.Abstract

Geometry and Analysis of Dual Networks on Questionnaires

Jerrod Isaac Ankenman

2014

We investigate databases of binary questionnaire type, which map the Cartesian product of two sets into two distinct values. These databases naturally take the form of matrices, where the rows and columns each represent one of the sets and can be expected to have independent structure and correlations. We define a generative model for data of this type by assuming that the data is generated by independent Bernoulli draws from a probability field which is a function from the Cartesian product into [0,1]. We define a notion of function smoothness with respect to a metric induced by a partition tree on a set, and we further define a basis for the space of these matrices which is the tensor product of Haar-like systems defined on a pair of partition trees on the rows and columns. Then we define a notion of smoothness in two variables with respect to this bi-Haar basis, and assume that the probability field which generates the observed data is smooth with respect to some such dual geometry on the rows and columns. We describe an algorithm for discovering such a geometry which involves building graphs on the rows and on the columns, and constructing partition trees on each based on a metric induced by a random walk on these graphs.. We then iteratively refine the partition trees using an Earth Mover's Distance with respect to the tree metric in the other direction. After discovering the geometry of the sets, we describe an algorithm for reconstructing the underlying probability field subject to the smoothness condition previously defined. We further introduce a novel method of estimating the probability field for out-of-sample columns by adaptively selecting and observing a small subset of the data.

Geometry and Analysis of Dual Networks on Questionnaires

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Jerrod Isaac Ankenman

Dissertation Director: Ronald R. Coifman

May 2014

Contents

Acknowledgements

A doctoral dissertation is not, and is not generally intended to be, the project of a student working alone. But given how unlikely it would have seemed at many times in the past that I would be writing an acknowledgements section on such a dissertation, I feel both especially gratified to have the opportunity to do so and especially grateful to the many individuals who have made this possible.

As his many collaborators can surely attest, Raphy Coifman is a first-rate mathematician. Not only that, he was a first-rate advisor, finding a project that was nicely suited to my skills and allowing me enough freedom to put my own mark on it, while remaining always accessible, supporting, and vastly knowledgeable. Without his guidance none of this would be possible. There's no one else I'd prefer to tell me "No, you're wrong. It's a theorem!"

My officemate and friend Will Leeb has been a great help to me over the past two years, participating in fruitful discussions, and especially in helping to clarify bits of analysis that escaped me the first time. He had many useful suggestions and pointed out many errors in drafts of this dissertation. I'm so glad that I lobbied to have him moved into my office way back when; my graduate school career has been much more rewarding for it.

Of all the people on this list, Bill Chen, my longtime business partner and book co-author, probably had to most to do with my ending up in the field of mathematics. His willingness to completely overlook disparities in experience and credentials and just invite me to work together on math and game theory in poker was really the catalyst for "waking up" my previously languishing abilities, and getting me back on a track to success many years ago.

My wife Michelle definitely had the most to do with my ending up in graduate school. It was she that first encouraged me to look into programs in mathemat-

ics, and after I finished at Columbia, it was she who encouraged me to apply to graduate school and to aim high and apply to places like Yale. Her patience and willingness to sacrifice to support me through this process has been nothing short of amazing.

I would like to thank Peter Jones, Yuval Kluger, and Mauro Maggioni for agreeing to serve as readers for this dissertation.

I also would like to thank all the people who I have worked with in one capacity or another, who helped to make this dissertation possible. Among others, Doug Costa, Oksana Katsuro-Hopkins, and Catherine Bliss all wrote me letters of recommendation along the way, and they seem to have worked. Daniel Beylkin and Roy Lederman both entered the Applied Math program at the same time as I did, and I hope I was able to help them as much as they helped me in getting through courses and preparing for qualifying exams. Karen Kavanaugh has always been more than willing to help with all the details that really keep things on track.

This would also never have been possible without the love and support of my mother and my stepchildren Michael and Abby.

Most of all, this dissertation is dedicated to my daughter Mara, who delights me with her delight at how Daddy is going to be a "doctor of math." Yes, honey, people will come to me with their math problems and I will try to make them better.

A Note on the Interactive Dissertation

This dissertation, while it contains some theoretical results, is fundamentally empirical in nature. As a result, a substantial part of the value of the dissertation resides in code written to experiment with and implement the algorithms it describes. Almost all of this work was done using the excellent open source scientific software stack for Python, consisting of *numpy*[22], *scipy*[16], *matplotlib*[15], *scikit-learn[23], and IPython*[24]. We have not yet reached a point where a dissertation can be submitted online as an interactive, executable software package. However, in that spirit, in lieu of the usual figures and tables, we include several IPython notebooks rendered in the text, which show the results of interactive sessions, including code, plots, and markdown that should be considered part of the text. All these notebooks, as well as almost all the data that supports them (for legal reasons, we cannot release one data set), will be available for download at https://github.com/hgfalling/pyquest. In the spirit of open source software, we encourage the reader to download, experiment with, and possibly extend the software both as a companion to this thesis and as a powerful tool for data analysis.

