire_motion(*m_vecSampleMotions[s], mAdjust);
47 │
48 │     m_vecSampleFigures[s]->SetPose(m_vecSampleMotions[s], 0);
49 !   }
50 │   SendMessage(WM_USER_RENDER_SCENE);
51 ! }
```

空間的正規化の実行例を Fig.2 に示します．なお，各キャラクタには，格子状に整列するような位置オフセットを加えてあります．左側のようにバラバラの方向と位置を示すキャラクタが，空間的正規化によって格子状に整然と並ぶ様子が確認できます．

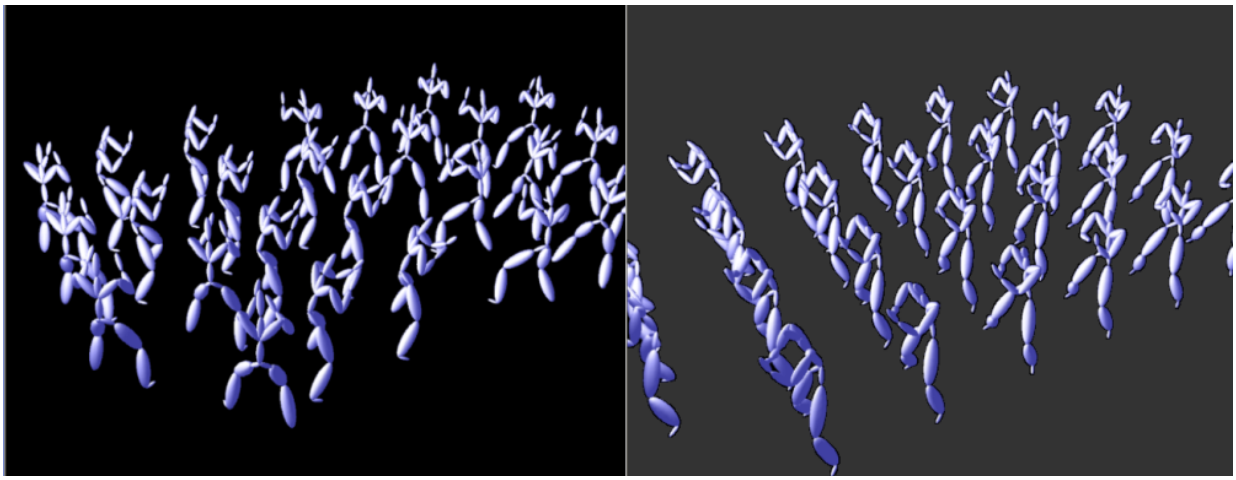

Fig.2 空間的正規化

時間的正規化

モーションブレンディングの記事で解説したように，補間計算に先立って，全てのサンプル動作の時間長やタイミングを正規化しなければなりません．ここでは，全てのサンプル動作を基準動作に同期させるように時間変換を施します．このとき，漸次的時間逆変換関数も保存しておきます．

```
 1   void CSceneView::OnMotionTemporalAlign()
 2   {
 3     CWaitCursor cur;
 4     while (!m_vecSampleITW.empty()) {
 5       delete m_vecSampleITW.back();
 6       m_vecSampleITW.pop_back();
 7     }
 8     m_vecSampleITW.push_back(new CVector(m_vecSampleMotions[0]->NumFrames(), 1.0));
 9
10     CMotionData dest;
11     for (size_t s = 1;s < m_vecSampleMotions.size();++s) {
12       m_vecSampleITW.push_back(new CVector);
13       NMotionSignal::Synchronize(dest, *m_vecSampleITW.back(), *m_vecSampleMotions[s],
14                                  *m_vecSampleMotions[0], *m_vecSampleFigures[0]);
15       *m_vecSampleMotions[s] = dest;
16     }
17     SendMessage(WM_USER_RENDER_SCENE);
18   }
```

動作パラメータの算出

各サンプル動作に対応する動作パラメータを算出します．パンチ動作に特化して実装した以下のコードは，各サンプル動作のパンチ位置:右手先の①体正面方向からの回転角度と，②地面からの高さを 2 次元の動作パラメータとして計算します．同時に動作パラメータの分布範囲，つまり各パラメータの最小値と最大値も計算しています．ここで求めた分布範囲に基づいて，動作パラメータを [-1, +1] の範囲に正規化する関数も以下に示します．

ついでに合成動作のフィギュアとモーションデータのメモリ領域を確保し，デフォルトの動作パラメータを計算しています．

```
 1   void CSceneView::OnMotionCalcParameters()
 2   {
 3     CWaitCursor cur;
 4     while (!m_vecSampleParams.empty()) {
 5       delete m_vecSampleParams.back();
 6       m_vecSampleParams.pop_back();
 7     }
 8     m_vParaRange[0][0] = DBL_MAX;
 9     m_vParaRange[0][1] = DBL_MIN;
10     m_vParaRange[1][0] = DBL_MAX;
11     m_vParaRange[1][1] = DBL_MIN;
12
13     size_t eid = m_vecSampleFigures[0]->FindNode("r_wrist_tip");
14     for (size_t s = 0;s < m_vecSampleFigures.size();++s) {
15       m_vecSampleParams.push_back(new CVector(2));
16
17       m_vecSampleFigures[s]->SetPose(m_vecSampleMotions[s], m_vecSampleMotions[s]->NumFrames() / 2);
18       D3DXVECTOR3 pos = m_vecSampleFigures[s]->GetWorldPosition(eid);
19       m_vecSampleParams.back()->Data(1) = static_cast<double>(pos.y);
20
