# Nonlinear Dimensionality Reduction I: Local Linear Embedding

36-350, Data Mining

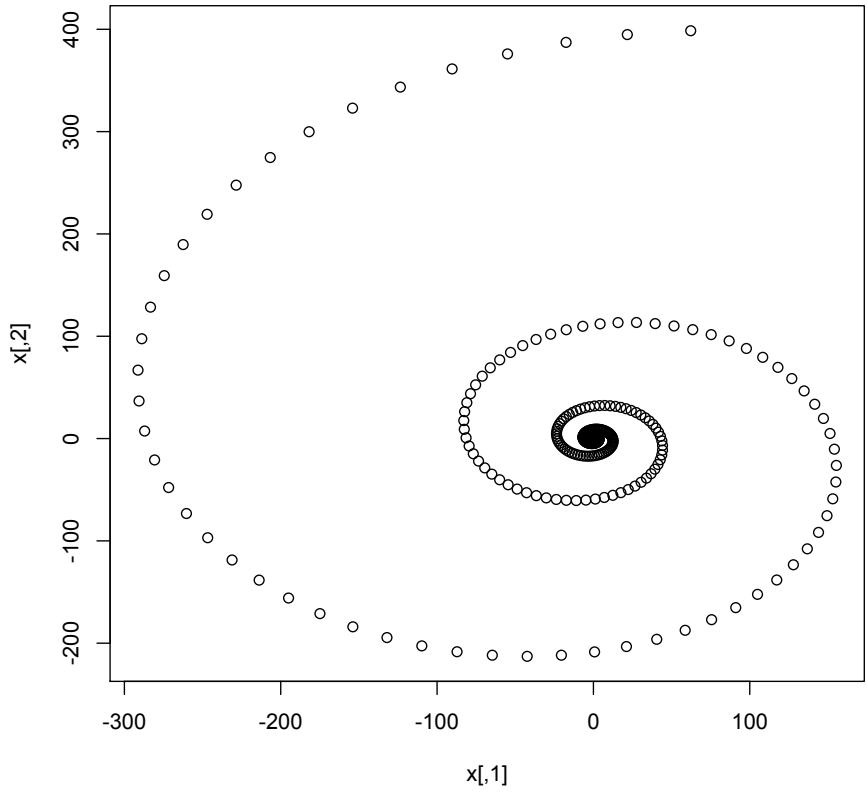
5 October 2009

## Contents



# 1 Why We Need Nonlinear Dimensionality Reduction

Consider the points shown in Figure 1. Even though there are two features, a.k.a. coordinates, all of the points fall on a one-dimensional curve (as it happens, a logarithmic spiral). This is exactly the kind of constraint which it would be good to recognize and exploit — rather than using two separate coordinates, we could just say how far along the curve a data-point is.




```
x=matrix(c(exp(-0.2*(-(1:300)/10))*cos(-(1:300)/10),
           exp(-0.2*(-(1:300)/10))*sin(-(1:300)/10)),
         ncol=2)
plot(x)
```

Figure 1: Two-dimensional data constrained to a smooth one-dimensional region, namely the logarithmic spiral, $r = e^{-0.2\theta}$ in polar coordinates.

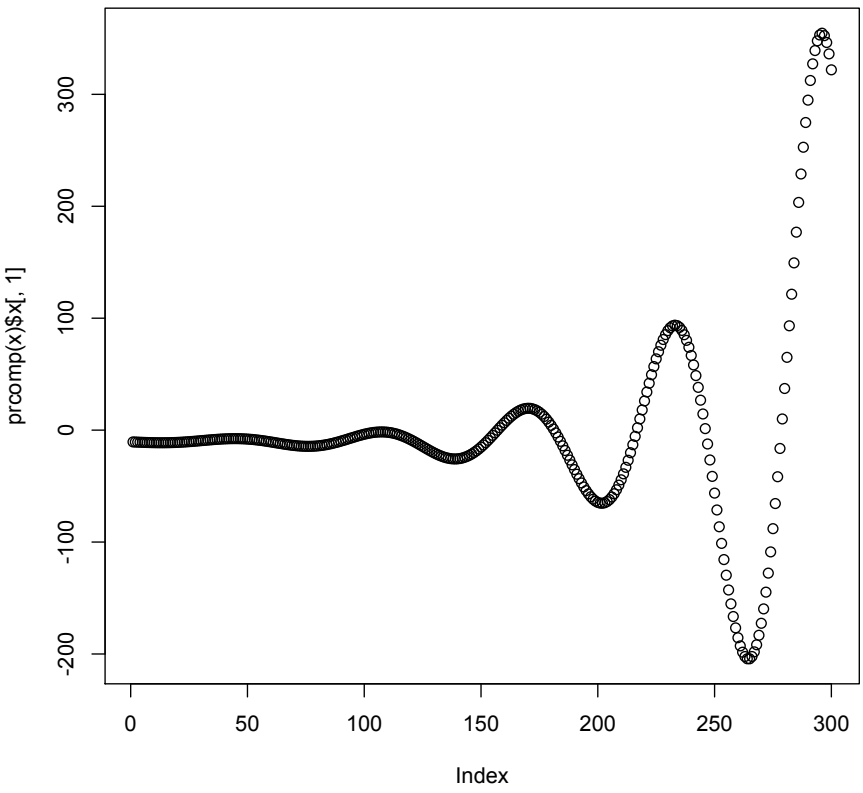
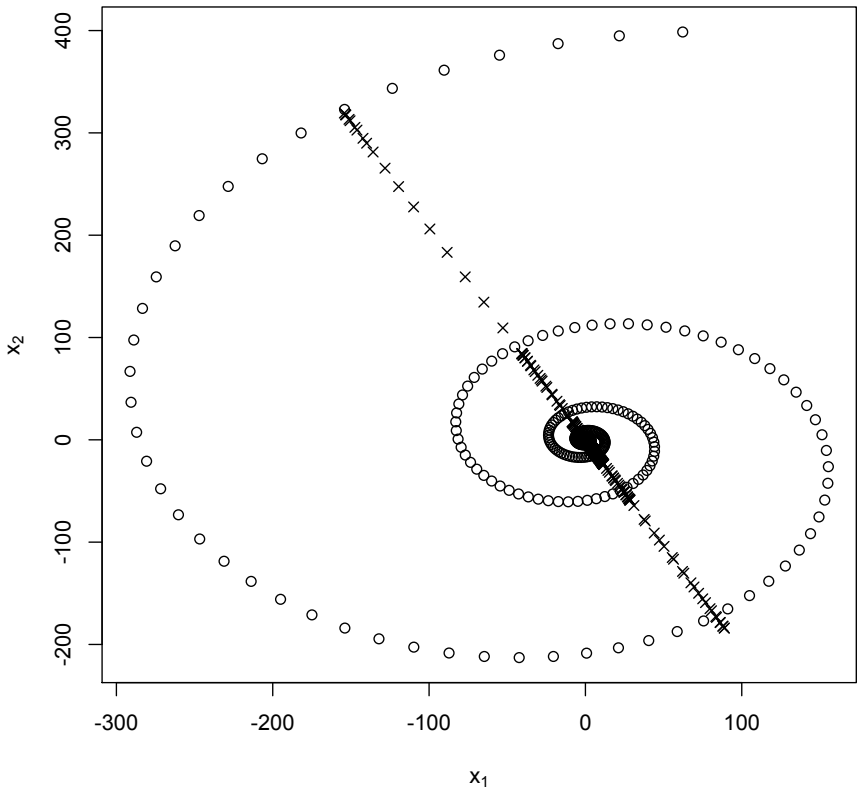



Figure 2: Projections of the spiral points on to their first principal component.

