モーションマッピング

http://www.tmps.org/index.php?%A5%E2%A1%BC%A5%B7%A5%E7%A5%F3%A5%DE%A5%C3%A5%D4%A5%F3%A5%B0

今回は，モーションデータを 2 次元平面上にマッピングするアプリケーションを作成しました．このシステムでは，Fig.1 のスナップショットに示すように，サンプルデータのキャラクタ姿勢と 2 次元マップ上の点が一対一で対応し，動作の時間経過がその軌跡として表されます．動作によってさまざまな軌跡が描かれるため，軌跡の形状を見ることで動作の大まかな特徴を把握することができます．また，マップ上の軌跡をスケッチすることで，新しい動作を合成することも可能です．使いやすさや実用性はともかく，お手軽にキャラクタアニメーションを制作できると思います．

なお，本記事はこちらの事項を踏まえて修正する予定です（2006/12/15）．

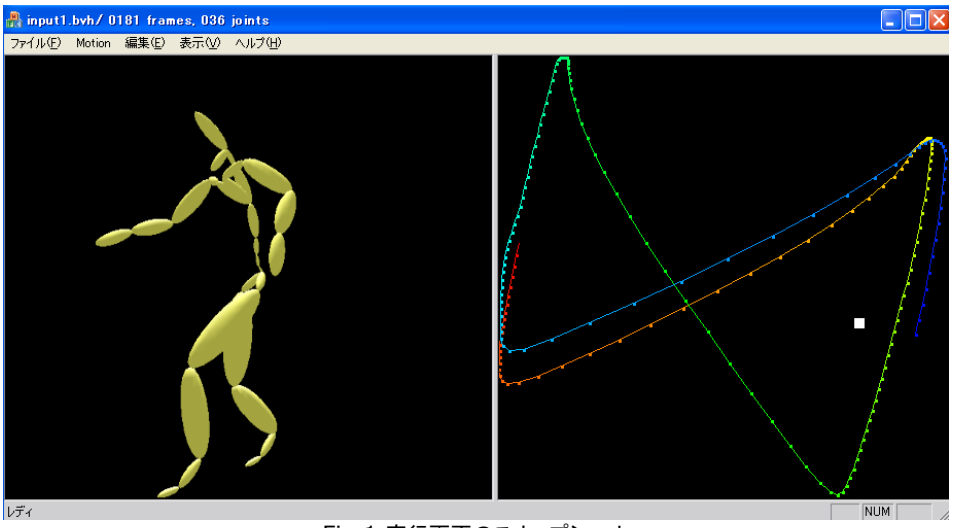


Fig.1 実行画面のスナップショット

- 記事の修正記録
- サンプルプログラム
  - 遊び方
- 参考文献
- マッピングアルゴリズム
  - 固有値分解，特異値分解
  - 主成分分析と特異値分解による次元数削減
  - 多次元尺度構成法：Multi-Dimensional Scaling
  - Isomap
  - Laplacian Eigenmap
  - Locally Linear Embedding
- モーションデータのマッピング
- 評価実験
- マップ上のスケッチによる動作合成
- まとめ

記事の修正記録

- 2008.06.27: SVD のバグの修正
  - サンプルプログラム（アーカイブ形式）は未対応です．本記事をご参考に修正願います．

サンプルプログラム

サンプルプログラムは，VisualC++ 2005 + DirectX SDK Update June 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 と Boost Libraries が必要です．アーカイブには，歩行動作の BVH モーションキャプチャデータも含めてあります．今回は，あれこれ面倒そうだったので DXUT 版は作成していませんが，そのかわりにバイナリデータをアップしておきます．DirectX SDK (June 2006) もしくは DirectX End User Runtime (June 2006) をインストールして遊んでみてください．

サンプルソースコード(VC++2005, MFC, 207kb)
サンプルバイナリ(200kb)

遊び方

1. 実行形式ファイル（MotionManifold.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 BVH モーションデータファイル，もしくは ASF-AMC モーションデータファイルをロードします．

3. [Motion]メニューからお好みのアルゴリズムを選択すると，右ウィンドウにモーションデータがマッピングされ，軌跡が表示されます．
   i. [ファイル]→[再生]を選択すると，左ウィンドウでアニメーションが再生され，右ウィンドウでは白ポインタが軌跡上を移動します．
   ii. 再生が停止した状態で，右ウィンドウ上でマウスを左クリックしながら移動させると，左ウィンドウに合成姿勢が表示されます．

参考文献

1. Hyun Joon Shin and Jehee Lee, Motion Synthesis and Editing in Low-Dimensional Spaces, Computer Animation and Virtual Worlds (CASA2006), vol.17, no.3-4, pp.219-227, 2006.
2. Keith Grochow, Steven L. Martin, Aaron Hertzmann, and Zoran Popovi醇J, Style-based Inverse Kinematics, ACM Transactions on Graphics (SIGGRAPH 2004), vol.23, no.3, pp.522-531, 2004.
3. Jackie Assa, Yaron Caspi, and Daniel Cohen-Or, Action Synopsis: Pose Selection and Illustration, ACM Transactions on Graphics (SIGGRAPH 2005), vol.24, no.3, pp.667-676, 2005.