21       m_vecSampleFigures[s]->SetPose(m_vecSampleMotions[s], 0);
22       pos -= m_vecSampleFigures[s]->GetRootPosition();
```

```
23 |      m_vecSampleParams.back()->Data(0) = static_cast<double>(atan2f(-pos.x, -pos.z));
24 |
25 |      m_vParaRange[0][0] = GSL_MIN(m_vParaRange[0][0], m_vecSampleParams.back()->Data(0));
26 |      m_vParaRange[0][1] = GSL_MAX(m_vParaRange[0][1], m_vecSampleParams.back()->Data(0));
27 |      m_vParaRange[1][0] = GSL_MIN(m_vParaRange[1][0], m_vecSampleParams.back()->Data(1));
28 |      m_vParaRange[1][1] = GSL_MAX(m_vParaRange[1][1], m_vecSampleParams.back()->Data(1));
29 !    }
30 |
31 |    if (m_pFigure != NULL)
32 |      delete m_pFigure;
33 |    m_pFigure = new CFigure(*m_vecSampleFigures.front());
34 |
35 |    if (m_pMotion != NULL)
36 |      delete m_pMotion;
37 |    m_pMotion = new CMotionData(*m_vecSampleMotions.front());
38 |    m_pFigure->SetPose(m_pMotion, 0);
39 |
40 |    m_vControlParam[0] = (m_vParaRange[0][0] + m_vParaRange[0][1]) * 0.5;
41 |    m_vControlParam[1] = (m_vParaRange[1][0] + m_vParaRange[1][1]) * 0.5;
42 |    SendMessage(WM_USER_RENDER_SCENE);
43 ! }
44
45    CVector CSceneView::NormalizeParameter(const CVector &src)
46 + {
47 |    CVector dest = src;
48 |    dest[0] = (src[0] - m_vParaRange[0][0]) / (m_vParaRange[0][1] - m_vParaRange[0][0]);
49 |    dest[1] = (src[1] - m_vParaRange[1][0]) / (m_vParaRange[1][1] - m_vParaRange[1][0]);
50 |    return dest;
51 ! }
```

算出した動作パラメータを可視化する描画関数は次のとおりです．身体中心の周り半径 80cm の円柱上にパンチ位置を描画します．そのため，ショートパンチなどでは手先が球に当たらない場合があります．

```
 1    void CSceneView::DrawParameters()
 2 + {
 3 |    m_pEffect->SetTechnique("PhongShader");
 4 |    D3DXMATRIX ms, mt;
 5 |
 6 +    // サンプルパラメータ
 7 !    m_pEffect->SetVector("g_vAmbiColor", &D3DXVECTOR4(0.8f, 0.8f, 0.5f, 1.0f));
 8 |    m_pEffect->SetVector("g_vSurfColor", &D3DXVECTOR4(0.8f, 0.8f, 0.5f, 1.0f));
 9 |    for (std::vector::const_iterator cit = m_vecSampleParams.begin();
10 +                                        cit != m_vecSampleParams.end();++cit) {
11 |      D3DXMatrixScaling(&mt, 5.0f, 5.0f, 5.0f);
12 |      D3DXMatrixTranslation(&ms, 0, 0, -80.0f);      mt *= ms;
13 |      D3DXMatrixRotationY(&ms, (*cit)->Data(0));     mt *= ms;
14 |      D3DXMatrixTranslation(&ms, 0, (*cit)->Data(1), 0); mt *= ms;
15 |      D3DXMatrixScaling(&ms, 0.1f, 0.1f, 0.1f);      mt *= ms;