PCA will do poorly with data like this. Remember that to get a one-dimensional representation out of it, we need to take the first principal component, which is *straight line* along which the data's projections have the most variance. If this works for capturing structure along the spiral, then projections on to the first PC should have the same order that the points have along the spiral.[1] Since, fortuitously, the data are already in that order, we can just plot the first PC against the index (Figure 2). The results are — there is really no other word for it — *screwy*.

So, PCA with one principal component fails to capture the one-dimensional structure of the spiral. We could add another principal component, but then we've just rotated our two-dimensional data. In fact, *any* linear dimensionality-

[1] It wouldn't matter if the coordinate increased as we went out along the spiral or decreased, just so long as it was monotonic.




```
fit.all = prcomp(x)
approx.all=fit.all$x[,1]%*%t(fit.all$rotation[,1])
plot(x,xlab=expression(x[1]),ylab=expression(x[2]))
points(approx.all,pch=4)
```

Figure 3: Spiral data (circles) replotted with their one-dimensional PCA approximations (crosses).

reduction method is going to fail here, simply because the spiral is not even approximately a one-dimensional *linear subspace*.

What then are we to do?

1. Stick to not-too-nonlinear structures.
2. Somehow decompose nonlinear structures into linear subspaces.
3. Generalize the eigenvalue problem of minimizing distortion.

There's not a great deal to be said about (1). Some curves can be approximated by linear subspaces without too much heartbreak. (For instance, see Figure 4.) We can use things like PCA on them, and so long as we remember that we're just seeing an approximation, we won't go too far wrong. But fundamentally this is *weak*. (2) is hoping that we can somehow build a strong method out of this weak one; as it happens we can, and it's called locally linear embedding (and its variants). The last is diffusion maps, which we'll cover next lecture.

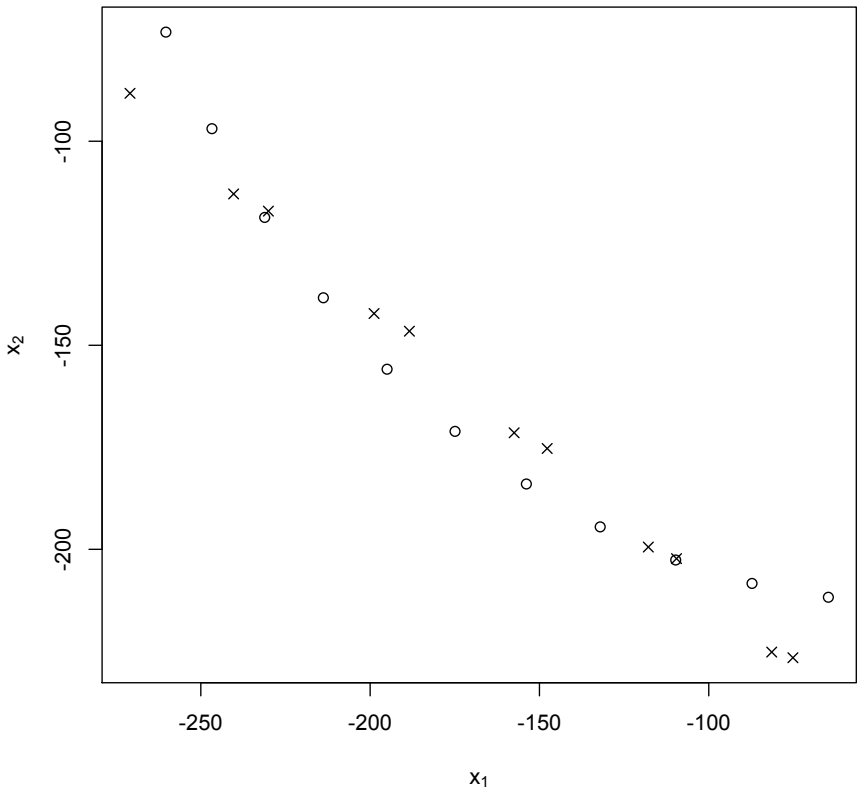
## 2 Local Linearity and Manifolds

Let's look again at Figure 4. A one-dimensional linear subspace is, in plain words, a straight line. By doing PCA on this part of the data alone, we are approximating a segment of the spiral curve by a straight line. Since the segment is not *very* curved, the approximation is reasonably good. (Or rather, the segment was chosen so the approximation would be good, consequently it had to have low curvature.) Notice that this error is not a random scatter of points around the line, but rather a systematic mis-match between the true curve and the line — a **bias** which would not go away no matter how much data we had from the spiral. The size of the bias depends on how big a region we are using, and how much the tangent direction to the curve changes across that region — the average **curvature**. By using small regions when the curvature is high and big regions when the curvature is low, we can maintain any desired degree of approximation.

If we shifted to a different part of the curve, we could do PCA on the data there, too, getting a different principal component and a different linear approximations to the data. Generally, as we move around the degree of curvature will change, so the size of the region we'd use would also need to grow or shrink.

This suggests that we could make some progress towards learning nonlinear structures in the data by patching together lots of linear structures. We could, for example, divide up the whole data space into regions, and do a separate PCA in each region. Here we'd hope that in each region we needed only a single principal component. Such hopes would generally be dashed, however, because this is a bit too simple-minded to really work.

1. We'd need to chose the number of regions, introducing a trade-off between having many points in each region (so as to deal with noise) and having small regions (to keep the linear approximation good).




```
fit = prcomp(x[270:280,])
pca.approx = fit$x[,1]%*%t(fit$rotation[,1])+colMeans(x[270:280,])
plot(rbind(x[270:280,],pca.approx),type="n",
     xlab=expression(x[1]),ylab=expression(x[2]))
points(x[270:280,])
points(pca.approx,pch=4)
```

Figure 4: Portion of the spiral data (circles) together with *its* one-dimensional PCA approximation (crosses).

2. Ideally, the regions should be of different sizes, depending on average curvature, but we don't know the curvature.

3. What happens at the boundaries between regions? The principal components of adjacent regions could be pointing in totally different directions.

Nonetheless, this is the core of a good idea. To make it work, we need to say just a little about **differential geometry**, specifically the idea of a **manifold**.[2] For our purposes, a manifold is a smooth, curved subset of a Euclidean space, in which it is **embedded**. The spiral curve (not the isolated points I plotted) is a one-dimensional manifold in the plane, just as are lines, circles, ellipses and parabolas. The surface of a sphere or a torus is a two-dimensional manifold, like a plane. The essential fact about a $q$-dimensional manifold is that it can be arbitrarily well-approximated by a $q$-dimensional linear subspace, the **tangent space**, by taking a sufficiently small region about any point.[3] (This generalizes the fact any sufficiently small part of a curve about any point looks very much like a straight line, the tangent line to the curve at that point.) Moreover, as we move from point to point, the local linear approximations change continuously, too. The more rapid the change in the tangent space, the bigger the curvature of the manifold. (Again, this generalizes the relation between curves and their tangent lines.) So if our data come from a manifold, we should be able to do a local linear approximation around every part of the manifold, and then smoothly interpolate them together into a single global system. To do dimensionality reduction — to learn the manifold — we want to find these global low-dimensional coordinates.[4]

# 3 Locally Linear Embedding (LLE)

Locally linear embedding (or: local linear embedding, you see both) is a clever scheme for finding low-dimensional global coordinates when the data lie on (or

[2]Differential geometry is a very beautiful and important branch of mathematics, with its roots in the needs of geographers in the 1800s to understand the curved surface of the Earth in detail (**geodesy**). The theory of curved spaces they developed for this purpose generalized the ordinary vector calculus and Euclidean geometry, and turned out to provide the mathematical language for describing space, time and gravity (Einstein's general theory of relativity; Lawrie (1990)), the other fundamental forces of nature (gauge field theory; Lawrie (1990)), dynamical systems Arnol'd (1973); Guckenheimer and Holmes (1983), and indeed statistical inference (information geometry; Kass and Vos (1997); Amari and Nagaoka (1993/2000)). Good introductions are Spivak (1965) and Schutz (1980) (which confines the physics to one (long) chapter on applications).