今回実装したシステムは，論文(1)をベースにしつつ，姿勢合成のアルゴリズムを改良したものです．ただし，論文で提案されている「虫めがね機能」は実装していません．また，低次元空間内でのスケッチからアニメーションデータを作成するシステムは，論文(2)などが有名です．論文(3)ではモーションデータを低次元空間に写像することで，動作の特徴を最も良く表すキーフレームを効率的に検索するシステムを提案しています．

マッピングアルゴリズム

モーションデータを 2 次元平面上にマッピングするためには，データの次元数＝関節自由度数を効果的に削減する必要があります．こうした次元数削減問題は，統計学や機械学習分野の主要な課題の一つであり，これまでにも多数のアルゴリズムが提案されています．今回実装したシステムでは，4 つの多様体学習アルゴリズムに主成分分析法と特異値分解法を加えた，計 6 つのアルゴリズムを選択して用いることが出来ます．

主成分分析法と特異値分解法では，変量間＝関節間の相関関係を利用して次元数を削減します．例えば，あるモーションデータにおいて肘関節と肩関節が常に同じ傾向を持って連動しているならば，その相関関係のもとで 2 つの関節運動を 1 つのパラメータで記述できるはずです．こうした考えに基づき，主成分分析や特異値分解を用いた次元数削減法では，関節間の相関関係を利用することで動作を少数のパラメータで表現します．

一方，多様体学習アルゴリズムは，元の多次元空間におけるデータ間の距離をなるべく保つように，データを低次元空間に再配置する手法です．古典的な方法としては，多次元尺度構成法(Multi-Dimensional Scaling, MDS)と呼ばれる方法が知られていますが，より複雑なデータ構造に対応するためのアルゴリズムが広く研究されています(多様体学習法リソースページ)．今回は代表的なアルゴリズムである MDS と Isomap, Laplacian Eigenmap(LE), Locally Linear Embedding(LLE)を実装してみました．

固有値分解，特異値分解

今回実装した全てのアルゴリズムは，固有値分解もしくは特異値分解を利用しているので，これらは共通ルーチンとして実装しておきます．

まず，GNU Scientifc Library の API を利用した固有値分解ルーチンを示します．引数の行列 data を固有値分解し，降順にソートされた固有値をベクトル evals に，行列 evecs の各列ベクトルに固有ベクトルを格納します．

なお，GSL のリファレンスでは，GSL の API のかわりに LAPACK の利用することが推奨されています．計算精度にも問題がある可能性があるので，LAPACK に慣れた方はコードを置き換えたほうがよいかもしれません．

```
1  bool NStatistics::EVD(CMatrix &evecs, CVector &evals, const CMatrix &data)
2  {
3    evecs.Initialize(data.Row(), data.Col());
4    evals.Initialize(data.Row());
5    CMatrix tmp = data;
6
7    // 固有値分解
8    gsl_eigen_symmv_workspace *w = gsl_eigen_symmv_alloc(data.Row());
9    int bsucc = gsl_eigen_symmv(tmp.m_pmMatrix, evals.m_pvVector, evecs.m_pmMatrix, w);
10   gsl_eigen_symmv_free(w);
11
12   if (bsucc == GSL_SUCCESS) {
13     // 固有値の絶対値によって降順ソート
14     gsl_eigen_symmv_sort(evals.m_pvVector, evecs.m_pmMatrix, GSL_EIGEN_SORT_ABS_DESC);
15     return true;
16   }
17   evecs.Destroy();
18   evals.Destroy();
19   return false;
20 }
```

次に特異値分解のルーチンを示します．非正方行列を効率的に扱うためのトリックを含んでいますが，基本的には GSL の API を呼び出しているだけです．なお，このコードではエコノミーサイズの特異値分解を行います．フルサイズの特異値分解を行うためにはコードの修正が必要なので注意してください．

```
1  bool NStatistics::SVD(CMatrix &u, CVector &s, CMatrix &v, const CMatrix &data)
2  {
3    if (data.Row() < data.Col())
```

```
    u = data * data.Transpose();
  else if (data.Row() > data.Col())
    u = data.Transpose() * data;
  else
    u = data;

  v.Initialize(u.Col(), u.Col());
  s.Initialize(u.Col());

  gsl_vector *vW = gsl_vector_alloc(u.Col());
  int error_code = gsl_linalg_SV_decomp(u.m_pmMatrix, v.m_pmMatrix, s.m_pvVector, vW);
  gsl_vector_free(vW);

  if (error_code != GSL_SUCCESS) {
    u.Destroy();
    s.Destroy();
    v.Destroy();
    return false;
  }

  if (data.Row() == data.Col())
    return;

  CMatrix is(s.Size(), s.Size(), 0);
  for (size_t i = 0;i < s.Size();++i) {
    s[i] = sqrt(s[i]);
    is(i, i) = s[i] > 0 ? 1.0 / s[i] : 0;
  }

  if (data.Row() < data.Col())
    v = data.Transpose() * u * is;
  else if (data.Row() > data.Col())
    u = data * v * is;

  return true;
}
```

