Monitoring Patient Motion Using Kinect Sensors to Optimize Spinal Cord Injury Therapy

Chiraag Nataraj

Mary P. and Dean C. Daily SURF Fellow

Mentor: Dr. Joel Burdick

The Burdick Group

09/29/14

Contents

1 Acknowledgments 1

2 Introduction 1

3 Background 2

4 First Attempt 2
4.1 Tree Classifiers 2
4.2 Method 3

5 Novel Method 3

6 Results 4

7 Challenges 5

8 Further Work 5

A Code 5

Abstract

The motivation of this research was to utilize depth sensor technology to track human body motion. Currently, many algorithms exist to do so — one of the most popular ones is SCAPE, which has two stages. The first stage is the initialization phase, whereby the generic human model is modified to resemble the actual human being tracked. The second stage is the tracking phase, whereby the generated model is then used along with chosen keypoints to track the motion of the person. The key contribution of this research is in the first phase. Normally, the initialization is done by hand by selecting several keypoints on both the captured image and the model. This research seeks to automate that process using a combination of edge detection and point-to-point correspondence given that the initial pose is roughly known.

1 Acknowledgments

I would like to thank the following people and organizations.

- **Dr. Joel Burdick** for being an excellent mentor
- **Caltech** for this wonderful opportunity
- **Mr. and Mrs. Daily** for generously donating funds for my SURF
- **Caltech** for financial support

2 Introduction

Spinal cord injury (SCI) affects nearly 1.2 million people in the US alone. Repercussions include:

- loss of bowel/bladder movement
- blood pressure anomalies
- bone and muscle loss
- problems with temperature regulation

It is unfortunate that 61% of these people acquire the injury between the ages of 16 and 30. In the future, scientists are looking at new ways of helping patients recover such as:

- stem cells
- neural tissue transplants
- growth factors
- genetic manipulation

Current therapy includes:

- physical therapy
- management of the repercussions
- occupational therapy

Epidural Spinal Stimulation (ESS) is an approach currently under investigation for SCI recovery. Physical exercise during the stimulation process is a critical part of the therapy.

While patients exercise in clinic, there is no problem, as there are always many clinicians around to assist the patient with training and safety. It is known that the therapy can help with recovery even after the patient leaves the clinic. There currently exists a machine to help patients continue recovery away from the clinic. However, currently, there is no way to effectively monitor the patients while they rehabilitate outside

the clinic. The goal of this project, therefore, is to develop a human tracking technology, based on low-cost RGB-D technology, which can monitor the patient's exercises while at home.

3 Background

Because human tracking is useful for so many applications, there are several possible tracking algorithms already available for use. One of the most prominent ones is an algorithm called SCAPE (see Figure 1), which has two distinct phases. The first is the initialization phase, which (normally) relies on a human to manually select correspondences between a captured image and a standard reference model. In effect, the standard model is deformed using this data to fit a specific human. This results in a set of transformation and deformation matrices which defines the SCAPE model. The second phase is the tracking phase, which tracks changes in the joints over time based on the model and RGB-D data.

The second phase is reasonably efficient once the first phase has taken place. However, the first phase is tedious and prone to human error. If there were some way to automate the process, it would be far more efficient and possibly less prone to error. This is the premise and goal of this work.

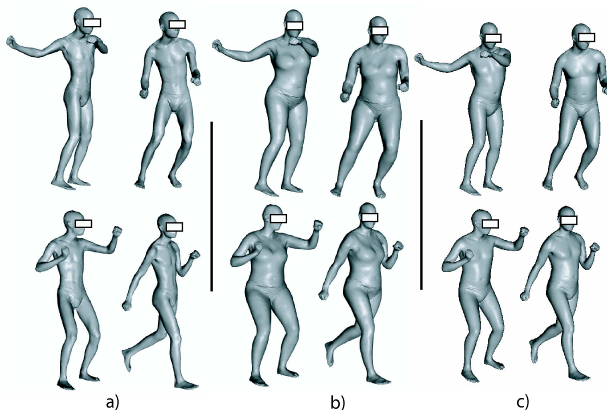

Figure 1: Deformation transfer by the SCAPE model. [1]

