

n -Dimensional Linear Vector Field Regression with NumPy

GUY K. KLOSS AND TIM F. KLOSS*

*Computer Science
Institute of Information
& Mathematical Sciences
Massey University at Albany*

** SKB Stadtfernsehen Brandenburg
Numeric Research Division
Brandenburg an der Havel
Tim.Kloss@arcor.de*

and
*School of Computing
+ Mathematical Sciences
Auckland University of Technology,
Auckland, New Zealand
G.Kloss@massey.ac.nz
Guy.Kloss@aut.ac.nz*

This manuscript describes an implementation of regression calculations for affine transformations in (virtually) arbitrary dimensionality from two point sets in \mathbb{R}^n as best as possible in the least squares sense. The implementation uses NumPy and is very fast, also for larger point sets and dimensionalities. Such transformations are often required for image registration, compensation for (simple) distortions, first (linear) approximations, etc.

Keywords: Linear regression; vector field; affine transformation; NumPy.

1 Overview

In sciences certain mathematical problems arise in various scenarios. One of these is the computation of linear regressions. In one dimensional space this is simple enough, for higher dimensionalities it becomes more of a challenge. Regressions on vector fields cannot be easily found and implemented in a straight forward way, even though the problem itself is not very difficult. Such cases arise for example if one needs to register images (two dimensions, \mathbb{R}^2) onto each other. This could be in the case areal photographs (for good approximations only with small angles of view) need to be registered to maps. Samples for higher dimensional cases are approximations for local flow fields (three dimensions, \mathbb{R}^3). A camera takes two successive images of illuminated tracer particles in a fluid. If the positions of tracer particles can be paired in the two images, the flow state can be approximated for small enough regions containing a number of tracers. In our particular case this implementation has been used for the linear approximation of a border line problem in a smoothed linear spline fitting to a three or four dimensional vector field.

This paper derives mathematically an extension of an affine transformation to an n -dimensional linear regression computation using a least squares approach. Rather than implementing a solver ourselves, we are using indirectly the excellent solver from LAPACK¹, which is wrapped in NumPy's [1, 2] `numpy.linalg` package.

¹<http://www.netlib.org/lapack>

2 Description

The algorithm is mathematically derived from a simple least squares based linear regression [3] towards higher dimensionalities. A review on the web resulted in an implementation in pure Python by Jarno Elonen² implementing an algorithm as described by Helmut Späth [4] for similar problems. This implementation in contrast follows directly the principles of the commonly known linear regression calculation.

We are assuming two point sets P and Q of m points describing the source and destination of the affine transformation given by

$$\mathbf{p}_i = (p_{1i}, \dots, p_{ni})^T, \quad \mathbf{q}_i = (q_{1i}, \dots, q_{ni})^T \quad (i = 1, \dots, m). \quad (1)$$

We now want to find some matrix \mathbf{A} and translation vector \mathbf{b}

$$\mathbf{A} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \quad (2)$$

$$\mathbf{b} = (b_1, \dots, b_n)^T$$

such that

$$\mathbf{q}_i \approx \mathbf{p}_i^T \cdot \mathbf{A} + \mathbf{b} \quad (i = 1, \dots, m). \quad (3)$$

Dot product multiplication of matrices does not commute. However, transposing both matrices allows us to reverse the order of the terms in the dot multiplication. The matrix \mathbf{A} is a square unknown, so it does not require any treatment, whereas the column vector \mathbf{p}_i is transposed to a row vector \mathbf{p}_i^T for this purpose. The reason for this rearrangement is that NumPy [1, 2] supports very fast and simple to perform dot product multiplication of vector arrays with matrices. This multiplication of the whole array of vectors can be executed very efficiently in a single function call (see for an example line 116 in the listing of Sect. 5.2).