16 |      DrawMeshSub(m_pMeshSphere, mt);
17 !    }
18 |
19 +    // ターゲットパラメータ
20 +    if (m_vControlParam.Norm() > 0) {
21 |      m_pEffect->SetVector("g_vAmbiColor", &D3DXVECTOR4(1.0f, 0.3f, 0.3f, 1.0f));
22 |      m_pEffect->SetVector("g_vSurfColor", &D3DXVECTOR4(1.0f, 0.3f, 0.3f, 1.0f));
23 |      D3DXMatrixScaling(&mt, 10.0f, 10.0f, 10.0f);
24 |      D3DXMatrixTranslation(&ms, 0, 0, -80.0f);      mt *= ms;
25 |      D3DXMatrixRotationY(&ms, m_vControlParam[0]);  mt *= ms;
26 |      D3DXMatrixTranslation(&ms, 0, m_vControlParam[1], 0); mt *= ms;
27 |      D3DXMatrixScaling(&ms, 0.1f, 0.1f, 0.1f);      mt *= ms;
28 |      DrawMeshSub(m_pMeshSphere, mt);
29 !    }
30 ! }
```

パンチ動作のサンプルパラメータと，その可視化結果を Fig.3 に示します．少し見難いですが，キャラクタの側上方からの視点です．各サンプルのパンチ位置に対し，黄色球にしめすパラメータが計算されたことがわかります．

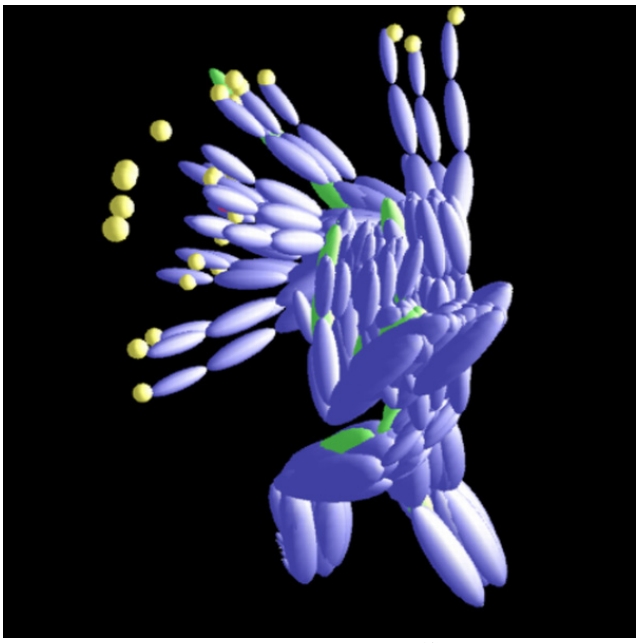

Fig.3 パンチ動作の動作パラメータ算出と可視化

動作パラメータからの動作補間

数学的な導出は補足解説で述べることにして，ここでは単純化したクリギング方程式を示します．まず，サンプル動作 M_i の動作パラメータを c_i とします．このとき，与えられた動作パラメータ c を満たす動作 $M(c) = \sum_i w_i(c) M_i$ を計算するための補間カーネル $w_i(c)$ は次の式で計算されます．

$$\begin{bmatrix} w_1(c) \\ w_2(c) \\ \vdots \\ w_N(c) \\ \lambda \end{bmatrix} = \begin{bmatrix} |c_1-c_1| & |c_1-c_2| & \cdots & |c_1-c_N| & 1 \\ |c_2-c_1| & |c_2-c_2| & & |c_2-c_N| & 1 \\ \vdots & & \ddots & \vdots & \vdots \\ |c_N-c_1| & \cdots & \cdots & |c_N-c_N| & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix}^{-1} \begin{bmatrix} |c_1-c| \\ |c_2-c| \\ \vdots \\ |c_N-c| \\ 1 \end{bmatrix}$$

サンプルパラメータ間のノルムによって構成される行列の逆行列と，与えられた動作パラメータとサンプルパラメータ間のノルムによって構成されるベクトルの積により，各サンプルの補間カーネルが求められます．なお，λ はラグランジュ乗数であり，行列やベクトルに付随する 1 や 0 と一緒に作用して，補間カーネルの総和を $\sum_i w_i(c) = 1.0$ に拘束するために導入されたものです．

ここで注意すべき点は，上式の動作パラメータは NormalizeParameter 関数によって [-1.0, +1.0] に正規化されたものを用いることです．これはレンジの異なる動作パラメータを同時に扱うための処理です．例えばパンチ動作の動作パラメータは，正面からの回転角度(radian)と地面からの高さ(cm)を用います．しかし，この 2 つのデータレンジには 100 倍ほどの開きがあるため，そのままでは動作パラメータ間のノルムの計算において回転角度がほとんど無視される結果になり，所望の補間結果が得られません．こうした不具合を軽減するため，動作パラメータのレンジを正規化する必要があります．

以下の OnMotionAnalyze 関数では上式の逆行列を計算し，OnMotionSynthesis 関数では右側のベクトルの計算と補間カーネルの算出，そして実際の動作補間処理を行います．

```
 1  void CSceneView::OnMotionAnalyze()
 2  {
 3    if (m_pKrigingSystem != NULL)
 4      delete m_pKrigingSystem;
 5    m_pKrigingSystem = new CMatrix(m_vecSampleParams.size() + 1, m_vecSampleParams.size() + 1, 1.0);
 6