主成分分析と特異値分解による次元数削減 [†]

主成分分析には，変量の相関行列を固有値分解する方法もありますが，今回は標準化されたデータ行列を特異値分解するアルゴリズムを採用しました．サンプルデータを各行ベクトルに格納した行列 data を主成分分析し，主成分スコア行列 scores，基底変換行列 rot，寄与率ベクトル cont，サンプルの平均ベクトル mean を計算します．

- 行列 scores
  - 各列ベクトルが主成分スコアに対応
  - 第 1 列から順に，第 1 主成分スコア，第 2 主成分スコア...と昇順に格納
- 基底変換行列 rot
  - 標準化されたデータを部分空間に写像するための基底変換行列．
- 寄与率 cont
  - 先頭から順に，第 1 主成分の寄与率，第 2 主成分の寄与率...と昇順に格納

```
bool NStatistics::PCA(CMatrix &scores, CMatrix &rot, CVector &cont, CVector &mean, const CMatrix &data)
{
  // 中心化
  CMatrix nm(data.Row(), data.Row(), -1.0 / data.Row());
  for (size_t i = 0; i < data.Row(); ++i)
    nm(i, i) += 1.0;
  CMatrix centered = nm * data;

  // 平均値ベクトル
  mean.Initialize(data.Row(), 1.0 / data.Row());
  mean = mean * data;

  // 標準化
  CMatrix scaled = centered;
  for (size_t c = 0;c < scaled.Col();++c) {
    CVector v = scaled.ColVector(c);
    if (v.Norm() > 0)
      v *= sqrt(v.Size() - 1.0) / v.Norm();
    for (size_t r = 0;r < scaled.Row();++r)
      scaled(r, c) = v[r];
  }

  // 標準化された行列の特異値分解
  CMatrix tmp;
  if (!NStatistics::SVD(tmp, cont, rot, scaled)) {
    scores.Destroy();
```

```
27 |      rot.Destroy();
28 |      cont.Destroy();
29 |      mean.Destroy();
30 |      return false;
31 !    }
32 |    cont *= 1.0 / cont.Size();
33 |
34 |    // 主成分スコアの算出
35 !    scores = centered * rot;
36 |    return true;
37 ! }
```

主成分分析を利用した次元数削減は，主成分スコアを所望の次元数で打ち切ることで実現されます．つまり，次元数を 2 次元に削減するためには，第 3 主成分スコア以降を打ち切ることになります．したがって，各サンプルデータに対応する第 1 主成分スコアと第 2 主成分スコアのペアが，各サンプルを 2 次元平面上にマッピングした座標となります．

```
 1    bool NStatistics::EmbedPCA(CMatrix &coords, const CMatrix &data, size_t dim)
 2  {
 3 |    CVector cont, mean;
 4 |    CMatrix scores, rot;
 5 |    if (!NStatistics::PCA(scores, rot, cont, mean, data)) {
 6 |      coords.Destroy();
 7 |      return false;
 8 !    }
 9 |
10 |    coords.Initialize(data.Row(), dim);
11 |    for (size_t r = 0;r < data.Row();++r)
12 |    for (size_t c = 0;c < dim;++c)
13 |      coords(r, c) = scores(r, c);
14 |
15 |    return true;
16 ! }
```

特異値分解を通じた次元数削減は，主成分分析の場合と同様に，高次の特異値を打ち切ることで実現されます．コードをよく見ると気付くと思いますが，次元数削減のための主成分分析と特異値分解の違いは，データ行列の標準化の有無だけです．後で実験結果に示すとおり，主成分分析のほうがデータ分布構造を把握しやすいマッピングを与えるようですが，特異値分解を用いる方法では平均値ベクトルの計算が不要なのでコーディングやデータの扱いは容易になります．

```
 1    bool NStatistics::EmbedSVD(CMatrix &coords, const CMatrix &data, size_t dim)
 2  {
 3 |    CMatrix u, v;
 4 |    CVector s;
 5 |    if (!NStatistics::SVD(u, s, v, data)) {
 6 |      coords.Destroy();
 7 |      return false;
 8 !    }
 9 |
10 |    CMatrix rotated = data * v;
11 |
12 |    coords.Initialize(data.Row(), dim);
13 |    for (size_t r = 0;r < coords.Row();++r)
14 |    for (size_t c = 0;c < dim;++c)
15 |      coords(r, c) = rotated(r, c);
16 |
17 |    return true;
18 ! }
```

多次元尺度構成法：Multi-Dimensional Scaling [†]