[3]If it makes you happier: every point has an open neighborhood which is homeomorphic to $\mathbb{R}^q$, and the transition from neighborhood to neighborhood is continuous and differentiable.

[4]There are technicalities here which I am going to gloss over, because this is not a class in differential geometry. (Take one, it's good for you!) The biggest one is that most manifolds don't admit of a truly *global* coordinate system, one which is good everywhere without exception. But the places where it breaks down are usually isolated point and easily identified. For instance, if you take a sphere, almost every point can be identified by latitude and longitude — except for the poles, where longitude becomes ill-defined. Handling this in a mathematically precise way is tricky, but since these are probability-zero cases, we can ignore them in a statistics class.

very near to) a manifold embedded in a high-dimensional space. The trick is to do a different linear dimensionality reduction at each point (because locally a manifold looks linear) and then combine these with minimal discrepancy. It was introduced by Roweis and Saul (2000), though Saul and Roweis (2003) has a fuller explanation. I don't think it uses any elements which were unknown, mathematically, since the 1950s. Rather than diminishing what Roweis and Saul did, this should make the rest of us feel humble...

The LLE procedure has three steps: it builds a neighborhood for each point in the data; finds the weights for linearly approximating the data in that neighborhood; and finally finds the low-dimensional coordinates best reconstructed by those weights. This low-dimensional coordinates are then returned.

To be more precise, the LLE algorithm is given as inputs an $n \times p$ data matrix $\mathbf{X}$, with rows $\vec{x}_i$; a desired number of dimensions $q < p$; and an integer $k$ for finding local neighborhoods, where $k \geq q + 1$. The output is supposed to be an $n \times q$ matrix $\mathbf{Y}$, with rows $\vec{y}_i$.

1. For each $\vec{x}_i$, find the $k$ nearest neighbors.
2. Find the weight matrix $\mathbf{w}$ which minimizes the residual sum of squares for reconstructing each $\vec{x}_i$ from its neighbors,

$$RSS(\mathbf{w}) \equiv \sum_{i=1}^{n} \|\vec{x}_i - \sum_{j \neq i} w_{ij}\vec{x}_j\|^2 \tag{1}$$

   where $w_{ij} = 0$ unless $\vec{x}_j$ is one of $\vec{x}_i$'s $k$-nearest neighbors, and for each $i$, $\sum_j w_{ij} = 1$. (I will come back to this constraint below.)
3. Find the coordinates $\mathbf{Y}$ which minimize the reconstruction error using the weights,

$$\Phi(\mathbf{Y}) \equiv \sum_{i=1}^{n} \|\vec{y}_i - \sum_{j \neq i} w_{ij}\vec{y}_j\|^2 \tag{2}$$

   subject to the constraints that $\sum_i Y_{ij} = 0$ for each $j$, and that $\mathbf{Y}^T\mathbf{Y} = \mathbf{I}$. (I will come back to those constraints below, too.)

## 3.1 Finding Neighborhoods

In step 1, we define local neighborhoods for each point. By defining these in terms of the $k$ nearest neighbors, we make them physically large where the data points are widely separated, and physically small when the density of the data is high. We don't *know* that the curvature of the manifold is low when the data are sparse, but we do know that, whatever is happening out there, we have very little idea what it is, so it's safer to approximate it crudely. Conversely, if the data are dense, we can capture both high and low curvature. If the actual curvature is low, we might have been able to expand the region without loss, but again, this is playing it safe. So, to summarize, using $k$-nearest neighborhoods means

we take a fine-grained view where there is a lot of data, and a coarse-grained view where there is little data.

It's not strictly necessary to use $k$-nearest neighbors here; the important thing is to establish *some* neighborhood for each point, and to do so in a way which conforms or adapts to the data.

## 3.2 Finding Weights

Step 2 can be understood in a number of ways. Let's start with the local linearity of a manifold. Suppose that the manifold was *exactly* linear around $\vec{x}_i$, i.e., that it and its neighbors belonged to a $q$-dimensional linear subspace. Since $q+1$ points in generally define a $q$-dimensional subspace, there would be *some* combination of the neighbors which reconstructed $\vec{x}_i$ exactly, i.e., some set of weights $w_{ij}$ such that

$$\vec{x}_i = \sum_j w_{ij} \vec{x}_j \tag{3}$$

Conversely, if there are such weights, then $\vec{x}_i$ and (some of) its neighbors *do* form a linear subspace. Since every manifold is locally linear, by taking a sufficiently small region around each point we get arbitrarily close to having these equations hold — $n^{-1} RSS(\mathbf{w})$ should shrink to zero as $n$ grows.

Vitally, the *same* weights would work to reconstruct $\mathbf{x}_i$ both in the high-dimensional embedding space and the low-dimensional subspace. This means that it is the weights around a given point which characterize what the manifold looks like there (provided the neighborhood is small enough compared to the curvature). Finding the weights gives us the same information as finding the tangent space. This is why, in the last step, we will only need the weights, not the original vectors.

Now, about the constraints that $\sum_j w_{ij} = 1$. This can be understood in two ways, geometrically and probabilistically. Geometrically, what it gives us is invariance under translation. That is, if we add any vector $\vec{c}$ to $\vec{x}_i$ and all of its neighbors, nothing happens to the function we're minimizing:

$$\begin{aligned} \vec{x}_i + \vec{c} - \sum_j w_{ij}(\vec{x}_j + \vec{c}) &= \vec{x}_i + \vec{c} - \left( \sum_j w_{ij} \vec{x}_j \right) - \vec{c} & (4) \\ &= \vec{x}_i - \sum_j w_{ij} \vec{x}_j & (5) \end{aligned}$$

Since we are looking at the same shape of manifold no matter how we move it around in space, translational invariance is a constraint we want to impose.

Probabilistically, forcing the weights to sum to one makes $\mathbf{w}$ a stochastic transition matrix.[5] This should remind you of page-rank, where we built a

[5] Actually, it really only does that if $w_{ij} \geq 0$. In that case we are approximating $\vec{x}_i$ not just by a linear combination of its neighbors, but by a **convex** combination. Often one gets all positive weights anyway, but it can be helpful to impose this extra constraint.

Markov chain transition matrix from the graph connecting web-pages. There is a tight connection here, which we'll return to next time under the heading of **diffusion maps**; for now this is just to tantalize.

We will see below how to actually minimize the squared error computationally; as you probably expect by now, it reduces to an eigenvalue problem. Actually it reduces to a bunch ($n$) of eigenvalue problems: because there are no constraints *across* the rows of $\mathbf{w}$, we can find the optimal weights for each point separately. Naturally, this simplifies the calculation.

### 3.2.1 $k > p$

If $k$, the number of neighbors, is greater than $p$, the number of features, then (in general) the space spanned by $k$ distinct vectors is the whole space. Then $\vec{x}_i$ can be written *exactly* as a linear combination of its $k$-nearest neighbors.[6] In fact, if $k > p$, then not only is there a solution to $\vec{x}_i = \sum_j w_{ij}\vec{j}$, there are generally *infinitely many* solutions, because there are more unknowns ($k$) than equations ($p$). When this happens, we say that the optimization problem is **ill-posed**, or **irregular**. There are many ways of **regularizing** ill-posed problems. A common one, for this case, is what is called $L_2$ or **Tikhonov** regularization: instead of minimizing

$$\|\vec{x}_i - \sum_j w_{ij}\vec{x}_j\|^2 \tag{6}$$

pick an $\alpha > 0$ and minimize

$$\|\vec{x}_i - \sum_j w_{ij}\vec{x}_j\|^2 + \alpha \sum_j w_{ij}^2 \tag{7}$$

This says: pick the weights which minimize a combination of reconstruction error *and* the sum of the squared weights. As $\alpha \to 0$, this gives us back the least-squares problem. To see what the second, sum-of-squared-weights term does, take the opposite limit, $\alpha \to \infty$: the squared-error term becomes negligible, and we just want to minimize the Euclidean ("$L_2$") norm of the weight vector $w_{ij}$. Since the weights are constrained to add up to 1, we can best achieve this by making all the weights equal — so some of them can't be vastly larger than the others, and they stabilize at a definite preferred value. Typically $\alpha$ is set to be small, but not zero, so we allow some variation in the weights if it really helps improve the fit.