 7    for (size_t r = 0;r < m_pKrigingSystem->Row() - 1;++r) {
 8      CVector vr = NormalizeParameter(*m_vecSampleParams[r]);
 9      for (size_t c = r;c < m_pKrigingSystem->Col() - 1;++c) {
10        CVector vc = NormalizeParameter(*m_vecSampleParams[c]);
11        m_pKrigingSystem->Data(r, c) = (vr - vc).Norm();
12        m_pKrigingSystem->Data(c, r) = m_pKrigingSystem->Data(r, c);
13      }
14    }
15    m_pKrigingSystem->Data(m_pKrigingSystem->Row() - 1, m_pKrigingSystem->Col() - 1) = 0.0;
16    *m_pKrigingSystem = m_pKrigingSystem->InverseLUD();
17    SendMessage(WM_USER_RENDER_SCENE);
```

```
18 ! }
```

```
 1   void CSceneView::OnMotionSynthesis()
 2 ┼ {
 3 │   CVector cv(m_vecSampleParams.size() + 1, 1.0);
 4 │   CVector npara = NormalizeParameter(m_vControlParam);
 5 ┼   for (size_t s = 0;s < m_vecSampleParams.size();++s) {
 6 │     CVector vp = NormalizeParameter(*m_vecSampleParams[s]);
 7 │     cv[s] = (vp - npara).Norm();
 8 !   }
 9 │
10 │   CVector weight = *m_pKrigingSystem * cv;
11 │   CVector itw(m_vecSampleITW.front()->Size(), 0);
12 │   std::vector<float> kernel(weight.Size() - 1);
13 ┼   for (size_t s = 0;s < kernel.size();++s) {
14 │     kernel[s] = weight[s];
15 │     itw += *m_vecSampleITW[s] * kernel[s];
16 !   }
17 │
18 │   CMotionData dest;
19 │   NMotionSignal::Interpolate(dest, m_vecSampleMotions, kernel);
20 │   NMotionSignal::IncrementalTimewarp(*m_pMotion, dest, itw);
21 │   m_pFigure->SetPose(m_pMotion, m_frame);
22 │   SendMessage(WM_USER_RENDER_SCENE);
23 ! }
```

結果

パンチ動作については，実際に遊んでみて効果をご確認ください．サンプルの分布範囲を外れると急速に誤差が増加する様子が確認できると思います．別種類の動作のパラメータ化結果についてはアチラやコチラをご参照ください．

まとめ

モーションブレンディングの発展手法であるモーションパラメータ化法を説明しました．動作パラメータから最適な補間カーネルを算出するアルゴリズムについては，以下の補足説明を参照ください．モーションパラメータ化法は比較的シンプルなアルゴリズムであり，動作パラメータの設計次第で様々な操作インターフェースを実現できます．

ただし，あくまでも内挿法(interpolation)に基づく技法なので，外挿(extrapolation)に弱いという欠点(といえる？)があります．また，理論的にも必ず補間誤差が生じることに注意が必要です．ミリ単位の補間精度が必要な場合には IK による姿勢補正や，論文[3]で用いられているような擬似サンプル法などを用いることで誤差を改善できます．あとは，本質的な問題ではありませんが，時間変換処理の遅さはやはり大きなネックになります．

補足: 数学的背景

最小二乗法にもとづく統計的なデータ補間法について，おおまかな補足説明を記します．ここではデータ間の相関強度を表す共分散にもとづき，地球統計学の基本技法であるクリギングの基本アルゴリズムを導出します．

以下の説明はなるべく短く，直観的にクリギング方程式を導出することが目的なので，統計学的に厳密でなかったり誤りを含む可能性があります．いくつかの重要な仮定や前提条件も説明を省略している点にご注意ください．(ご指摘やご叱責は歓迎します)

補足1: 前提知識- 共分散

共分散(covariance)とは，2 つの変量間の相関の大きさ(共変動)を表す指標です．例えば N 個の変量 a_i と b_i，それぞれの平均値 $\bar{a}_i$ と $\bar{b}_i$ が与えられたとき，a と b の共分散 $cov(a.b)$ は次式で計算できます．