多次元尺度構成法は，全てのサンプルデータ間の距離関係を保つように，データを低次元空間に再配置する手法です．このアルゴリズムでは，データ間の相対距離だけを与えればよいので，各サンプルの値そのものが不明な場合でも利用可能です．例えば，日本の県庁所在地間の距離が与えられた時に，各都市の位置を再現することができます．したがって，相対的な距離尺度しか計測できないような場合でもデータの分布構造を推定できるので，心理学や感性工学などでも広く利用されているようです．詳しいアルゴリズムや理論的背景は，多次元尺度構成法講義ノートがとても参考になります．以下のコードも，こちらの文献を参考にして実装しました．

なお，サンプルデータベクトルのユークリッド距離を用いた多次元尺度構成法は，主成分分析と同一の結果を与えることが知られています．そのため，主成分分析に対する多次元尺度構成法の優位点は，ユーザの目的に応じた距離関数を定義することでマッピングを変更できる点だと考えられます．

多次元尺度構成法を用いた次元数削減コードを以下に示します．各サンプル間の距離を格納した距離行列 dist を引数として渡します．距離行列の各成分 $dist(i,j)$ はそれぞれ，サンプル i とサンプル j 間の距離を示します．そのため，$dist(i,j) = dist(j,i)$ となるので，距離行列は必ず対称行列となります．もちろん，対角成分 $dist(i,i) = 0$ です．

```
 1    bool NStatistics::EmbedMDS(CMatrix &coords, const CMatrix &dist, size_t dim)
 2  {
```

```
  if (dist.Row() != dist.Col())
    return false;

  // 距離行列の2乗
  CMatrix dist2(dist.Row(), dist.Col());
  for (size_t r = 0;r < dist2.Row();++r)
  for (size_t c = 0;c < dist2.Col();++c)
    dist2(r, c) = dist(r, c) * dist(r, c);

  // 内積行列
  CMatrix nm(dist.Row(), dist.Col(), -1.0 / dist.Row());
  for (size_t i = 0;i < dist.Row();++i)
    nm(i, i) += 1.0;
  CMatrix ipm = nm * dist2 * nm.Transpose() * -0.5;

  // 内積行列の特異値分解
  CMatrix mu, mv;
  CVector vs;
  if (!NStatistics::SVD(mu, vs, mv, ipm)) {
    coords.Destroy();
    return false;
  }

  coords.Initialize(dist.Row(), dim);
  for (size_t r = 0;r < dist.Row();++r)
  for (size_t c = 0;c < dim;++c)
    coords(r, c) = mu(r, c) * sqrt(vs[c]);

  return true;
}
```

Isomap

Isomap は 2000 年の Science 誌に掲載された非線形次元数削減アルゴリズムで，データの局所的な分布構造を考慮するように改良された多次元尺度構成法です(Isomap 本家ホームページ)．アルゴリズムは次の3ステップで表されます．

1. 各サンプルデータとその近傍のデータをエッジ接続することで，グラフネットワークを構築．
    - 近傍データの探索には，k最近傍法(k-Nearest Neighbor)や近傍半径を利用した方法が用いられる．
2. グラフ上の最小距離(測地距離)によって，距離行列を再計算．
3. 多次元尺度構成法により，低次元空間におけるデータ座標を決定．
    - 近傍数もしくは近傍半径はユーザが決定
    - 近傍数＝(全データ数 - 1)などとして全てのデータをエッジ接続する場合には，当然 Isomap は多次元尺度構成法と同一の結果を算出．

次のコードは，最近傍法を利用した Isomap のルーチンを示します．引数 num_neighbs に近傍数を指定します．まず．edge_vec に接続しているデータのインデクスのペア情報と，weight_vec にそのエッジの重み＝データ間の距離を格納します．次に，測地距離によって距離行列を再計算し，最後に多次元尺度構成法によりデータを写像します．

```
bool NStatistics::EmbedIsomapNearest(CMatrix &coords, const CMatrix &dist, size_t dim, size_t num_neighbs)
{
  std::vector> edge_vec;
  std::vector<double> weight_vec;

  // 近傍データの探索
  for (size_t r = 0;r < dist.Row();++r) {
    CVector v = dist.RowVector(r);
    v[r] = DBL_MAX;

    std::pair e;
    e.first = r;
    for (size_t c = 0;c < num_neighbs;++c) {
      size_t mid = v.MinIndex();
      e.second = mid;
      edge_vec.push_back(e);
      weight_vec.push_back(dist(e.first, e.second));
      v[mid] = DBL_MAX;
    }
  }

  // 測地距離による距離行列の再計算
  CMatrix newdist;
  if (!NStatisticsLocal::recreate_dist_matrix(newdist, edge_vec, weight_vec, dist.Row()))
    return false;

  // 古典的MDSによる写像
  return EmbedMDS(coords, newdist, dim);
}
```