We will see how to actually implement this regularization later, when we look at the eigenvalue problems connected with LLE. The $L_2$ term is an example of a **penalty term**, used to stabilize a problem where just matching the data gives irregular results, and there is an art to optimally picking $\lambda$; in practice, however, LLE results are often fairly insensitive to it, when it's needed at all. Remember, the whole situation only comes up when $k > p$, and $p$ can easily be very large — 6380 for the gene-expression data, much larger for the *Times* corpus, etc.

[6] This is easiest to see when $\vec{x}_i$ lies inside the body which has its neighbors as vertices, their **convex hull**, but is true more generally.

(The fact that the scaling factor for the penalty term is a Greek letter is no accident. If we set as a constraint that $\sum_j w_{ij}^2 \leq c$, the natural way to enforce it in the optimization problem would be through a Lagrange multiplier, say $\alpha$, and we would end up minimizing

$$\|\vec{x}_i - \sum_j w_{ij}\vec{x}_j\|^2 - \alpha\left(c - \sum_j w_{ij}^2\right) \tag{8}$$

However, the $\alpha c$ term drops out when we take derivatives with respect to $w$. There is a correspondence, generally not worth working out in detail, between $\alpha$ and $c$, with big values of $\alpha$ implying small values of $c$ and vice versa. In fact, the usual symbol is $\lambda$ instead of $\alpha$, but we'll be wanting $\lambda$ for Lagrange multipliers enforcing explicit constraints later.)

### 3.3 Finding Coordinates

As I said above, if the local neighborhoods are small compared to the curvature of the manifold, weights in the embedding space and weights on the manifold should be the same. (More precisely, the two sets of weights are exactly equal for linear subspaces, and for other manifolds they can be brought arbitrarily close to each other by shrinking the neighborhood sufficiently.) In the third and last step of LLE, we have just calculated the weights in the embedding space, so we take them to be *approximately* equal to the weights on the manifold, and solve for coordinates on the manifold.

So, taking the weight matrix $\mathbf{w}$ as fixed, we ask for the $\mathbf{Y}$ which minimizes

$$\Phi(\mathbf{Y}) = \sum_i \left\| \vec{y}_i - \sum_{j \neq i} \vec{y}_j w_{ij} \right\|^2 \tag{9}$$

That is, what should the coordinates $\vec{y}_i$ be on the manifold, that *these* weights reconstruct them?

As mentioned, some constraints are going to be needed. Remember that we saw above that we could add any constant vector $\vec{c}$ to $\vec{x}_i$ and its neighbors without affecting the sum of squares, because $\sum_j w_{ij} = 1$. We could do the same with the $\vec{y}_i$, so the minimization problem, as posed, has an infinity of equally-good solutions. To fix this — to "break the degeneracy" — we impose the constraint

$$\frac{1}{n}\sum_i \vec{y}_i = 0 \tag{10}$$

Since if the mean vector was *not* zero, we could just subtract it from all the $\vec{y}_i$ without changing the quality of the solution, this is just a book-keeping convenience.

Similarly, we also impose the convention that

$$\frac{1}{n}\mathbf{Y}^T\mathbf{Y} = \mathbf{I} \tag{11}$$

i.e., that the covariance matrix of $\mathbf{Y}$ be the ($q$-dimensional) identity matrix. This is not as substantial as it looks. If we found a solution where the covariance matrix of $\mathbf{Y}$ was *not* diagonal, we could use PCA to rotate the new coordinates on the manifold so they were uncorrelated, giving a diagonal covariance matrix. The only bit of this which is not, again, a book-keeping convenience is assuming that all the coordinates have the same variance — that the diagonal covariance matrix is in fact $\mathbf{I}$.

This optimization problem is like multi-dimensional scaling: we are asking for low-dimensional vectors which preserve certain relationships (averaging weights) among high-dimensional vectors. We are also asking to do it under constraints, which we will impose through Lagrange multipliers. Once again, it turns into an eigenvalue problem, though one just a bit more subtle than what we saw with PCA.

(One reason to suspect the appearance of eigenvalues, in addition to my very heavy-handed foreshadowing, is that eigenvectors are automatically orthogonal to each other and normalized, so making the columns of $\mathbf{Y}$ be the eigenvectors of some matrix would automatically satisfy Eq. 11.)

Unfortunately, the finding the coordinates does *not* break up into $n$ smaller problems, the way finding the weights did, because each row of $\mathbf{Y}$ appears in $\Phi$ multiple times, once as the focal vector $\vec{y}_i$, and then again as one of the neighbors of other vectors.

# 4 More Fun with Eigenvalues and Eigenvectors

To sum up: for each $\vec{x}_i$, we want to find the weights $w_{ij}$ which minimize

$$RSS_i(\mathbf{w}) = \|\vec{x}_i - \sum_j w_{ij}\vec{x}_j\|^2 \tag{12}$$

where $w_{ij} = 0$ unless $\vec{x}_j$ is one of the $k$ nearest neighbors of $\vec{x}_i$, under the constraint that $\sum_j w_{ij} = 1$. Given those weights, we want to find the $q$-dimensional vectors $\vec{y}_i$ which minimize

$$\Phi(\mathbf{Y}) = \sum_{i=1}^{n} \|\vec{y}_i - \sum_j w_{ij}\vec{y}_j\|^2 \tag{13}$$

with the constraints that $n^{-1}\sum_i \vec{y}_i = 0$, $n^{-1}\mathbf{Y}^T\mathbf{Y} = \mathbf{I}$.

## 4.1 Finding the Weights

In this subsection, assume that $j$ just runs over the neighbors of $\vec{x}_i$, so we don't have to worry about the weights (including $w_{ii}$) which we know are zero.

We saw that $RSS_i$ is invariant if we add an arbitrary $\vec{c}$ to all the vectors.

Set $\vec{c} = -\vec{x}_i$, centering the vectors on the focal point $\vec{x}_i$:

$$RSS_i = \| \sum_j w_{ij}(\vec{x}_j - \vec{x}_i) \|^2 \quad (14)$$

$$= \| \sum_j w_{ij}\vec{z}_j \|^2 \quad (15)$$

defining $\vec{z}_j = \vec{x}_j - \vec{x}_i$. If we correspondingly define the $k \times p$ matrix $\mathbf{z}$, and set $\mathbf{w}_i$ to be the $k \times 1$ matrix, the vector we get from the sum is just $\mathbf{w}_i^T\mathbf{z}$. The squared magnitude of any vector $\vec{r}$, considered as a row matrix $\mathbf{r}$, is $\mathbf{r}\mathbf{r}^T$, so

$$RSS_i = \mathbf{w}_i^T\mathbf{z}\mathbf{z}^T\mathbf{w}_i \quad (16)$$

Notice that $\mathbf{z}\mathbf{z}^T$ is a $k \times k$ matrix consisting of all the inner products of the neighbors. This symmetric matrix is called the **Gram matrix** of the set of vectors, and accordingly abbreviated $\mathbf{G}$ — here I'll say $\mathbf{G}_i$ to remind us that it depends on our choice of focal point $\vec{x}_i$.

$$RSS_i = \mathbf{w}_i^T\mathbf{G}_i\mathbf{w}_i \quad (17)$$

Notice that the data matter only in so far as they determine the Gram matrix $\mathbf{G}_i$; the problem is invariant under any transformation which leaves all the inner products alone (translation, rotation, mirror-reversal, etc.).