$$cov(a.b) = \frac{1}{N}\sum_{i}^{N}(a_i - \bar{a})(b_i - \bar{b})$$

共分散は個々のサンプルについても定義でき，各サンプル間の相関の大きさを表す指標として用いられます．

$$cov(a_i.b_j) = (a_i - \bar{a})(b_j - \bar{b})$$

ここで，単変量についての共分散を考えます．

$$cov(a_i.a_j)=(a_i-\bar{a})(a_j-\bar{a})$$

なお，$i=j$ の共分散は各サンプルの分散（variance）$var(a_i)=(a_i-\bar{a})^2$ に一致します．

また，全てのサンプル対についての共分散もしくは分散を計算し，対称行列にまとめたものは分散共分散行列と呼ばれます．

$$\begin{bmatrix} var_1 & cov_{12} & \cdots & cov_{1N} \\ cov_{21} & var_2 & & cov_{2N} \\ \vdots & & \ddots & \vdots \\ cov_{N1} & \cdots & \cdots & var_N \end{bmatrix}$$

ここで $cov_{ij}=cov(a_i.a_j)$，$var_i=var(a_i)$ です．

補足2：サンプルの補間によるオリジナルデータの近似 [†]

N 個のサンプルベクトルデータ X_i の加重和によって任意のベクトルデータ Y を近似する問題を考えます．

$$Y \approx \sum_i^N w_i X_i$$

この問題は，$\sum_i^N w_i=1.0$ という拘束条件のもとで加重係数 w_i についての最小二乗問題として解くことができます．

$$w=\mathrm{argmin}_w |Y-\sum_i^N w_i X_i|^2$$

この最小化の目的関数は，$\sum_i^N w_i=1.0$ を用いて次のように変形できます．

$$f(w)=|\sum_i^N w_i(X_i-Y)|^2=|w_1(X_1-Y)+w_2(X_2-Y)+\cdots+w_N(X_N-Y)|^2$$

この平方和の計算式を展開することで，次の目的関数が得られます．

$$f(w)=\sum_i\sum_j w_i w_j(X_i-Y)(X_j-Y)$$

ここで，オリジナルデータ Y がサンプルデータの平均値に等しく $Y=\overline{X}=\frac{1}{N}\sum_i Xi$ が成り立つと仮定すると，上式は X_i の共分散を用いて次式のように書き換えられます．

$$f(w)=\sum_i\sum_j w_i w_j cov(X_i.X_j)$$

この共分散 $cov(X_i.X_j)$ は，オリジナルデータ Y を中心とした共分散であることから，局所的共分散と呼ばれます．

次に，この目的関数 $f(w)$ を最小化する加重係数 w_i を最小二乗法により計算します．この最小化問題は，w_i に関して $f(w)$ の一次偏微分値を全て 0 にする解を求める問題と同義です．

$$\frac{\partial f(w)}{\partial w_1}=0,\ \frac{\partial f(w)}{\partial w_2}=0,\ \cdots\cdots,\ \frac{\partial f(w)}{\partial w_N}=0$$

w_1 についての一次偏微分を導出すると，次式の関係が得られます．

$$\frac{\partial}{\partial w_1}\sum_i\sum_j w_i w_j cov(X_i.X_j)=2\sum_j w_j cov(X_1.X_j)=0$$

この関係を全ての w_i について求め，行列とベクトルの積としてまとめると，分散共分散行列を用いた線形システムが得られます．

$$\begin{bmatrix} cov_{11} & cov_{12} & \cdots & cov_{1N} \\ cov_{21} & cov_{22} & & cov_{2N} \\ \vdots & & \ddots & \vdots \\ cov_{N1} & \cdots & \cdots & cov_{NN} \end{bmatrix}\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

ここで $cov_{ij}=cov(X_i.X_j)$ です．ただし，この線形システムの解は $\sum_i w_i=1.0$ の拘束条件を満たしません．そこでラグランジュの未定乗数法により，拘束条件を導入した目的関数を作成します．すなわち，ラグランジュ乗数 λ を用いて最小化の目的関数を次のように修正します．

$$w=\mathrm{argmin}_w|Y-\sum_i w_i X_i|^2+2\lambda(\sum_i w_i-1)$$

$$f(w)=\sum_i\sum_j w_i w_j cov(X_i.X_j)+2\lambda(\sum_i w_i-1)$$