測地距離の計算には Boost Graph Library を利用しました. 以下のコードは johnson_all_pairs_shortest_paths 関数の利用例コードをほぼそのまま流用しています.

```
bool recreate_dist_matrix(CMatrix &newdist, const std::vector> &edge_list,
                          std::vector<double> &weight_list, size_t size)
{
  using namespace boost;
  typedef adjacency_listdouble, propertydouble>>> Graph;
  typedef std::pair Edge;

  Edge *edge_array = new Edge[edge_list.size()];
  for (size_t i = 0;i < edge_list.size();++i)
    edge_array[i] = edge_list[i];
  Graph g(edge_array, edge_array + edge_list.size(), size);

  property_map::type w = get(edge_weight, g);
  std::vector<double>::const_iterator cwit = weight_list.begin();

  graph_traits::edge_iterator eit, eit_end;
  for (boost::tie(eit, eit_end) = edges(g);eit != eit_end;++eit)
    w[*eit] = *cwit++;

  double **ndist = new double*[size];
  for (size_t r = 0;r < size;++r)
    ndist[r] = new double[size];
  bool bpath = johnson_all_pairs_shortest_paths(g, ndist);

  newdist.Initialize(size, size);
  for (size_t r = 0;r < size;++r)
  for (size_t c = 0;c < size;++c)
    newdist(r, c) = *(*(ndist + r) + c);

  for (size_t r = 0;r < size;++r)
    delete[] ndist[r];
  delete[] ndist;
  delete[] edge_array;

  if (!bpath)
    newdist.Destroy();
  return bpath;
}
```

Laplacian Eigenmap

Laplacian Eigenmap はグラフラプラシアン行列のスペクトル分解を利用した次元数削減法です.

ラプラシアン行列とは, グラフ構造を行列表現したものです. 隣接エッジに重みが与えられない場合, ラプラシアン行列の対角成分 $L(i,i)$ はノード i に接続するエッジの総数を示し, 非対角成分 $L(i,j)$ にはノード i とノード j が隣接しているならば -1 が与えられます. 一方, 隣接エッジの重みが与えられる場合は, ラプラシアン行列の非対角成分 $L(i,j)$ には, ノード i とノード j が隣接しているならばそのエッジの重み $w(i,j)$ が, 対角成分には接続している全てのエッジの重みの総和 $L(i,i)=\sum_j w(i,j)$ が与えられます.

ラプラシアン行列を固有値分解することで, グラフ構造をスペクトル分解することができます. すなわち, 低次の固有値・固有ベクトルがグラフ構造の大まかな構造(低周波成分？)を表し, より高次の成分がより詳細なグラフ構造(高周波成分？)を表すように, グラフ構造をいくつかの詳細度レベルに分解できます.

今回のシステムではデータの大まかな分布構造を得ることが目的なので, ラプラシアン行列の高次固有ベクトルを打ち切り, 低次固有ベクトルをマッピング先の座標値として計算します. ただし, 最小の固有値は 0 ないし無視できる値となるので, 実際には 2～3 番目に小さな固有値に対応する固有ベクトルを用いて 2 次元マップを生成します.

以下のコードは, 最近傍法による Laplacian Eigenmap ルーチンを示します. 多次元尺度構成法の場合と同様に, 距離行列 dist と写像先の次元数 dim, 近傍数 num_neighbs を指定します. また, 隣接エッジの重みを調整するパラメータ heat を追加しています. $heat \leq 0$ の場合には隣接エッジの重みは全て 1.0 とし, その他の場合には隣接エッジ重みを Heat Kernel: $exp(-dist(i,j)^2/heat)$ で計算します.

```
bool NStatistics::EmbedLENearest(CMatrix &coords, const CMatrix &dist, size_t dim, double heat, size_t num_neighbs)
{
  // 隣接行列
  CMatrix w(dist.Row(), dist.Col(), 0);
  for (size_t r = 0;r < dist.Row();++r) {
    CVector v = dist.RowVector(r);
    v[r] = DBL_MAX;
    for (size_t c = 0;c < num_neighbs;++c) {
      size_t mid = v.MinIndex();
      w(r, mid) = heat <= 0 ? 1.0 : exp(-dist(r, v.MinIndex()) * dist(r, v.MinIndex()) / heat);
      w(mid, r) = w(r, mid);
```

```
12 |       v[mid] = DBL_MAX;
13 !     }
14 !   }
15 |
16 |   // ラプラシアン行列
17 !   CMatrix d(dist.Row(), dist.Col(), 0);
18 |   for (size_t c = 0;c < d.Col();++c)
19 |   for (size_t r = 0;r < d.Row();++r)
20 |     d(c, c) += w(r, c);
21 |   CMatrix l = d - w;
22 |
23 |   // ラプラシアン行列の固有値分解
24 !   CMatrix evecs, tmp;
25 |   CVector evals;
26 | //NStatistics::SVD(evecs, evals, tmp, l);
27 !   NStatistics::EVD(evecs, evals, l);
28 |
29 |   coords.Initialize(dist.Row(), dim);
30 |   for (size_t r = 0;r < coords.Row();++r)
31 |   for (size_t c = 0;c < coords.Col();++c)
32 |     coords(r, c) = evecs(r, evecs.Col() - c - 2);
33 |   return true;
34 ! }
```

Locally Linear Embedding

Locally Linear Embedding は Isomap と同時掲載された非線形次元数削減アルゴリズムです(LLE本家ホームページ). アルゴリズムは本家の擬似コードに詳しいですが, 大雑把に次の 3 ステップで表されます.