We want to minimize $RSS_i$, but we have the constraint $\sum_j w_{ij} = 1$. We impose this via a Lagrange multiplier, $\lambda$.[7] To express the constraint in matrix form, introduce the $k \times 1$ matrix of all 1s, call it $\mathbf{1}$.[8] Then the constraint has the form $\mathbf{1}^T\mathbf{w}_i = 1$, or $\mathbf{1}^T\mathbf{w}_i - 1 = 0$. Now we can write the Lagrangian:

$$\mathcal{L}(\mathbf{w}_i, \lambda) = \mathbf{w}_i^T\mathbf{G}_i\mathbf{w}_i - \lambda(\mathbf{1}^T\mathbf{w} - 1) \quad (18)$$

Taking derivatives, and remembering that $G_i$ is symmetric,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_i} = 2\mathbf{G}_i\mathbf{w}_i - \lambda\mathbf{1} = 0 \quad (19)$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = \mathbf{1}^T\mathbf{w}_i - 1 = 0 \quad (20)$$

or

$$\mathbf{G}_i\mathbf{w}_i = \frac{\lambda}{2}\mathbf{1} \quad (21)$$

If the Gram matrix is invertible,

$$\mathbf{w}_i = \frac{\lambda}{2}\mathbf{G}_i^{-1}\mathbf{1} \quad (22)$$

where $\lambda$ can be adjusted to ensure that everything sums to 1.

[7] This $\lambda$ should not be confused with the penalty-term $\lambda$ used when $k > p$. See next sub-section.

[8] This should not be confused with the identity matrix, $\mathbf{I}$.

#### 4.1.1 $k > p$

If $k > p$, we modify the objective function to be

$$\mathbf{w}_i^T \mathbf{G}_i \mathbf{w}_i + \alpha \mathbf{w}_i^T \mathbf{w}_i \tag{23}$$

where $\alpha > 0$ determines the degree of regularization. Proceeding as before to impose the constraint,

$$\mathcal{L} = \mathbf{w}_i^T \mathbf{G}_i \mathbf{w}_i + \alpha \mathbf{w}_i^T \mathbf{w}_i - \lambda(\mathbf{1}^T \mathbf{w}_i - 1) \tag{24}$$

where now $\lambda$ is the Lagrange multiplier. Taking the derivative with respect to $\mathbf{w}_i$ and setting it to zero,

$$\begin{aligned} 2\mathbf{G}_i \mathbf{w}_i + 2\alpha \mathbf{w}_i &= \lambda \mathbf{1} && (25) \\ (\mathbf{G}_i + \alpha \mathbf{I})\mathbf{w}_i &= \frac{\lambda}{2}\mathbf{1} && (26) \\ \mathbf{w}_i &= \frac{\lambda}{2}(\mathbf{G}_i + \alpha \mathbf{I})^{-1}\mathbf{1} && (27) \end{aligned}$$

where, again, we pick $\lambda$ to properly normalize the right-hand side.

### 4.2 Finding the Coordinates

As with PCA, it's easier to think about the $q = 1$ case first; the general case follows similar lines. So $\vec{y}_i$ is just a single scalar number, $y_i$, and $\mathbf{Y}$ reduces to an $n \times 1$ column of numbers. We'll revisit $q > 1$ at the end.

The objective function is

$$\begin{aligned} \Phi(\mathbf{Y}) &= \sum_{i=1}^{n} \left( y_i - \sum_j w_{ij} y_j \right)^2 && (28) \\ &= \sum_{i=1}^{n} y_i^2 - y_i \left( \sum_j w_{ij} y_j \right) - \left( \sum_j w_{ij} y_j \right) y_i + \left( \sum_j w_{ij} y_j \right)^2 && (29) \\ &= \mathbf{Y}^T \mathbf{Y} - \mathbf{Y}^T (\mathbf{w}\mathbf{Y}) - (\mathbf{w}\mathbf{Y})^T \mathbf{Y} + (\mathbf{w}\mathbf{Y})^T (\mathbf{w}\mathbf{Y}) && (30) \\ &= ((\mathbf{I} - \mathbf{w})\mathbf{Y})^T ((\mathbf{I} - \mathbf{w})\mathbf{Y}) && (31) \\ &= \mathbf{Y}^T (\mathbf{I} - \mathbf{w})^T (\mathbf{I} - \mathbf{w}) \mathbf{Y} && (32) \end{aligned}$$

Define the $m \times m$ matrix $\mathbf{M} = (\mathbf{I} - \mathbf{w})^T (\mathbf{I} - \mathbf{w})$.

$$\Phi(\mathbf{Y}) = \mathbf{Y}^T \mathbf{M} \mathbf{Y} \tag{33}$$

This looks promising — it's the same sort of quadratic form that we maximized in doing PCA.

Now let's use a Lagrange multiplier $\mu$ to impose the constraint that $n^{-1}\mathbf{Y}^T\mathbf{Y} = \mathbf{I}$ — but, since $q = 1$, that's the $1 \times 1$ identity matrix, i.e., the scalar number 1.

$$\mathcal{L}(\mathbf{Y}, \mu) = \mathbf{Y}^T \mathbf{M} \mathbf{Y} - \mu(n^{-1}\mathbf{Y}^T\mathbf{Y} - 1) \tag{34}$$

Note that this $\mu$ is not the same as the $\mu$ which constrained the weights!

Proceeding as we did with PCA,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{Y}} = 2\mathbf{M}\mathbf{Y} - 2\mu n^{-1}\mathbf{Y} = 0 \tag{35}$$

or

$$\mathbf{M}\mathbf{Y} = \frac{\mu}{n}\mathbf{Y} \tag{36}$$

so $\mathbf{Y}$ must be an eigenvector of $\mathbf{M}$. Because $\mathbf{Y}$ is defined for each point in the data set, it is a function of the data-points, and we call it an **eigenfunction**, to avoid confusion with things like the eigenvectors of PCA (which are $p$-dimensional vectors in feature space). Because we are trying to minimize $\mathbf{Y}^T\mathbf{M}\mathbf{Y}$, we want the eigenfunctions going with the smallest eigenvalues — the **bottom** eigenfunctions — unlike the case with PCA, where we wanted the **top** eigenvectors.

$\mathbf{M}$ being an $n \times n$ matrix, it has, in general, $n$ eigenvalues, and $n$ mutually orthogonal eigenfunctions. The eigenvalues are real and non-negative; the smallest of them is always zero, with eigenfunction $\mathbf{1}$. To see this, notice that $\mathbf{w}\mathbf{1} = \mathbf{1}$.[9] Then

$$(\mathbf{I} - \mathbf{w})\mathbf{1} = 0 \tag{37}$$

$$(\mathbf{I} - \mathbf{w})^T(\mathbf{I} - \mathbf{w})\mathbf{1} = 0 \tag{38}$$

$$\mathbf{M}\mathbf{1} = 0 \tag{39}$$

Since this eigenfunction is constant, it doesn't give a useful coordinate on the manifold. To get our first coordinate, then, we need to take the two bottom eigenfunctions, and discard the constant.

Again as with PCA, if we want to use $q > 1$, we just need to take multiple eigenfunctions of $M$. To get $q$ coordinates, we take the bottom $q+1$ eigenfunctions, discard the constant eigenfunction with eigenvalue 0, and use the others as our coordinates on the manifold. Because the eigenfunctions are orthogonal, the no-covariance constraint is automatically satisfied. Notice that adding another coordinate just means taking another eigenfunction of the *same* matrix $\mathbf{M}$ — as is the case with PCA, but not with factor analysis.