1 Introduction

Problems involving the collection, analysis, and inference on large sets of high-dimensional Euclidean data are increasingly important in applications. The development of methods for dealing with such datasets has largely focused on setting the data in low-dimensional Euclidean space or on linear methods which struggle with the curse of dimensionality and other difficulties. Many such methods attempt to exploit the global geometry of data points by clustering, kernel methods, or classification by hyperplanes, but these methods often ignore the geometry in the other direction (correlations among the dimensions of observations). In this thesis, we further develop previously published methods for iteratively organizing data using correlations among observations to inform the organization of the dimensions of those observations, and vice versa. We propose a generative model for data of the binary questionnaire type, and propose methods of reconstructing the hidden parameters of that model. We further propose an adaptive method of reconstructing the model for a new person by only observing a selected but highly-reduced subset of the questions.

1.1 Problem setting

Consider the following common problems:

- A questionnaire consisting of m questions, to which n patients answer either yes or no.
- A database of sports match information, where n players or teams face each other in matches, whose outcomes are recorded as a win or loss.
- A database of documents, where n documents are queried for occurrences of m words, with the counts of each recorded.

- A recommendation platform, where n individuals rate a subsample of m movies/books/songs.

Each of these problems shares a common structure and a natural representation as a two-dimensional $m \times n$ matrix, and we will refer to data of this form as being of **questionnaire type**. Particularly, if the answers are restricted to two different options (ie yes/no), we refer to the data as **as being of binary questionnaire type**. Often we will refer to the rows of such a matrix as **questions** and the columns of such a matrix as **people**, and frame the matrix as the results of a questionnaire where n people were interrogated in m different ways. But this formulation is completely general; the method treats the transposed matrix in exactly the same way; both rows and columns are organized identically.

Let the dataset we want to consider be $X = \{x_1, ..., x_n\}$, a set of n observations in $\mathbb{R}^m$. Then the dataset naturally organizes into a $m \times n$ rectangular matrix. Now the standard assumptions of classical statistics, independence and identical distribution, immediately break down because there are significant, potentially non-linear correlations both among the rows and among the columns. Importantly, there is no particular reason to prefer treating one dimension as independent, while the other is treated as dependent. We first describe a generative model for data of binary questionnaire type, and follow that with a thorough description of the methods by which we find that model's parameters.

2 A Generative Model for Questionnaire-type Data

Suppose we are given X, a binary questionnaire type dataset. The answer to each question $X(i,j)$ is either "yes" $(+1)$ or "no" (-1). There are a few questions to be asked. What does it mean to find the "best" organization of rows and columns? What does that organization imply? And if we knew how to find it, what could we do with it?

2.1 Probability Fields

Suppose that we consider the process by which individuals respond to the questions in the questionnaire as a signal to be processed. In one sense, there is no "noise" in the record of the responses, assuming the respondent actually did mark "yes" or "no" in response to each question and those choices were accurately recorded and passed on. If the response of each individual to each question were deterministic, then this would be a completely noiseless setting, and organizing the data by its bidirectional similarity as well as possible would be an end goal. However, we can make the following empirical observation: in the list of questions, there are many pairs or groups which appear to be nearly identical in meaning. Yet it is not uncommon for individuals to answer these questions differently. It may be that there is some subtlety to the precise wordings that causes this effect. But surely any test-taker can relate to situations where the right answer is unclear, and the decision to answer "yes" or "no" is made in some arbitrary way that may not be repeatable. The fact that the answer to a question must be quantized into "yes" or "no" is therefore a form of noise. So instead of modeling the dataset as a true and noiseless capture of the intersection of people and questions, we instead model each question/person pair as a random variable instead.

- We assume that there exists an $m \times n$ matrix F whose elements are on $[0, 1]$. Then X (the observed data) is the result of sampling independent Bernoulli variables with parameters equal to the entries of F; that is:

$$X(i, j) \sim (2)\text{Bernoulli}(F(i, j)) - 1$$

2.2 Smoothness

The above model, though, is insufficient to allow for meaningful description of F. The observed data X exhibits what we might call **quantization noise**, which occurs because the output values are restricted to $\{-1, 1\}$. For example, suppose that under the above model we sought a maximum likelihood estimator F; since there are no constraints on the structure of F, we would find that

$$F(i, j) = \begin{cases} 1 & X(i, j) = +1 \\ 0 & X(i, j) = -1 \end{cases}$$

One useful condition to place on F is that it satisfy some smoothness condition with respect to a geometry on the rows and columns of the matrix. The condition we will enforce was first introduced in [26], and updated by [6], and can be characterized as a **bi-Hölder** condition: that averages of data on rectangles in the matrix have a mixed derivative that is bounded by a power of the sizes of the rectangles. We will define the rectangles and this condition precisely in 4.4, but we include the following brief example of estimating a smooth probability field from sampled binary data when the geometry is known.