1. サンプルデータ s_i に隣接する K 個の近傍データ s_k を探索.
2. サンプル s_i を, 近傍データの加重和 $s_i \approx \sum w_k s_k$ によって近似するための最良な加重係数 w_k を求め, 加重係数行列 W の第 i 行ベクトルに $W(i.k) = w_k$ のように格納.
3. 行列 $M = (I-W)^T(I-W)$ を固有値分解し, 2 番目と 3 番目に小さな固有値に対応する 2 つの固有値ベクトルを用いて写像先の座標を決定.

本家ホームページに図示されているように, Locally Linear Embedding では, 元のデータ空間における隣接関係を加重係数によって表現し, 写像先の空間においてもその加重係数の関係が保たれるようにマッピングするアルゴリズムです(いまいち理解できていませんが...)

公開されている擬似コードに沿って実装した Locally Linear Embedding ルーチンを以下に示します. データ行列 data と 距離行列 dist, 写像先の次元数 dim, 近傍数 num_neighbs を指定します. ただしこの実装コードは, かなり怪しいマッピング結果を与えるので, バグを含む可能性が非常に高いです. 擬似コードをそのまま実装したつもりですし, 加重係数も正しく計算できていそうなのですが...はたまたこの写像結果で正しいのか, うーん.... 何かお気づきの方はぜひ教えてください.

```
 1   bool NStatistics::EmbedLLE(CMatrix &coords, const CMatrix &data, const CMatrix &dist, size_t dim, size_t num_neighbs)
 2 | {
 3 |   CMatrix mw(dist.Row(), dist.Col(), 0);
 4 |   for (size_t i = 0;i < data.Row();++i) {
 5 |     // 隣接行列
 6 !     CMatrix nm(num_neighbs, data.Col());
 7 |     CVector vd = dist.RowVector(i);
 8 |     vd[i] = DBL_MAX;
 9 |     for (size_t r = 0;r < num_neighbs;++r) {
10 |       size_t mid = vd.MinIndex();
11 |       for (size_t c = 0;c < data.Col();++c)
12 |         nm(r, c) = data(mid, c) - data(i, c);
13 |       vd[mid] = DBL_MAX;
14 !     }
15 |
16 |     // 局所的共分散行列
17 !     CMatrix lc = nm * nm.Transpose();
18 |
19 |     // 加重ベクトル
20 !     CVector v1(lc.Row(), 1.0), vw(lc.Row(), 0);
21 |     gsl_permutation *p = gsl_permutation_alloc(lc.Row());
22 |     int sign;
23 |     gsl_linalg_LU_decomp(lc.m_pmMatrix, p, &sign);
24 |     gsl_linalg_LU_solve(lc.m_pmMatrix, p, v1.m_pvVector, vw.m_pvVector);
25 |     gsl_permutation_free(p);
26 |
27 |     double wsum = 0;
28 |     for (size_t r = 0;r < vw.Size();++r)
29 |       wsum += vw[r];
30 |     vd = dist.RowVector(i);
31 |     vd[i] = DBL_MAX;
32 |     for (size_t r = 0;r < vw.Size();++r) {
33 |       size_t mid = vd.MinIndex();
34 |       mw(i, mid) = vw[r] / wsum;
35 |       vd[mid] = DBL_MAX;
```

```
36 !      }
37 !    }
38 |
39 |    CMatrix sm = -mw;
40 |    for (size_t i = 0;i < sm.Row();++i)
41 |      sm(i, i) += 1.0;
42 |    sm = sm.Transpose() * sm;
43 |
44 |    CMatrix evecs, tmp;
45 |    CVector evals;
46 |  //NStatistics::SVD(evecs, evals, tmp, sm);
47 !    NStatistics::EVD(evecs, evals, sm);
48 |
49 |    coords.Initialize(dist.Row(), dim);
50 |    for (size_t r = 0;r < coords.Row();++r)
51 |    for (size_t c = 0;c < coords.Col();++c)
52 |      coords(r, c) = evecs(r, evecs.Col() - c - 2);
53 |    return true;
54 ! }
```

モーションデータのマッピング [†]

マッピング対象となるデータは，ルートの位置と回転成分を除く，全ての関節の回転クォータニオンの時系列を与えています．また，多次元尺度構成法などで用いる距離行列は，姿勢距離測度によって算出しています．データ行列と距離行列の計算ルーチンを以下に示します．