(What happened to the mean-zero constraint? Well, we can add another Lagrange multiplier $\nu$ to enforce it, but the constraint is linear in $\mathbf{Y}$, it's $\mathbf{A}\mathbf{Y} = 0$ for some matrix $\mathbf{A}$ [Exercise: write out $\mathbf{A}$], so when we take partial derivatives we get

$$\frac{\partial \mathcal{L}(\mathbf{Y}, \mu, \nu)}{\partial \mathbf{Y}} = 2\mathbf{M}\mathbf{Y} - 2\mu\mathbf{Y} - \nu\mathbf{A} = 0 \tag{40}$$

and this is the *only* equation in which $\nu$ appears. So we are actually free to pick any $\nu$ we like, and may as well set it to be zero. Geometrically, this is the translational invariance yet again. In optimization terms, the size of the Lagrange multiplier tells us about how strongly the constraint pushes us away

[9] Each row of $\mathbf{w}\mathbf{1}$ is a weighted average of the other rows of $\mathbf{1}$. But all the rows of $\mathbf{1}$ are the same.

```
# Local linear embedding of data vectors
# Inputs: n*p matrix of vectors, number of dimensions q to find (< p),
  # number of nearest neighbors per vector, scalar regularization setting
# Calls: find.kNNs, reconstruction.weights, coords.from.weights
# Output: n*q matrix of new coordinates
lle <- function(x,q,k=q+1,alpha=0.01) {
  stopifnot(q>0, q<ncol(x), k>q, alpha>0) # sanity checks
  kNNs = find.kNNs(x,k) # should return an n*k matrix of indices
  w = reconstruction.weights(x,kNNs,alpha) # n*n weight  matrix
  coords = coords.from.weights(w,q) # n*q coordinate matrix
  return(coords)
}
```

**Code Example 1:** Locally linear embedding in R. Notice that this top-level function is very simple, and mirrors the math exactly.

from the unconstrained optimum — when it's zero, as here, it means that the constrained optimum is *also* an unconstrained optimum — but we knew that already!)

# 5 Calculation

Let's break this down from the top. The nice thing about doing this is that the over-all function is four lines, one of which is just the return (Example 1).

## 5.1 Finding the Nearest Neighbors

The following approach is straightforward (exploiting an R utility function, `order`), but not recommended for "industrial strength" uses. A *lot* of thought has been given to efficient algorithms for finding nearest neighbors, and this isn't even close to the state of the art. For large $n$, the difference in efficiency would be quite substantial. For the present, however, this will do.

To find the $k$ nearest neighbors of each point, we first need to calculate the distances between all pairs of points. The neighborhoods only depend on these distances, not the actual points themselves. We just need to find the $k$ smallest entries in each row of the distance matrix (Example 2).

Most of the work is done either by `dist`, a built-in function optimized for calculating distance matrices, or by `smallest.by.rows` (Example 3), which we are about to write. The $+1$ and $-1$ in the last two lines come from simplifying that. Instead of `dist`, we could have recycled code from the first lecture, but this really is faster, and more flexible.

`smallest.by.rows` uses the utility function `order`. Given a vector, it returns the *permutation* that puts the vector into increasing order, i.e., its return

```
# Find multiple nearest neighbors in a data frame
# Inputs: n*p matrix of data vectors, number of neighbors to find,
  # optional arguments to dist function
# Calls: smallest.by.rows
# Output: n*k matrix of the indices of nearest neighbors
find.kNNs <- function(x,k,...) {
  x.distances = dist(x,...) # Uses the built-in distance function
  x.distances = as.matrix(x.distances) # need to make it a matrix
  kNNs = smallest.by.rows(x.distances,k+1) # see text for +1
  return(kNNs[,-1]) # see text for -1
}
```

**Code Example 2:** Finding the $k$ nearest neighbors of all the row-vectors in a data frame.

```
# Find the k smallest entries in each row of an array
# Inputs: n*p array, p >= k, number of smallest entries to find
# Output: n*k array of column indices for smallest entries per row
smallest.by.rows <- function(m,k) {
  stopifnot(ncol(m) >= k) # Otherwise "k smallest" is meaningless
  row.orders = t(apply(m,1,order))
  k.smallest = row.orders[,1:k]
  return(k.smallest)
}
```

**Code Example 3:** Finding which columns contain the smallest entries in each row.

is a vector of integers as long as its input.[10] The first line of `smallest.by.rows` applies `order` to each row of the input matrix `m`. The first column of `row.orders` now gives the column number of the smallest entry in each row of `m`; the second column, the second smallest entry, and so forth. By taking the first $k$ columns, we get the set of the smallest entries in each row. `find.kNNs` applies this function to the distance matrix, giving the indices of the closest points. However, every point is closest to itself, so to get $k$ *neighbors*, we need the $k+1$ closest *points*; and we want to discard the first column we get back from `smallest.by.rows`.

Let's check that we're getting sensible results from the parts.

```
> r
     [,1] [,2]
[1,]    7    2
[2,]    3    4
> smallest.by.rows(r,1)
[1] 2 1
> smallest.by.rows(r,2)
     [,1] [,2]
[1,]    2    1
[2,]    1    2
```

Since $7 > 2$ but $3 < 4$, this is correct. Now try a small distance matrix, from the first five points on the spiral:

```
> round(as.matrix(dist(x[1:5,])),2)
     1    2    3    4    5
1 0.00 0.11 0.21 0.32 0.43
2 0.11 0.00 0.11 0.22 0.33
3 0.21 0.11 0.00 0.11 0.22
4 0.32 0.22 0.11 0.00 0.11
5 0.43 0.33 0.22 0.11 0.00
> smallest.by.rows(as.matrix(dist(x[1:5,])),3)
  [,1] [,2] [,3]
1    1    2    3
2    2    1    3
3    3    2    4
4    4    3    5
5    5    4    3
```

Notice that the first column, as asserted above, is saying that every point is closest to itself. But the two nearest neighbors are right.

```
> find.kNNs(x[1:5,],2)
  [,1] [,2]
```

[10] There is a lot of control over ties, but we don't care about ties. See `help(order)`, though, it's a handy function.

```
# Least-squares weights for linear approx. of data from neighbors
# Inputs: n*p matrix of vectors, n*k matrix of neighbor indices,
  # scalar regularization setting
# Calls: local.weights
# Outputs: n*n matrix of weights
reconstruction.weights <- function(x,neighbors,alpha) {
  stopifnot(is.matrix(x),is.matrix(neighbors),alpha>0)
  n=nrow(x)
  stopifnot(nrow(neighbors) == n)
  w = matrix(0,nrow=n,ncol=n)
  for (i in 1:n) {
    i.neighbors = neighbors[i,]
    w[i,i.neighbors] = local.weights(x[i,],x[i.neighbors,],alpha)
  }
  return(w)
}
```

**Code Example 4:** Iterative (and so not really recommended) function to find linear least-squares reconstruction weights.

```
1    2    3
2    1    3
3    2    4
4    3    5
5    4    3
```

Success!

## 5.2 Calculating the Weights

First, the slow iterative way (Example 4). Aside from sanity-checking the inputs, this just creates a square, $n \times n$ weight-matrix $\mathbf{w}$, initially populated with all zeroes, and then fills each line of it by calling a to-be-written function, `local.weights` (Example 5).