4 First Attempt

4.1 Tree Classifiers

Before delving into the specifics of the method, it is necessary to have a simple introduction to tree classifiers. First, a tree is a generic data structure. Each element is called a node — terminal nodes are nodes with no sub-nodes. A tree consists of a "root node", with edges connecting the root to its children nodes. A tree classifier is a tree which takes data as input (in this case, a set of points) and categorizes each element of the data. Each "split" in the classifier categorizes the input data to that node into, in this case, two distinct groups (the two branches of the node). The criterion which decides how the input data is categorized at each non-terminal node is called the "question". Terminal nodes are given a classification which identifies the data which is categorized into that node, thus fulfilling the goal of the classifier.

Every tree classifier has two main characteristics. The first is the set of so-called "questions" which determines how the input data is classified. The second is the stopping rule, which dictates when a node is not split any further. The classifier is trained by selecting the questions that minimize or maximize a given criterion. In this specific scenario, the input data is a set of points. Additionally, the body is segmented into several sections, with each section corresponding roughly to a body part (upper thigh, foot, upper arm, hand, and

so on). Therefore, in this case, the criterion was the amount of information regarding the categorization of the point gained about the data. Naturally, if the classifier is to be any good, the set of possible questions must be quite large to allow the classifier to select the *best* one at each node of the tree.

4.2 Method

First, investigation was done of the available methods which could be used for initialization. All candidate methods needed to be relatively efficient as well as relatively accurate. In our investigation, this led to a fairly small collection of tree classifiers.

Tree classifiers have two main disadvantages. They require a tremendous amount of training data and the training is extremely computationally expensive. Despite this, however, I attempted to implement a specific type of classifier implemented previously in [9].

The question then arises as to how to select the set of possible questions. Following [9], a "feature" was defined as in Equation (1) and was used in the formulation of the question.

$$f\left(\vec{x}, \vec{u}, \vec{v}\right) = d\left(\vec{x} + \frac{\vec{u}}{d\left(\vec{x}\right)}\right) - d\left(\vec{x} + \frac{\vec{v}}{d\left(\vec{x}\right)}\right) \tag{1}$$

$\vec{x}$ is the vector from the origin to the point at which the feature is being calculated. $\vec{u}$ and $\vec{v}$ are offset vectors which define the feature. $d\left(\vec{x}\right)$ is a function which calculates the depth at a given point. If the point is a point on the point cloud, its depth is returned. Otherwise, the depth of the closest point is returned.

The next question, however, was how to select the set of features to optimize over. This was performed by taking the bounding box formed by the body and discretizing it. Too fine of a discretization led to inordinately large computation times, whereas too coarse of a discretization led to a badly performing tree classifier.

On top of this, however, there was the issue of training data and overall training time. Most training takes on the order of at least hundreds of thousands of training data sets and at least a day. Not having access to a supercomputer, it would have taken several weeks just to see if the tree worked, even if we had had access to many data sets. After working on implementing this method for several weeks and attempting to make many adjustments to find out if using few data sets and carefully designed features would work, we decided to abandon this method.

5 Novel Method

We designed a novel method to find and match corresponding points on a captured image and a reference model. The method is a modified gradient descent method designed to compute the transformations necessary to align the captured point cloud with the reference model. As of this writing, full implementation has not been completed. Preliminary efforts show great promise, but more work is needed to confirm the viability of this strategy.

There are two steps in this method. The first is to obtain a correspondence between points on the captured point cloud and points on the model. This is fairly straightforward if the position is roughly known and the captured point cloud and the model point cloud contain the same number of points. The second step is to utilize a gradient descent algorithm to modify the joint and shape parameters of the model point cloud to more closely coincide with the captured point cloud. The method resembles an iterated ICP method in the sense that there are two phases to each iteration of the second step. The first phase consists of modifying the model to fit the data. The second phase consists of updating the model parameters to reflect the changed orientation.