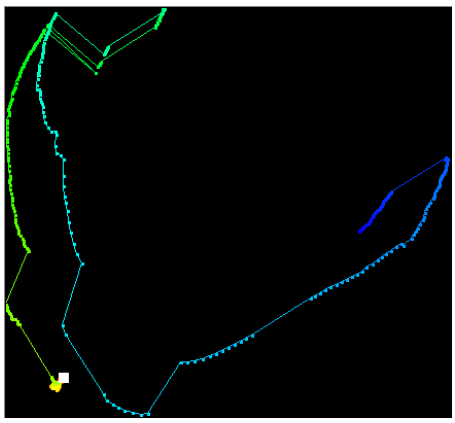
```
 1    CMatrix create_motion_matrix(const CMotionData &mot)
 2  {
 3  |   CMatrix motion(mot.NumFrames(), mot.NumJoints() * 4 - 4);
 4  |   for (size_t frm = 0;frm < mot.NumFrames();++frm)
 5  |   for (size_t j = 0;j < mot.NumJoints() - 1;++j) {
 6  |     D3DXQUATERNION q = mot.GetRotation(frm, j + 1);
 7  |     motion(frm, j * 4 + 0) = q.x;
 8  |     motion(frm, j * 4 + 1) = q.y;
 9  |     motion(frm, j * 4 + 2) = q.z;
10  |     motion(frm, j * 4 + 3) = q.w;
11  !   }
12  |   return motion;
13  ! }
14
15    CMatrix create_distance_matrix(CFigure &fig, const CMotionData &mot)
16  {
17  |   CMatrix distance(mot.NumFrames(), mot.NumFrames(), 0);
18  |   for (size_t frm1 = 0;frm1 < mot.NumFrames();++frm1)
19  |   for (size_t frm2 = frm1 + 1;frm2 < mot.NumFrames();++frm2) {
20  |     distance(frm1, frm2) = NFigureUtil::PoseDistance(fig, mot, frm1, frm2);
21  |     distance(frm2, frm1) = distance(frm1, frm2);
22  !   }
23  |   return distance;
24  ! }
```

評価実験 [†]

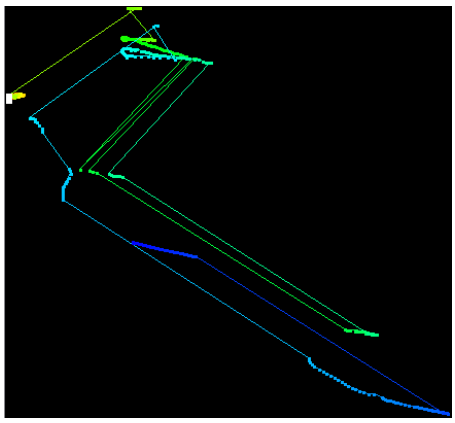
実装した各種アルゴリズムのマッピング性能を比較するために，CMUモーションキャプチャデータベースの Subject 64 に含まれるゴルフスイング動作データを 2 次元マッピングしました．このアニメーションムービーを以下に示します．

ゴルフスイング動作(m1v, 157KB)

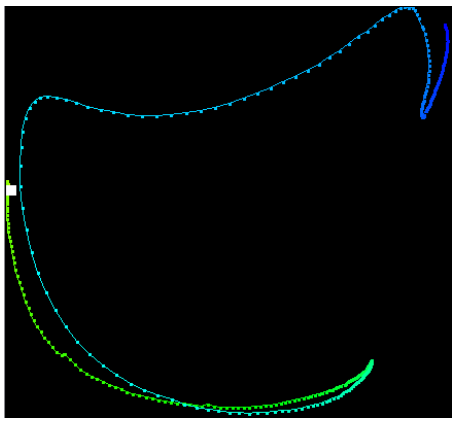
生成された 2 次元マップを以下に示します．軌跡の赤色から青色への変化が時間経過に対応します．ただ，スイング開始までの静止時間が長いので，赤～黄色はほとんど見えないと思います．計算パラメータはかなり適当に，Isomap, LE, LLE の近傍数を 20, LE の heat パラメータは 2.0 に設定しています．