The number of unknowns is $n^2 + n$, therefore we need at least $m > n^2 + n$. For this linear regression computation the Euclidean error ϵ needs to be minimised globally.

$$\epsilon_i = \mathbf{p}_i^T \cdot \mathbf{A} + \mathbf{b} - \mathbf{q}_i \quad (4)$$

A way to do this is to find the least sum of the squared errors Q :

$$Q(\mathbf{A}, \mathbf{b}) = \sum_{i=1}^m \|\mathbf{p}_i^T \cdot \mathbf{A} + \mathbf{b} - \mathbf{q}_i\|^2$$

$$= \sum_{i=1}^m \left[\begin{pmatrix} p_{1i} \\ \vdots \\ p_{ni} \end{pmatrix}^T \cdot \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} - \begin{pmatrix} q_{1i} \\ \vdots \\ q_{ni} \end{pmatrix} \right]^2 \quad (5)$$

$$= \sum_{i=1}^m \left[\begin{pmatrix} p_{1i}a_{11} + p_{2i}a_{21} + \cdots + p_{ni}a_{n1} \\ p_{1i}a_{12} + p_{2i}a_{22} + \cdots + p_{ni}a_{n2} \\ \vdots \\ p_{1i}a_{1n} + p_{2i}a_{2n} + \cdots + p_{ni}a_{nn} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} - \begin{pmatrix} q_{1i} \\ q_{2i} \\ \vdots \\ q_{ni} \end{pmatrix} \right]^2$$

²<http://elonen.iki.fi/code/misc-notes/affine-fit/>

The minimum can be found by determining \mathbf{A} and \mathbf{b} where the partial derivatives of Q become zero:

$$\begin{aligned}
 \frac{\partial Q}{\partial a_{11}} = 0 &= 2 \sum_{i=1}^m p_{1i} (p_{1i}a_{11} + p_{2i}a_{21} + \cdots + p_{ni}a_{n1} + b_1 - q_{1i}) \\
 \frac{\partial Q}{\partial a_{12}} = 0 &= 2 \sum_{i=1}^m p_{1i} (p_{1i}a_{12} + p_{2i}a_{22} + \cdots + p_{ni}a_{n2} + b_2 - q_{2i}) \\
 &\vdots \\
 \frac{\partial Q}{\partial a_{21}} = 0 &= 2 \sum_{i=1}^m p_{2i} (p_{1i}a_{11} + p_{1i}a_{21} + \cdots + p_{ni}a_{n1} + b_1 - q_{1i}) \\
 \frac{\partial Q}{\partial a_{22}} = 0 &= 2 \sum_{i=1}^m p_{2i} (p_{1i}a_{12} + p_{2i}a_{22} + \cdots + p_{ni}a_{n2} + b_2 - q_{2i}) \\
 &\vdots \\
 \frac{\partial Q}{\partial a_{nn}} = 0 &= 2 \sum_{i=1}^m p_{ni} (p_{1i}a_{1n} + p_{2i}a_{2n} + \cdots + p_{ni}a_{nn} + b_n - q_{ni})
 \end{aligned} \tag{6}$$

and

$$\begin{aligned}
 \frac{\partial Q}{\partial b_1} = 0 &= 2 \sum_{i=1}^m (p_{1i}a_{11} + p_{2i}a_{21} + \cdots + p_{ni}a_{n1} + b_1 - q_{1i}) \\
 \frac{\partial Q}{\partial b_2} = 0 &= 2 \sum_{i=1}^m (p_{1i}a_{12} + p_{2i}a_{22} + \cdots + p_{ni}a_{n2} + b_2 - q_{2i}) \\
 &\vdots \\
 \frac{\partial Q}{\partial b_n} = 0 &= 2 \sum_{i=1}^m (p_{1i}a_{1n} + p_{2i}a_{2n} + \cdots + p_{ni}a_{nn} + b_n - q_{ni})
 \end{aligned} \tag{7}$$

From (6) and (7) we obtained a linear equation system we can describe in the form

$$\mathbf{C}\mathbf{x} = \mathbf{d} \tag{8}$$

with the sparse matrix \mathbf{C} to solve for the “solution vector” x containing the unknowns of the matrix \mathbf{A} as well as the vector \mathbf{b} . The system has got $t = n^2 + n$ unknowns. The $t \times t$ size matrix is populated and solved using the `numpy.linalg.solve` solver from the NumPy package. Alternatively, the system could also easily be described as a set of n equation systems consisting of one equation for each system containing of a_{ij} with $i = 1, \dots, n$ from (6), and one equation for b_i from (7). This approach would result in less memory consumption for the then non-sparse matrix and less memory consumption for the solver.