In order to utilize the gradient descent algorithm, it is necessary to have both a function which will be minimized as well as a method to calculate the gradient of that function. The function chosen is of the form

of Equation (2), where $\bar{p}_i$ represents the position of the measured point, $m_i(g)$ represents the position of the corresponding point on the theoretical model, and g is the transformation that more closely aligns the model points with the measured points.

$$E(g) = \sum_{i=1}^{M} || \bar{p}_i - m_i(g) || \tag{2}$$

6 Results

Currently, only a simplified version of the first step has been implemented. In this current implementation, the task is to correctly identify corresponding points in two different point clouds. Both contain identical numbers of points and one point cloud is shifted by 0.5 units to the right. These results are summarized in Table 1 and Figure 2.

The code attempts to protect against issues arising when the two point clouds overlap by calculating the normal vector at each point and selecting the matching point from the set of points with a parallel normal vector. This leads to the great results seen in Table 1.

There are, however, several problems with this version. This code expects that both point clouds will contain the same number of points, which is not necessarily true. Furthermore, it expects no noise, which is completely unreasonable in the context of the larger picture. That being said, this code will be improved upon to take those factors into consideration while still yielding reasonable results.

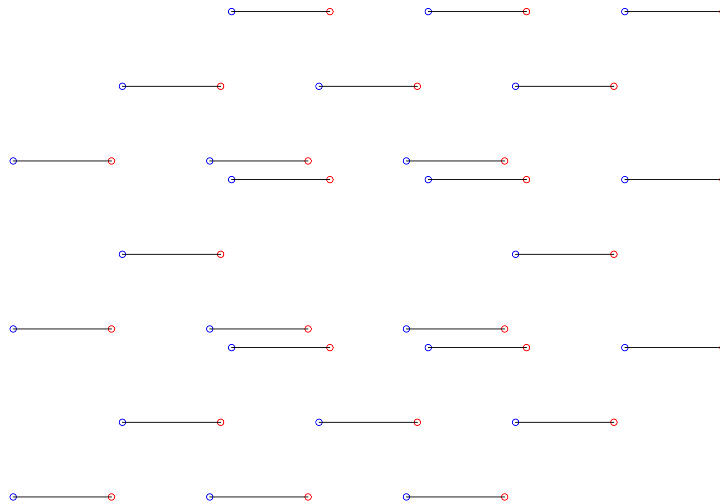

Figure 2: Point matching is performed between original point cloud (blue) and shifted point cloud (red)

Translated Point	Real Original Point	Matched Original Point
(1.5,-1,-1)	(1,-1,-1)	✓
(-0.5,1,-1)	(-1,1,-1)	✓

(-0.5,-1,1)	(-1,-1,1)	✓
(-0.5,-1,-1)	(-1,-1,-1)	✓
(1.5,-1,0)	(1,-1,0)	✓
(-0.5,1,0)	(-1,1,0)	✓
(-0.5,0,1)	(-1,0,1)	✓
(-0.5,-1,0)	(-1,-1,0)	✓
(-0.5,0,-1)	(-1,0,-1)	✓
(1.5,-1,1)	(1,-1,1)	✓
(-0.5,1,1)	(-1,1,1)	✓
(1.5,0,-1)	(1,0,-1)	✓
(0.5,1,-1)	(0,1,-1)	✓
(0.5,-1,1)	(0,-1,1)	✓
(0.5,-1,-1)	(0,-1,-1)	✓
(1.5,0,0)	(1,0,0)	✓
(0.5,1,0)	(0,1,0)	✓
(0.5,0,1)	(0,0,1)	✓
(-0.5,0,0)	(-1,0,0)	✓
(0.5,-1,0)	(0,-1,0)	✓
(0.5,0,-1)	(0,0,-1)	✓
(1.5,0,1)	(1,0,1)	✓
(0.5,1,1)	(0,1,1)	✓
(1.5,1,-1)	(1,1,-1)	✓
(1.5,1,0)	(1,1,0)	✓
(1.5,1,1)	(1,1,1)	✓

Table 1: Matching Points algorithm

7 Challenges

The main challenges associated with this project have had to do with the sheer amount of data and computation required by the tree-based classifier approach. As mentioned earlier, efforts to pare down the amount of data needed did not work as expected and led to trees that did not operate properly.

8 Further Work

The method described in this paper will be implemented and tested and will hopefully yield results. Preliminary testing indicates that the point-to-point correspondence step is somewhat easily achievable given that both point clouds have the same number of points as shown in the results section. Much work remains to be done to fully implement the algorithm.