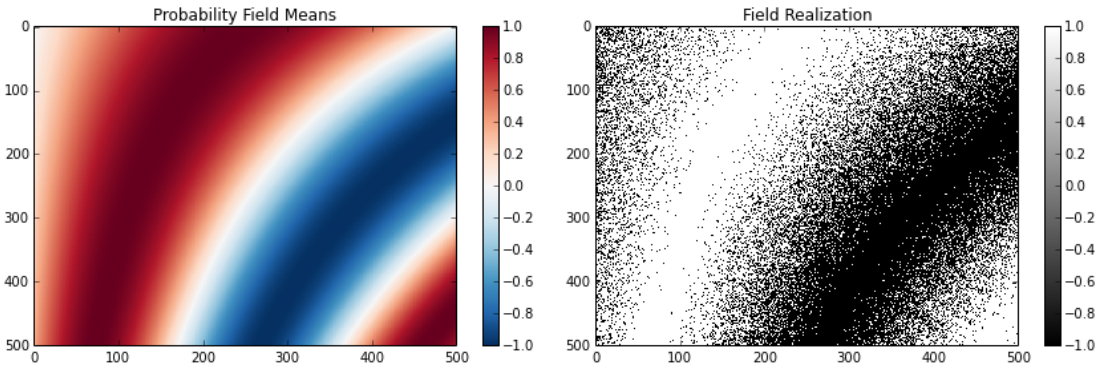
Reconstruction of a Sampled Smooth Function

In this short example, we generate a smooth artificial probability field. We then sample the field as independent Bernoulli variables, and then recover the underlying probability field from the sampled data, using binary trees built on the known geometry.

In [1]:
```
from imports import *
```

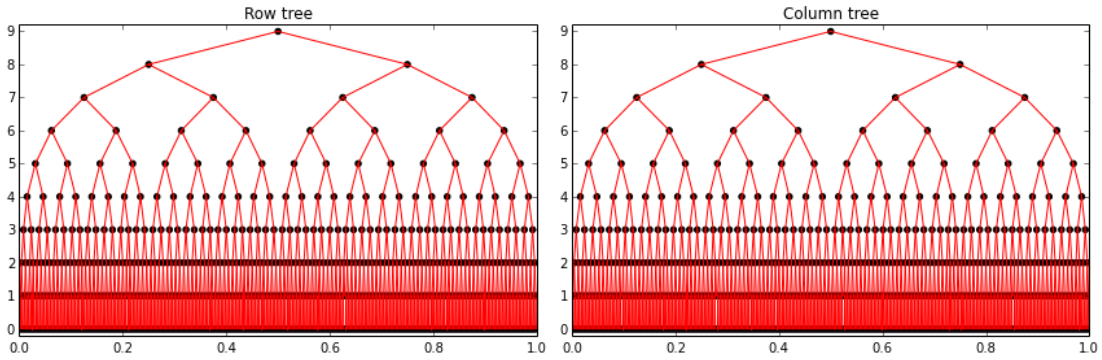
We define the sample probability field to be a 500×500 matrix P. We let i vary uniformly on $\left[0, \frac{\pi}{4}\right]$ and j vary uniformly on $[0, 2\pi]$ over the rows and columns of the matrix respectively. Then we let the entries of the matrix be $P(i,j) = \frac{1}{2}\left(1 + \sin\left(\frac{i+j+2ij}{2}\right)\right)$. We then sample from the field, taking successes to be $+1$ and failures to be -1.

In [2]:
```
n_rows,n_columns = 500,500
means_matrix = np.zeros([n_rows,n_columns])
for i in xrange(n_rows):
    for j in xrange(n_columns):
        ii = i*0.25*np.pi/n_rows
        jj = j*2.0*np.pi/n_columns
        means_matrix[i,j] = np.sin((ii+jj+2*ii*jj)/2.0)*1.0

np.random.seed(20140304)
pf = artificial_data.ProbabilityField(means_matrix/2.0+0.5)
orig_data = pf.realize()

fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
cplot(means_matrix,colorbar=True,title="Probability Field Means")
fig.add_subplot(122)
bwplot2(orig_data,colorbar=True,title="Field Realization")
plt.tight_layout()
plt.show()
```



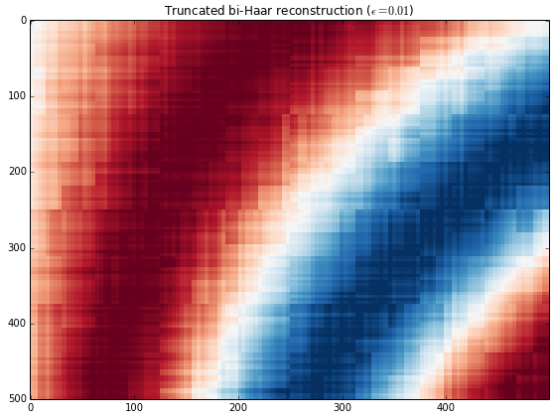
Next we define binary trees on the rows and columns.