$$\frac{\partial f(w)}{\partial w_i}=2\sum_j w_j cov(X_i.X_j)+2\lambda=0,\quad \frac{\partial f(w)}{\partial \lambda}=\sum_i w_i-1=0$$

したがって，目的関数を最小化する w_i と λ は，次の線形システムを解くことで求められます．

$$\begin{bmatrix} cov_{11} & cov_{12} & \cdots & cov_{1N} & 1 \\ cov_{21} & cov_{22} & & cov_{2N} & 1 \\ \vdots & & \ddots & \vdots & \vdots \\ cov_{N1} & \cdots & \cdots & cov_{NN} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix}\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \\ \lambda \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

この線形システムを実装したテストプログラムを示します．main 関数内の SAMPLES がサンプルの数，DIMS がデータベクトルの次元数を表します．SAMPLES>>DIMSのときは高精度でオリジナルデータを近似できますが，SAMPLES<=DIMSでは誤差が増加することが確認できると思います．

cov.cpp

```
#include
#include
#include
#include
#include

#ifdef _DEBUG
#pragma comment(lib, "gsld.lib")
#pragma comment(lib, "gslcblasd.lib")
#else
#pragma comment(lib, "gsl.lib")
#pragma comment(lib, "gslcblas.lib")
#endif

void print_matrix(const char *title, const gsl_matrix *m)
{
    printf("%s¥n", title);
    for (size_t r = 0;r < m->size1;++r)
    {
        printf("%6.3f", gsl_matrix_get(m, r, 0));
        for (size_t c = 1;c < m->size2;++c)
            printf(" %6.3f", gsl_matrix_get(m, r, c));
        putchar('¥n');
    }
    putchar('¥n');
}
```

```
27
28  void print_vector(const char *title, const gsl_vector *v)
29  {
30      printf("%s¥n", title);
31      printf("%6.3f", gsl_vector_get(v, 0));
32      for (size_t s = 1;s < v->size;++s)
33          printf(" %8.5f", gsl_vector_get(v, s));
34      putchar('¥n');
35      putchar('¥n');
36  }
37
38  int main(void)
39  {
40      const size_t SAMPLES = 7;
41      const size_t DIMS = 6;
42
43      // サンプル行列
44      gsl_matrix *data = gsl_matrix_alloc(DIMS, SAMPLES);
45      for (size_t r = 0;r < data->size1;++r)
46      for (size_t c = 0;c < data->size2;++c)
47          gsl_matrix_set(data, r, c, rand() * 5.0 / RAND_MAX);
48      print_matrix("data matrix", data);
49
50      // ターゲットベクトル
51      gsl_vector *target = gsl_vector_alloc(DIMS);
52      for (size_t r = 0;r < target->size;++r)
53          gsl_vector_set(target, r, rand() * 5.0 / RAND_MAX);
54      print_vector("target vector", target);
55
56      // 差分行列とその転置行列
57      gsl_matrix *nm = gsl_matrix_calloc(data->size1, data->size2);
58      for (size_t r = 0;r < nm->size1;++r)
59      for (size_t c = 0;c < nm->size2;++c)