A Code

Listing 1: CalculateNormals.cpp

```
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/features/normal_3d_omp.h>
#include <cmath>
#include <iostream>
```

```
bool parallel(pcl::Normal &u, pcl::Normal &v)
{
  float x1 = u.normal_x;
  float y1 = u.normal_y;
  float z1 = u.normal_z;
  float x2 = v.normal_x;
  float y2 = v.normal_y;
  float z2 = v.normal_z;

  float dotProduct = x1 * x2 + y1 * y2 + z1 * z2;
  float magU = sqrt(pow(x1, 2) + pow(y1, 2) + pow(z1, 2));
  float magV = sqrt(pow(x2, 2) + pow(y2, 2) + pow(z2, 2));

  float cosine = dotProduct / (magU * magV);

  if(cosine > 0.9999)
    {
      return true;
    }
  else
    {
      return false;
    }
}

void calculateNormals(pcl::PointCloud<pcl::PointXYZ>::Ptr &cloud,
                      pcl::PointCloud<pcl::Normal>::Ptr &normals)
{
  pcl::NormalEstimationOMP<pcl::PointXYZ, pcl::Normal> NE;
  NE.setInputCloud(cloud);
  pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search::KdTree<pcl::
     PointXYZ>());
  NE.setSearchMethod(tree);
  NE.setRadiusSearch(2);
  NE.compute(*normals);
}

void matchPoints(pcl::PointCloud<pcl::PointXYZ>::Ptr &model,
                 pcl::PointCloud<pcl::PointXYZ>::Ptr &real,
                 pcl::PointCloud<pcl::Normal>::Ptr &modelNormals,
                 pcl::PointCloud<pcl::Normal>::Ptr &realNormals,
                 std::vector<std::tuple<pcl::PointXYZ, pcl::PointXYZ>> &mapping)
{
  pcl::KdTreeFLANN<pcl::PointXYZ> kdtree;
  int K = 1;
  std::vector<int> indices;
  std::vector<float> distances;
  pcl::Normal prevNorm;
  pcl::Normal curNorm;
  pcl::Normal tmpNorm;

  for(size_t i = 0; i < real->points.size(); i++)
    {
      curNorm = realNormals->points[i];

      int k = 0;

      pcl::PointCloud<pcl::PointXYZ> reducedCloud;
      reducedCloud.resize(model->points.size());
```

```
      for(size_t j = 0; j < modelNormals->points.size(); j++)
        {
          tmpNorm = modelNormals->points[j];
          if(parallel(curNorm, tmpNorm))
            {
              float a = (*model).points[j].x;
              float b = (*model).points[j].y;
              float c = (*model).points[j].z;
              reducedCloud.points[k].x = a;
              reducedCloud.points[k].y = b;
              reducedCloud.points[k].z = c;
              k += 1;
            }
        }
      reducedCloud.resize(k);

      pcl::PointCloud<pcl::PointXYZ>::Ptr reducedCloudPtr(new pcl::PointCloud<pcl
         ::PointXYZ>(reducedCloud));

      kdtree.setInputCloud(reducedCloudPtr);
      kdtree.nearestKSearch((*real).points[i], K, indices, distances);
      std::tuple<pcl::PointXYZ, pcl::PointXYZ> tmp(real->points[i],
         reducedCloudPtr->points[indices[0]]);
      mapping.push_back(tmp);
      prevNorm = curNorm;
    }
}

void construct_hat_matrix(std::vector <float> &vec,
                          std::vector <std::vector <float> > &res)
{
  res.clear();
  std::vector <float> tmp;
  tmp.push_back(0);
  tmp.push_back(-vec[2]);
  tmp.push_back(vec[1]);
  res.push_back(tmp);
  tmp.clear();
  tmp.push_back(vec[2]);
  tmp.push_back(0);
  tmp.push_back(-vec[0]);
  res.push_back(tmp);
  tmp.clear();
  tmp.push_back(-vec[1]);
  tmp.push_back(vec[0]);
  tmp.push_back(0);
  res.push_back(tmp);
}

void matrix_vector_mult(std::vector <std::vector <float> > &R,
                        pcl::PointXYZ &p, std::vector <float> &res)
{
  res.clear();
  for(int i = 0; i < R.size(); i++)
    {
      float r = 0;
      for(int j = 0; j < R[i].size(); j++)
        {
```

```
          if(j == 0)
            {
              r += R[i][j] * p.x;
            }
          else if(j == 1)
            {
              r += R[i][j] * p.y;
            }
          else
            {
              r += R[i][j] * p.z;
            }
        }
      res.push_back(r);
    }
}

void initialize_R_matrix(std::vector <std::vector <float> > &R)
{
  R.clear();
  std::vector <float> tmp;
  tmp.push_back(0);
  tmp.push_back(0);
  tmp.push_back(0);
  R.push_back(tmp);
  tmp.clear();
  tmp.push_back(0);
  tmp.push_back(0);
  tmp.push_back(0);
  R.push_back(tmp);
  tmp.clear();
  tmp.push_back(0);
  tmp.push_back(0);
  tmp.push_back(0);
  R.push_back(tmp);
}

void calculate_dmdg(std::vector <std::vector <float> > &dmdg,
                    pcl::PointXYZ &p,
                    std::vector <std::vector <float> > &R)