For testing, it would really be better to break `local.weights` up into two sub-parts — one which finds the Gram matrix, and another which solves for the weights — but let's just test it altogether this once.

```
> matrix(mapply("*",local.weights(x[1,],x[2:3,],0.01),x[2:3,]),nrow=2)
          [,1]       [,2]
[1,]  2.014934 -0.4084473
[2,] -0.989357  0.3060440
> colSums(matrix(mapply("*",local.weights(x[1,],x[2:3,],0.01),x[2:3,]),nrow=2))
[1]  1.0255769 -0.1024033
> colSums(matrix(mapply("*",local.weights(x[1,],x[2:3,],0.01),x[2:3,]),nrow=2))
+ - x[1,]
```

```
# Calculate local reconstruction weights from vectors
# Inputs: focal vector (1*p matrix), k*p matrix of neighbors,
  # scalar regularization setting
# Outputs: length k vector of weights, summing to 1
local.weights <- function(focal,neighbors,alpha) {
  # basic matrix-shape sanity checks
  stopifnot(nrow(focal)==1,ncol(focal)==ncol(neighbors))
  # Should really sanity-check the rest (is.numeric, etc.)
  k = nrow(neighbors)
  # Center on the focal vector
  neighbors=t(t(neighbors)-focal) # exploits recycling rule, which
    # has a weird preference for columns
  gram = neighbors %*% t(neighbors)
  # Try to solve the problem without regularization
  weights = try(solve(gram,rep(1,k)))
    # The try function tries to evaluate its argument and returns
    # the value if successful; otherwise it returns an error
    # message of class "try-error"
  if (identical(class(weights),"try-error")) {
    # Un-regularized solution failed, try to regularize
      # TODO: look at the error, check if it's something
      # regularization could fix!
    weights = solve(gram+alpha*diag(k),rep(1,k))
  }
  # Enforce the unit-sum constraint
  weights = weights/sum(weights)
  return(weights)
}
```

**Code Example 5:** Find the weights for approximating a vector as a linear combination of the rows of a matrix.

```
[1]  0.0104723155 -0.0005531495
```

The `mapply` function is another of the `lapply` family of utility functions. Just as `sapply` sweeps a function along a vector, `mapply` sweeps a multi-argument function (hence the `m`) along multiple argument vectors, recycling as necessary. Here the function is multiplication, so we're getting the products of the reconstruction weights and the vectors. (I re-organize this into a matrix for comprehensibility.) Then I add up the weighted vectors, getting something that looks reasonably close to `x[1,]`. This is confirmed by actually subtract the latter from the approximation, and seeing that the differences are small for both coordinates.

This didn't use the regularization; let's turn it on and see what happens.

```
> colSums(matrix(mapply("*",local.weights(x[1,],x[2:4,],0.01),x[2:4,]),nrow=3))
+ -x[1,]
Error in drop(.Call("La_dgesv", a, as.matrix(b), tol, PACKAGE = "base")) :
  system is computationally singular: reciprocal condition number = 6.73492e-19
[1]  0.01091407 -0.06487090
```

The error message alerts us that the unregularized attempt to solve for the weights failed, since the determinant of the Gram matrix was as close to zero as makes no difference, hence it's uninvertible. (The error message could be suppressed by adding a `silent=TRUE` option to `try`; see `help(try)`.) However, with just a touch of regularization ($\alpha = 0.01$) we get quite reasonable accuracy.

Let's test our iterative solution. Pick $k = 2$, each row of the weight matrix should have two non-zero entries, which should sum to one. (We might expect some small deviation from 1 due to finite-precision arithmetic.) First, of course, the weights should match what the `local.weights` function says.

```
> x.2NNs <- find.kNNs(x,2)
> x.2NNs[1,]
[1] 2 3
> local.weights(x[1,],x[x.2NNs[1,],],0.01)
[1]  1.9753018 -0.9753018
> wts<-reconstruction.weights(x,x.2NNs,0.01)
> wts[1,1:6]
[1]  0.0000000  1.9753018 -0.9753018  0.0000000  0.0000000  0.0000000
> sum(wts[1,] != 0)
[1] 2
> all(rowSums(wts != 0)==2)
[1] TRUE
> all(rowSums(wts) == 1)
[1] FALSE
> summary(rowSums(wts))
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1       1       1       1       1       1
```

Why does `summary` say that all the rows sum to 1, when directly testing that says otherwise? Because some rows don't *quite* sum to 1, just closer-than-display tolerance to 1.

```
> sum(wts[1,]) == 1
[1] FALSE
> sum(wts[1,])
[1] 1
> sum(wts[1,]) - 1
[1] -1.110223e-16
> summary(rowSums(wts)-1)
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-2.220e-16  0.000e+00  0.000e+00 -1.406e-17  0.000e+00  2.220e-16
```

So the constraint is satisfied to $\pm 2 \cdot 10^{-16}$, which is good enough for all practical purposes. It does, however, mean that we have to be careful about testing the constraint!

```
> all(abs(rowSums(wts)-1) < 1e-7)
[1] TRUE
```

Of course, iteration is usually Not the Way We Do It in R — especially here, where there's no dependence between the rows of the weight matrix.[11] What makes this a bit tricky is that we need to combine information from two matrices — the data frame and the matrix giving the neighborhood of each point. We could try using something like `mapply` or `Map`, but it's cleaner to just write a function to do the calculation for each row (Example 6), and then apply it to the rows.

As always, check the new function:

```
> w.1 = local.weights.from.indices(1,x,x.2NNs,0.01)
> w.1[w.1 != 0]
[1]  1.9753018 -0.9753018
> which(w.1 != 0)
[1] 2 3
```

So (at least for the first row!) it has the right values in the right positions.

Now the final function is simple (Example 7), and passes the check:

```
> wts.2 = reconstruction.weights.2(x,x.2NNs,0.01)
> identical(wts.2,wts)
[1] TRUE
```

[11] Remember what makes loops slow in R is that every time we change an object, we actually create a new copy with the modified values and then destroy the old one. If $n$ is large, then the weight matrix, with $n^2$ entries, is very large, and we are wasting a lot of time creating and destroying big matrices to make small changes.

```
# Get approximation weights from indices of point and neighbors
# Inputs: index of focal point, n*p matrix of vectors, n*k matrix
  # of nearest neighbor indices, scalar regularization setting
# Calls: local.weights
# Output: vector of n reconstruction weights
local.weights.for.index <- function(focal,x,NNs,alpha) {
  n = nrow(x)
  stopifnot(n> 0, 0 < focal, focal <= n, nrow(NNs)==n)
  w = rep(0,n)
  neighbors = NNs[focal,]
  wts = local.weights(x[focal,],x[neighbors,],alpha)
  w[neighbors] = wts
  return(w)
}
```

**Code Example 6:** Finding the weights for the linear approximation of a point given its index, the data-frame, and the matrix of neighbors.

```
# Local linear approximation weights, without iteration
# Inputs: n*p matrix of vectors, n*k matrix of neighbor indices,
  # scalar regularization setting
# Calls: local.weights.for.index
# Outputs: n*n matrix of reconstruction weights
reconstruction.weights.2 <- function(x,neighbors,alpha) {
  # Sanity-checking should go here
  n = nrow(x)
  w = sapply(1:n,local.weights.for.index,x=x,NNs=neighbors,
             alpha=alpha)
  w = t(w) # sapply returns the transpose of the matrix we want
  return(w)
}
```

**Code Example 7:** Non-iterative calculation of the weight matrix.

```
# Find intrinsic coordinates from local linear approximation weights
# Inputs: n*n matrix of weights, number of dimensions q, numerical
  # tolerance for checking the row-sum constraint on the weights
# Output: n*q matrix of new coordinates on the manifold
coords.from.weights <- function(w,q,tol=1e-7) {
  n=nrow(w)
  stopifnot(ncol(w)==n) # Needs to be square
  # Check that the weights are normalized
    # to within tol > 0 to handle round-off error
  stopifnot(all(abs(rowSums(w)-1) < tol))
  # Make the Laplacian
  M = t(diag(n)-w)%*%(diag(n)-w)
    # diag(n) is n*n identity matrix
  soln = eigen(M) # eigenvalues and eigenvectors (here,
    # eigenfunctions), in order of decreasing eigenvalue
  coords = soln$vectors[,((n-q):(n-1))] # bottom eigenfunctions
    # except for the trivial one
  return(coords)
}
```

**Code Example 8:** Getting manifold coordinates from approximation weights by finding eigenfunctions.

## 5.3 Calculating the Coordinates

Having gone through all the eigen-manipulation, this is a straightforward calculation (Example 8).