3 Requirements

The code solely depends on a (not too old) (C) Python as well as NumPy [1, 2]. It has been developed and tested under different versions of Python 2.6 with NumPy in the versions 1.2.1 and 1.3.0.

4 Results

The code as presented in this paper (see Sect. Implementation below) performs quite well. It is possible to construct simpler numerical problems (as described in the last paragraph of Sect. 2). But the solver used by NumPy is extremely capable. Therefore, the extra effort of refactoring for better memory efficiency was not justified for our problems at hand. In fact, the implementation would have been less clear, so this was not pursued any further. The efficiency of the implementation became obvious after performing some tests. It was capable of solving a vector field regression problem with a set of 2000 sample points for each of P and Q in a 42-dimensional vector field in about 3.5 seconds (on an Intel T7500 Core 2 Duo CPU, 2.20 GHz, 2 GB RAM, under Ubuntu Lucid Lynx 10.04).

This implementation has been used with minor variation in a series of different cases. It has always performed more than adequately in execution time, and has yielded correct results for all suitable problems. As we are using the solver from the (already used) NumPy package, so the implementation of the `solve()` function is very compact and fast in comparison to a self implemented alternative in pure Python.

5 Implementation

This code is free software: you can redistribute and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

5.1 The `solve()` Function

The `solve()` function sets up the actual linear equation system to determine the best fitting affine transformation from P to Q .

```

3 import numpy
5 def solve(point_list):
6     """
7     This function solves the linear equation system involved in the n
8     dimensional linear extrapolation of a vector field to an arbitrary point.
10
11      $f(x) = x * A + b$ 
12
13     with:
14     A - The "slope" of the affine function in an n x n matrix.
15     b - The "offset" value for the n dimensional zero vector.
16
17     The function takes a list of n+1 point-value tuples (x, f(x)) and returns
18     the matrix A and the vector b. In case anything goes wrong, the function
19     returns the tuple (None, None).
20
21     These can then be used to compute directly any value in the linear
22     vector field.
23     """
24     # Some helpers.
25     dimensions = len(point_list[0][0])
26     unknowns = dimensions ** 2 + dimensions
27     number_points = len(point_list[0])
28
29     # Bail out if we do not have enough data.
30     if number_points < unknowns:
31         print 'For a %d dimensional problem I need at least %d data points.' \
32             % (dimensions, unknowns)
33         print 'Only %d data points were given.' % number_points
34         return None, None

```

```

35     # Ensure we are working with a NumPy array.
36     point_list = numpy.asarray(point_list)

38     # For the solver we are stating the problem as
39     # C * x = d
40     # with the problem_matrix C and the problem_vector d

42     # We're going to feed our linear problem into these arrays.
43     # This one is the matrix C.
44     problem_matrix = numpy.zeros([unknowns, unknowns])
45     # This one is the vector d.
46     problem_vector = numpy.zeros([unknowns])

48     # Populate data matrix C and vector d.
49     x_values, y_values = point_list[0], point_list[1]
50     for i in range(dimensions):
51         x_i, y_i = x_values[:, i], y_values[:, i]
52         for j in range(dimensions):
53             y_j = y_values[:, j]
54             row = dimensions * i + j
55             problem_vector[row] = (x_i * y_j).sum()
56             problem_matrix[row, dimensions ** 2 + j] = x_i.sum()
57             problem_matrix[dimensions ** 2 + j, dimensions * i + j] = x_i.sum()
58             for k in range(dimensions):
59                 x_k = x_values[:, k]
60                 problem_matrix[row, dimensions * k + j] = (x_i * x_k).sum()
61         row = dimensions ** 2 + i
62         problem_vector[row] = y_i.sum()
63         problem_matrix[row, dimensions ** 2 + i] = number_points

66     matrix_A, vector_b = None, None
67     try:
68         result_vector = numpy.linalg.solve(problem_matrix, problem_vector)

70     # Check whether we really did get the right answer.
71     # This is advised by the NumPy doc string.
72     if numpy.linalg.norm(numpy.dot(problem_matrix, result_vector)
73                          - problem_vector) < 1e-6:
74         # We're good, so hack up the result into the matrix and vector.
75         matrix_A = result_vector[:dimensions ** 2]
76         matrix_A.shape = (dimensions, dimensions)
77         vector_b = result_vector[dimensions ** 2:]
78     else:
79         print "For whatever reason our linear equations didn't solve."
80         print numpy.linalg.norm(numpy.dot(result_vector, problem_matrix)
81                                - problem_vector)
82     except numpy.linalg.linalg.LinAlgError:
83         print "Things didn't work out as expected, eh."

85     return matrix_A, vector_b