{
  std::vector <float> tmp, rotated;
  std::vector <std::vector <float> > Rzi;
  matrix_vector_mult(R, p, rotated);
  construct_hat_matrix(rotated, Rzi);
  tmp.push_back(1);
  tmp.push_back(0);
  tmp.push_back(0);
  tmp.push_back(Rzi[0][0]);
  tmp.push_back(Rzi[0][1]);
  tmp.push_back(Rzi[0][2]);
  dmdg.push_back(tmp);
  tmp.clear();
  tmp.push_back(0);
  tmp.push_back(1);
  tmp.push_back(0);
  tmp.push_back(Rzi[1][0]);
  tmp.push_back(Rzi[1][1]);
  tmp.push_back(Rzi[1][2]);
```

```
  dmdg.push_back(tmp);
  tmp.clear();
  tmp.push_back(0);
  tmp.push_back(0);
  tmp.push_back(1);
  tmp.push_back(Rzi[2][0]);
  tmp.push_back(Rzi[2][1]);
  tmp.push_back(Rzi[2][2]);
  dmdg.push_back(tmp);
}

void transpose(std::vector <std::vector <float> > &cis,
               std::vector < std::vector <float> > &trans)
{
  for(int i = 0; i < cis[0].size(); i++)
    {
      std::vector <float> tmp;
      for(int j = 0; j < cis.size(); j++)
        {
          tmp.push_back(cis[j][i]);
        }
      trans.push_back(tmp);
    }
}

void scalar_mult(std::vector <std::vector <float> > &mat,
                 float mult)
{
  for(int i = 0; i < mat.size(); i++)
    {
      for(int j = 0; j < mat[i].size(); j++)
        {
          mat[i][j] *= mult;
        }
    }
}

void vector_subtract(std::vector <float> &a,
                     std::vector <float> &b,
                     std::vector <float> &res)
{
  for(int i = 0; i < a.size(); i++)
    {
      res.push_back(a[i] - b[i]);
    }
}

void vector_self_add(std::vector <float> &orig,
                     std::vector <float> &add)
{
  if(orig.size() > 0)
    {
      for(int i = 0; i < orig.size(); i++)
        {
          orig[i] += add[i];
        }
    }
  else
    {
```

```
      for(int i = 0; i < add.size(); i++)
        {
          orig.push_back(add[i]);
        }
    }
}

void error_gradient(std::vector <std::tuple<pcl::PointXYZ, pcl::PointXYZ> > &
    mapping,
                    std::vector <float> &g,
                    std::vector<float> &sum)
{
  sum.clear();
  for(int i = 0; i < mapping.size(); i++)
    {
      pcl::PointXYZ m = std::get<0>(mapping[i]);
      pcl::PointXYZ p = std::get<1>(mapping[i]);
      std::vector <float> sum_1, sum_2, sum_3;
      std::vector <std::vector <float> > dmdg, R, dmdg_transposed;
      initialize_R_matrix(R);
      calculate_dmdg(dmdg, m, R);
      transpose(dmdg, dmdg_transposed);
      matrix_vector_mult(dmdg_transposed, p, sum_1);
      matrix_vector_mult(dmdg_transposed, m, sum_2);
      vector_subtract(sum_1, sum_2, sum_3);
      vector_self_add(sum, sum_3);
    }
}

int main()
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cisCloud(new pcl::PointCloud<pcl::PointXYZ>)
      ;
  pcl::PointCloud<pcl::PointXYZ>::Ptr transCloud(new pcl::PointCloud<pcl::PointXYZ
      >);

  pcl::io::loadPCDFile<pcl::PointXYZ> ("/home/chiraag/Documents/Caltech/SURF_2014/
      Code/Python/cube-example.pcd", *cisCloud);
  pcl::io::loadPCDFile<pcl::PointXYZ> ("/home/chiraag/Documents/Caltech/SURF_2014/
      Code/Python/cube-example-translated-0.5.pcd", *transCloud);

  pcl::PointCloud<pcl::Normal>::Ptr cisCloudNormals (new pcl::PointCloud<pcl::
      Normal>);
  pcl::PointCloud<pcl::Normal>::Ptr transCloudNormals (new pcl::PointCloud<pcl::
      Normal>);

  std::vector<std::tuple<pcl::PointXYZ, pcl::PointXYZ>> mapping;

  std::cout << "Calculating normals..." << std::endl;

  calculateNormals(cisCloud, cisCloudNormals);
  calculateNormals(transCloud, transCloudNormals);

  std::cout << "Calculated normals!" << std::endl;

  std::cout << "Matching points..." << std::endl;

  matchPoints(cisCloud, transCloud, cisCloudNormals, transCloudNormals, mapping);
```

```
  std::cout << "Matched points!" << std::endl;

  for(int i = 0; i < mapping.size(); i++)
    {
      pcl::PointXYZ a = std::get<0>(mapping[i]);
      pcl::PointXYZ b = std::get<1>(mapping[i]);
      std::cout << "(" << a.x << "," << a.y << "," << a.z << ") -> (" << b.x << ",
          " << b.y << "," << b.z << ")" << std::endl;
    }

  // std::vector<float> g;
  // std::vector<float> gradient;
  // g.push_back(-1.5);
  // g.push_back(0);
  // g.push_back(0);

  // error_gradient(mapping, g, gradient);

  // std::cout << gradient[0] << " " << gradient[1] << " " << gradient[2] << std::
     endl;

  return 0;
}
```