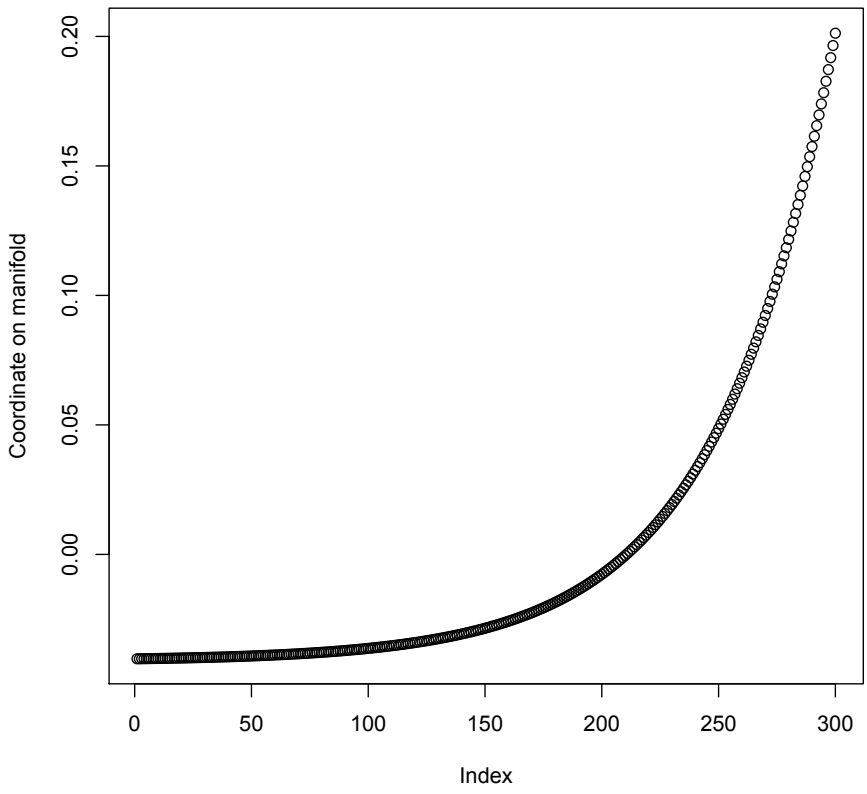
Notice that $\mathbf{w}$ will in general be a very **sparse** matrix — it has only $k$ non-zero entries per row, and typically $k \ll n$. There are special techniques for rapidly solving eigenvalue problems for sparse matrices, which are not being used here — another way in which this is not an industrial-strength version.

Let's try this out: make the coordinate (with $q = 1$), plot it (Figure 5), and check that it really is monotonically increasing, as the figure suggests.

```
> spiral.lle = coords.from.weights(wts,1)
> plot(spiral.lle,ylab="Coordinate on manifold")
> all(diff(spiral.lle) > 0)
[1] TRUE
```

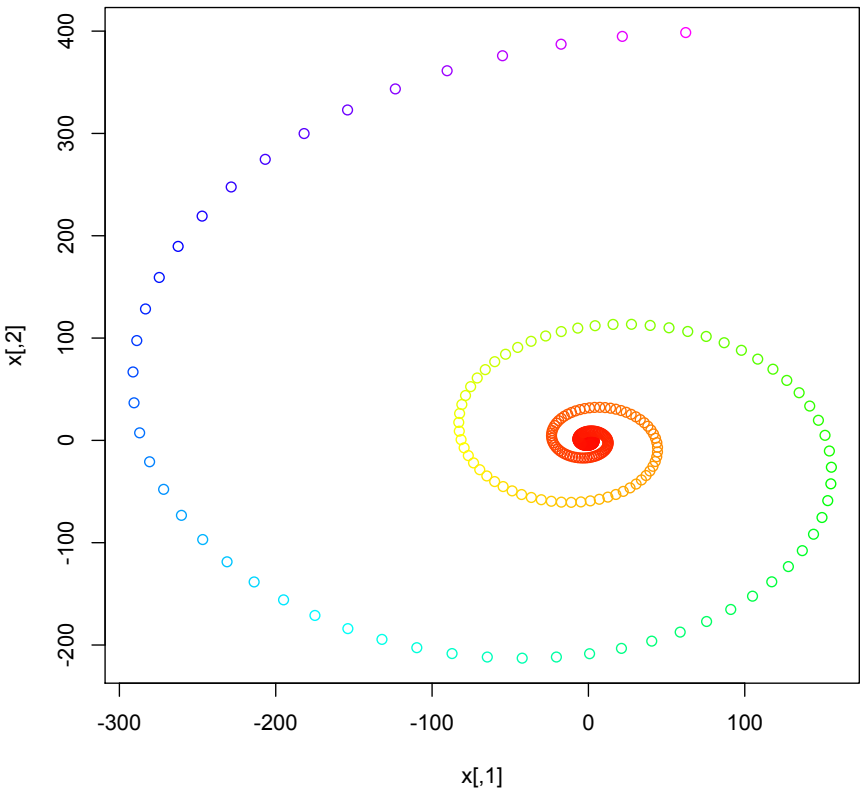
So the coordinate we got through LLE increases along the spiral, just as it should, and we have successfully recovered the underlying structure of the data. To verify this in a more visually pleasing way, Figure 6 plots the original data again, but now with points colored so that their color in the rainbow corresponds to their inferred coordinate on the manifold.

Before celebrating our final victory, test that everything works when we put it together:




```
plot(coords.from.weights(wts,1),ylab="Coordinate on manifold")
```

Figure 5: Coordinate on the manifold estimated by locally-linear embedding for the spiral data. Notice that it increases monotonically along the spiral, as it should.




```
plot(x,col=rainbow(300,end=5/6)[cut(spiral.lle,300,labels=FALSE)])
```

Figure 6: The original spiral data, but with color advancing smoothly along the spectrum according to the intrinsic coordinate found by LLE.

```
> all(lle(x,1,2)==spiral.lle)
[1] TRUE
```

□

# References


Amari, Shun-ichi and Hiroshi Nagaoka (1993/2000). *Methods of Information Geometry*. Providence, Rhode Island: American Mathematical Society. Translated by Daishi Harada. As *Joho Kika no Hoho*, Tokyo: Iwanami Shoten Publishers.

Arnol'd, V. I. (1973). *Ordinary Differential Equations*. Cambridge, Massachusetts: MIT Press. Trans. Richard A. Silverman from *Obyknovennye differentsial'nye Uravneniya*.

Guckenheimer, John and Philip Holmes (1983). *Nonlinear Oscillations, Dynamical Systems and Bifurcations of Vector Fields*. New York: Springer-Verlag.

Kass, Robert E. and Paul W. Vos (1997). *Geometrical Foundations of Asymptotic Inference*. New York: Wiley.

Lawrie, Ian D. (1990). *A Unified Grand Tour of Theoretical Physics*. Bristol, England: Adam Hilger.

Roweis, Sam T. and Laurence K. Saul (2000). "Nonlinear Dimensionality Reduction by Locally Linear Embedding." *Science*, **290**: 2323–2326. doi:10.1126/science.290.5500.2323.

Saul, Lawrence K. and Sam T. Roweis (2003). "Think Globally, Fit Locally: Supervised Learning of Low Dimensional Manifolds." *Journal of Machine Learning Research*, **4**: 119–155. URL `http://jmlr.csail.mit.edu/papers/v4/saul03a.html`.

Schutz, Bernard F. (1980). *Geometrical Methods of Mathematical Physics*. Cambridge, England: Cambridge University Press.

Spivak, Michael (1965). *Calculus on Manifolds: A Modern Approach to Classical Theorems of Advanced Calculus*. Menlo Park, California: Benjamin Cummings.