60          gsl_matrix_set(nm, r, c, gsl_matrix_get(data, r, c) - gsl_vector_get(target, r));
61      gsl_matrix *nmt = gsl_matrix_calloc(data->size2, data->size1);
62      gsl_matrix_transpose_memcpy(nmt, nm);
63      print_matrix("residual matrix", nm);
64
65      // 分散共分散行列（ラグランジュの未定乗数法）
66      gsl_matrix *lc = gsl_matrix_alloc(data->size2 + 1, data->size2 + 1);
67      gsl_matrix_set_all(lc, 1.0);
68      gsl_matrix_set(lc, data->size2, data->size2, 0);
69      gsl_matrix_view lcv = gsl_matrix_submatrix(lc, 0, 0, data->size2, data->size2);
70      gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 1.0, nmt, nm, 0, &lcv.matrix);
71      print_matrix("local covariance matrix", lc);
72
73      // 重み係数の算出
74      gsl_vector *vw = gsl_vector_calloc(lc->size1);
75      gsl_vector *v1 = gsl_vector_calloc(lc->size1);
76      gsl_vector_set(v1, v1->size - 1, 1.0);
77      gsl_permutation *p = gsl_permutation_alloc(lc->size1);
78      int sign;
79      // "lc * vw = v1"を満たすvwの最小二乗解を計算
80      gsl_linalg_LU_decomp(lc, p, &sign);
81      gsl_linalg_LU_solve(lc, p, v1, vw);
82      gsl_permutation_free(p);
83      print_vector("weight coefficients", vw);
84
85      // ターゲットベクトルの再計算
86      gsl_vector *vr = gsl_vector_calloc(target->size);
87      for (size_t r = 0;r < data->size1;++r)
88      for (size_t c = 0;c < data->size2;++c)
89          *gsl_vector_ptr(vr, r) += gsl_vector_get(vw, c) * gsl_matrix_get(data, r, c);
90      print_vector("resulting vector", vr);
91
92      gsl_matrix_free(data);
93      gsl_vector_free(target);
94      gsl_matrix_free(nm);
95      gsl_matrix_free(nmt);
96      gsl_matrix_free(lc);
97      gsl_vector_free(v1);
98      gsl_vector_free(vw);
99      gsl_vector_free(vr);
100
101     return EXIT_SUCCESS;
102 }
```

補足3: クリギングへの発展 [†]

サンプルデータ X_i とそれに対応したパラメータ c_i を用いて，任意のパラメータ c を満たすデータ $X(c)$ を計算する問題を考えます．

$$X(c)=\sum_{i}^{N}w_i(c)X_i$$

最も単純な方法としては，サンプルパラメータ c_i から目的のパラメータ c を近似するための加重係数を用いる方法が考えられます．つまり，$c=\sum_i w_i c_i$ を満たす w_i によって，$X(c)=\sum_i w_i X_i$ を計算する方法です．しかしこの方法は，単にパラメータ間の相関を考慮するものでデータ間の相関は無視されるため，誤った計算結果を導きます．

そこで，補足 2 と同様に最小二乗法による解法を用います．最小化すべき目的関数は，補足 2 の手順にしたがって次のように導かれます．

$$w(c)=\mathrm{argmin}_w f(w(c))$$

$$f(w(c))=\sum_i\sum_j w_i(c)w_j(c)(X_i-X(c))(X_j-X(c))$$