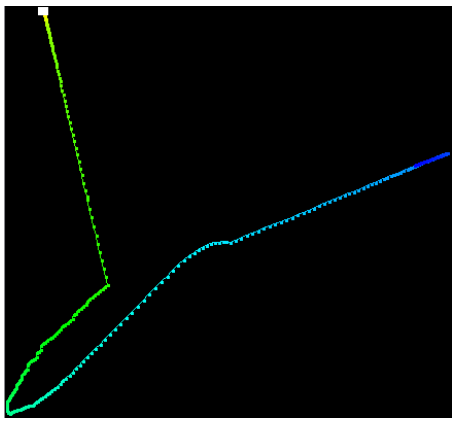
(1) PCA



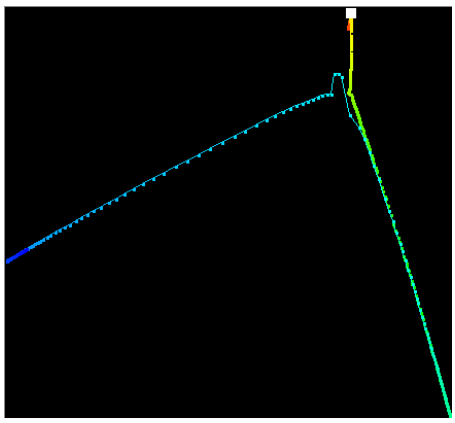
(2) SVD



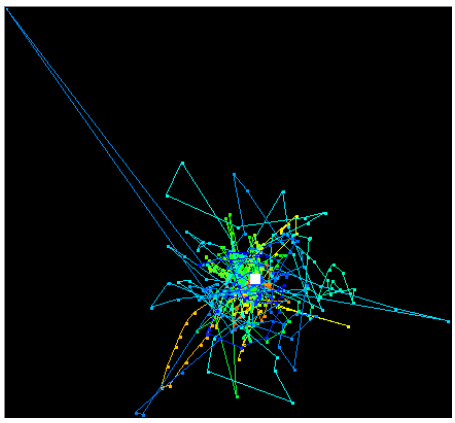
(3) MDS



(4) Isomap



(5) LE



(6) LLE

1. MDS
    - 初期姿勢→テイクバックと，テイクバック→インパクトの軌跡が近くに描かれつつ，微妙な差異を示しています．2 次元上に滑らかにマッピングされており，他手法と比較的すると直観的に動作の特徴を理解できます．
2. PCA
    - 概形は MDS と近いのですが，軌跡のガタつきやジャンプが見られます．激しい動作をマッピングすると，この傾向がより顕著に現れるようです．
3. SVD
    - 処理手順における PCA との差異はデータ行列の標準化の有無だけですが，マッピング結果は大きく異なることが確認できます．マッピングの均一性でみると，SVD は PCA や MDS にやや劣るようです．
4. Isomap, LE
    - 時刻が近いほど姿勢も類似しているという，モーションデータの時間相関が反映されたマッピングが行われています．つまり，見た目の姿勢は非常に近いにも関わらず，時間的な離間がマップ上の距離として現れています．ただ，近傍数などのパラメータをうまく調整すれば，MDS よりも優れたマッピングを与えるかもしれません．
5. LLE
    - うーん，やはり実装ミスでしょうね...

マップ上のスケッチによる動作合成 †

マップ上の任意の位置をマウスで指定すると，周辺のサンプル姿勢の加重和によって新しい姿勢を合成します．基本的には，マウスクリック位置と各サンプルの座標間の距離に応じて，線形補間によって各サンプル姿勢への重みを計算しています．ただし，重みの総和が 1.0 になる必要があるので，ラグランジュの未定乗数法を用いて重み総和に関する拘束条件を導入しています．

```
 1  void CManifoldView::ShowMap(const CMatrix &coords)
 2  {
 3    .........
 4
 5    if (m_pCov != NULL)
 6      delete m_pCov;
 7    m_pCov = new CMatrix(m_pCoords->Row() + 1, m_pCoords->Row() + 1, 1.0);
 8    for (size_t r = 0;r < m_pCov->Row() - 1;++r)
 9    for (size_t c = r;c < m_pCov->Col() - 1;++c) {
10      m_pCov->Data(r, c) = (m_pCoords->RowVector(r) - m_pCoords->RowVector(c)).Norm();
11      m_pCov->Data(c, r) = m_pCov->Data(r, c);
12    }
13    m_pCov->Data(m_pCov->Row() - 1, m_pCov->Col() - 1) = 0.0;
14    *m_pCov = m_pCov->InverseLUD();
15
16    SendMessage(WM_USER_RENDER_MAP);
17  }
18
19  void CManifoldView::OnMouseMove(UINT nFlags, CPoint point)
20  {
21    if (nFlags != MK_LBUTTON || m_pCoords == NULL)
22      return;
23
24    RECT rect;
25    GetClientRect(&rect);
26
27    CVector p(2);
28    p[0] = static_cast<double>(point.x) / static_cast<double>(rect.right - rect.left);
29    p[1] = static_cast<double>(point.y) / static_cast<double>(rect.bottom - rect.top);
30
31    CVector cv(m_pCoords->Row() + 1, 1.0);
32    for (size_t i = 0;i < m_pCoords->Row();++i)
33      cv[i] = (m_pCoords->RowVector(i) - p).Norm();
34
35    CVector w = *m_pCov * cv;
36    SendMessage(WM_USER_RENDER_MAP, point.x, point.y);
37    ((CSceneView *)GetDocument()->GetSceneView())->BlendPose(w);
38
39    CView::OnMouseMove(nFlags, point);
40  }
```

まとめ †

部分空間法や多様体学習法を用いて，キャラクタモーションデータを 2 次元平面上にマッピングするシステムを実装しました．個々のアルゴリズムは詳しく説明しなかったので，興味のある方は原著論文や Web 上の資料を参照してください．スケッチングシステムもやっつけ仕事で実装したので，サンプルデータによっては計算が破綻すると思いますが，その点はご容赦ください．

本来は，Gaussian Process Latent Variable Models(GPLVM)との比較も行いたかったのですが，公開されているコードをそのまま使うだけの記事になりそうなので，今回はパスしました．なかなか面白い結果が得られるので，興味がある方はぜひお試しください(ただし，GPLVM は最適化計算を伴うのでかなりの計算時間が必要です)．