References

[1] Dragomir Anguelov et al. "SCAPE: shape completion and animation of people". In: *ACM Trans. Graph* 24 (2005), pp. 408–416.

[2] Adam Baumberg. "Reliable feature matching across widely separated views". In: *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on.* 2000.

[3] Leo Breiman et al. *Classification And Regression Trees.* Chapman & Hall, 1993.

[4] Ross Girshick et al. "Efficient Regression of General-Activity Human Poses from Depth Images". In: *IEEE* (2011).

[5] Bastian Leibe, Ales Leonardis, and Bern Schiele. "Combined Object Categorization and Segmentation with an Implicit Shape Model". In: *ECCV'04 Workshop on Statistical Learning in Computer Vision.* May 2004.

[6] Vincent Lepetit and Pascal Fua. *Towards Recognizing Feature Points using Classification Trees.* Tech. rep. EPFL, 2004.

[7] Vincent Lepetit, Pascal Lagger, and Pascal Fua. "Randomized Trees for Real-Time Keypoint Recognition". In: *Proc. CVPR.* IEEE, 2005.

[8] Vincent Lepetit, Julien Pilet, and Pascal Fua. "Point Matching as a Classification Problem for Fast and Robust Object Pose Estimation". In: *Proc. CVPR.* IEEE, 2004.

[9] Jamie Shotton et al. "Real-Time Human Pose Recognition in Parts from Single Depth Images". In: *Communications of the ACM* (2013).

[10] Jonathan Taylor et al. "The Vitruvian Manifold: Inferring Dense Correspondences for One-Shot Human Pose Estimation". In: *Proc. CVPR.* IEEE, 2012.

[11] Alexander Weiss, David Hirshberg, and Michael J. Black. "Home 3D Body Scans from Noisy Image and Range Data". In: *IEEE International Conference on Computer Vision.* 2011.