補足 2 では，この目的関数を $X(c)$ がサンプルデータの平均値であるという仮定をおき，局所的共分散を用いた解を導出しました．しかし．ここでは $X(c)$ が不明なので局所共分散は計算できません．仕方ないので目的関数をそのまま展開します．

$$f(w(c))=\sum_i\sum_j(w_i(c)w_j(c)X_iX_j)-2\sum_i(w_i(c)X_iX(c))+X^2(c)$$

次に，この目的関数の $w_i(c)$ に関する一次偏微分を求めます．

$$\frac{\partial f(w(c))}{\partial w_i(c)}=2\sum_j w_j(c)(X_iX_j)-2X_iX(c)=0$$

ここで，データの平均値が $\overline{X}=0$ であると仮定すると，$X_iX_j=(X_i-\overline{X})(X_j-\overline{X})=cov(X_i.X_j)$ が成り立ちます．

$$\frac{\partial f(w(c))}{\partial w_i(c)}=2\sum_j w_j(c)cov(X_i.X_j)-2cov(X_i.X(c))=0$$

ただし，この第 2 項は $X(c)$ が不明なので計算できません．そこで，2 つのデータ間の共分散は対応するパラメータの相対関係 c_i-c_j によって算出できると考え，パラメータの差分によって共分散を計算するモデル，すなわち共分散関数を導入します．

$$C(c_i-c_j)=cov(X_iX_j)$$

共分散関数を用いて書き換えた一次偏微分の式を示します．

$$\frac{\partial f(w(c))}{\partial w_i(c)}=\sum_j w_j(c)C(c_i-c_j)-C(c_i-c)=0$$

したがって，ラグランジュの未定乗数法を導入した補間カーネルの線形システムは次のようにまとめられます．

$$\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1N} & 1 \\ C_{21} & C_{22} & & C_{2N} & 1 \\ \vdots & & \ddots & \vdots & \vdots \\ C_{N1} & \cdots & \cdots & C_{NN} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix}\begin{bmatrix} w_1(c) \\ w_2(c) \\ \vdots \\ w_N(c) \\ \lambda \end{bmatrix}=\begin{bmatrix} C(c_1-c) \\ C(c_2-c) \\ \vdots \\ C(c_N-c) \\ 1 \end{bmatrix}$$

ここで $C_{ij}=C(c_i-c_j)$ です．

さらにこの線形システムにおける共分散関数は，データ間の非類似度とパラメータの関係を表すモデルであるバリオグラム（variogram）$\gamma(c_i-c_j)$に置き換えることが可能です．これは，共分散関数とバリオグラムの間に $\gamma(c_i-c_j)=C(0)-C(c_i-c_j)$ なる関係が成り

立つことから導かれます.

$$\begin{bmatrix} \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1N} & 1 \\ \gamma_{21} & \gamma_{22} & & \gamma_{2N} & 1 \\ \vdots & & \ddots & \vdots & \vdots \\ \gamma_{N1} & \cdots & \cdots & \gamma_{NN} & 1 \\ 1 & 1 & \cdots & 1 & 0 \end{bmatrix} \begin{bmatrix} w_1(c) \\ w_2(c) \\ \vdots \\ w_N(c) \\ \lambda \end{bmatrix} = \begin{bmatrix} \gamma(c_1 - c) \\ \gamma(c_2 - c) \\ \vdots \\ \gamma(c_N - c) \\ 1 \end{bmatrix}$$

ここで $\gamma_{ij} = \gamma(c_i - c_j)$ です.

このアプローチにもとづくデータ補間法をクリギングとよび, この線形システムはクリギング方程式と呼ばれます. クリギングを用いるための前提条件を改めてまとめておきます.

- サンプルデータ X_i の平均値は $\overline{X} = \frac{1}{N}\sum_i X_i = 0$
- バリオグラム(共分散関数)は 2 つのパラメータの相対関係 $(c_i - c_j)$ のみによって決定.

もっとも単純なクリギングでは, バリオグラム γ はパラメータ間の距離のみによって計算します. たとえば, 動作パラメータ化のサンプルプログラムでは, $\gamma(c_i - c_j) = |c_i - c_j|$ というように, パラメータ間の距離にしたがってデータ非類似度が線形に増加するという, かなりシンプルなバリオグラムを用いています. しかし, 本来はデータ分布を反映したバリオグラムを推定しなければなりません. このあたりの詳細は本サイトの範疇外なので, 専門書や統計関係のウェブページ等を参照ください.