```

5.2 The main() Function

The `main()` function contains a bit of bootstrapping code to set up an example of 100 random data points P (lines 110–113) in \mathbb{R}^3 space (lines 100–101), create a random affine transformation (lines 103–108) to compute a set of corresponding points Q (lines 115–116) that have been “peppered” with a bit of noise (lines 118–123). The problem is solved, checked for the presence of a solution, and finally the results of estimated versus the expected values are printed.

```

88 def main():
89     """
90     We're testing the 3D case of a mapping
91
92     y = x * A + b
93
94     We are setting up a mystery matrix (A) and vector (b), creating a solid
95     number of random samples with noise from that, and then we're finding
96     out a good approximation for A and b by solving the regression problem.
97     The approximation is compared to the original mystery values.
98     """
99     # We're testing the 3D case. And using 100 data points.
100    dimensions = 3
101    data_points = 100
102
103    # Let's make a mystery matrix and a mystery vector with elements < 10.0.
104    mystery_matrix = numpy.random.uniform(low=0.0, high=10.0,
105                                         size=dimensions ** 2)
106    mystery_matrix.shape = (dimensions, dimensions)
107    mystery_vector = numpy.random.uniform(low=0.0, high=10.0,
108                                         size=dimensions)
109
110    # Now let's make 100 sample points ...
111    x_values = numpy.random.uniform(low=0.0, high=10.0,
112                                   size=dimensions * data_points)
113    x_values.shape = (data_points, dimensions)
114
115    # ... and calculate their corresponding values ...
116    y_values = numpy.dot(x_values, mystery_matrix) + mystery_vector
117
118    # ... and add a bit of noise.
119    noise_scale = 1.5
120    noise = numpy.random.normal(loc=0.0, scale=noise_scale,
121                               size=dimensions * data_points)
122    noise.shape = (data_points, dimensions)
123    y_values += noise
124
125    # Solve the n-D linear regression problem.
126    estimated_A, estimated_b = solve([x_values, y_values])
127
128    if estimated_A is not None:
129        # And test it on a nice point somewhere
130        dest_point = numpy.random.uniform(low=0.0, high=10.0,
131                                         size=dimensions)
132        expected_value = numpy.dot(dest_point, mystery_matrix) + mystery_vector
133        estimated_value = numpy.dot(dest_point, estimated_A) + estimated_b
134        distance = numpy.linalg.norm(expected_value - estimated_value)
135
136        print 'Point at:\n\t%s' % dest_point.round(3)
137        print 'Expected:\n\t%s' % expected_value.round(3)
138        print 'Estimated:\n\t%s' % estimated_value.round(3)
139        print 'Distance:\n\t%s' % distance.round(3)
140    else:
141        print 'Sorry, problem could not be solved.'

```

References

- [1] T. E. Oliphant, “NumPy Project,” <http://numpy.scipy.org/>, last accessed July 2010.
- [2] —, *Guide to NumPy*, T. E. Oliphant, Ed. Trelgol Publishing, 2006. [Online] <http://www.tramy.us/guidetoscopy.html>
- [3] Wikipedia, “Simple linear regression,” http://en.wikipedia.org/wiki/Simple_linear_regression, last accessed July 2010.
- [4] H. Späth, “Fitting affine and orthogonal transformations between two sets of points,” *Mathematical Communications*, vol. 9, no. 1, pp. 27–34, June 2004.