In [3]:
```
rt = bin_tree_build.random_binary_tree(n_rows,1.0)
ct = bin_tree_build.random_binary_tree(n_columns,1.0)

fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
plot_tree(rt,title="Row tree")
fig.add_subplot(122)
plot_tree(ct,title="Column tree")
plt.tight_layout()
plt.show()
```



We then reconstruct the probability field by expressing the sampled data in the bi-Haar basis implied by these trees, and truncating the coefficients corresponding to folders smaller than ϵ to 0.

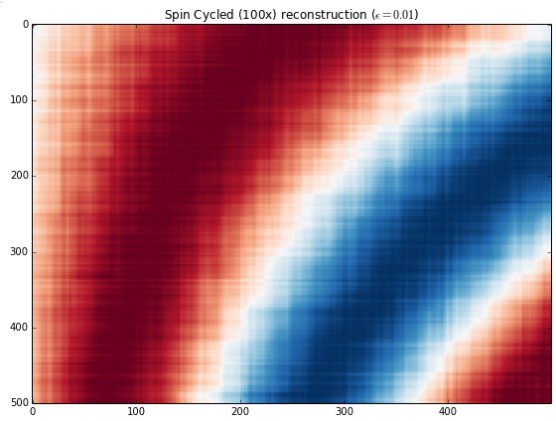
In [4]:
```
tr = tree_recon.recon_2d_haar_folder_size(orig_data,rt,ct,0.01)
tree_recon.threshold_recon(tr,-1.0,1.0)
fig = plt.figure(figsize=(8,6))
fig.add_subplot(111)
cplot(tr,title="Truncated bi-Haar reconstruction ($\epsilon = 0.01$)")
plt.tight_layout()
plt.show()
```



We now consider the result of "spin cycling" the reconstruction to remove artifacts. We introduce a slight element of randomness to the tree construction, and generate ten pairs of trees. Then we cross-match the row and column trees with each other, generating 100 pairs of trees, each of which has a slightly perturbed basis. Then we reconstruct as before, and average over the 100 tree-pairs.

In [5]:
```
row_trees, col_trees = [],[]
spun_recon = np.zeros(orig_data.shape)
for i in xrange(10):
    row_trees.append(bin_tree_build.random_binary_tree(n_rows,1.2+0.2*i))
    col_trees.append(bin_tree_build.random_binary_tree(n_columns,1.2+0.2*i))
for rt in row_trees:
    for ct in col_trees:
        tr = tree_recon.recon_2d_haar_folder_size(orig_data,rt,ct,0.01)
        tree_recon.threshold_recon(tr,-1.0,1.0)
        spun_recon += tr
spun_recon /= 100.0
```

In [6]:
```
fig = plt.figure(figsize=(8,6))
fig.add_subplot(111)
cplot(spun_recon,title="Spin Cycled (100x) reconstruction ($\epsilon = 0.01$)")
plt.tight_layout()
plt.show()
```



Now in this toy example, the reconstruction method here is not particularly efficient. Because the geometry of the image is uniform in each direction, standard signal processing techniques could be applied with good effect. We present this only as a short example of the process of recovering a smooth function from a given geometry. In other settings, the geometry of the rows or columns ("people") might be of higher dimension, or oscillating at different rates. In these cases, these reconstruction techniques continue to work just as well.

The point here is that our model assumes the following:

- There exists some permutation of the rows and some permutation of the columns such that if we consider F on the permutation, F is smooth.

Suppose we are given a potentially shuffled matrix of binary data where we can reasonably make these assumptions. Our goals are to first discover (from the empirical data) the dual geometry on the rows and columns with respect to which F is smooth. We can then use the geometry we learn, in conjunction with the empirical data, to estimate F.

3 Diffusion Maps

Suppose that we have X, an $m \times n$ matrix, with both m and n of reasonable size, and we are seeking a geometry on the rows and columns. One natural thing to do is to use the geometry of the original Euclidean space from which the data came. However, this will fail as a general method of defining a geometry, because of the well-known result that the discriminatory power of distances falls off rapidly as the dimension of the data increases. Most common approaches apply some type of dimensionality reduction to the problem. Additionally, in settings that contain significant noise (such as the binary samples from a probability field), we are especially interested in metrics robust to noise which heavily weight local geometry and deemphasize the preservation of large distances. For these purposes, we will employ diffusion maps as introduced by [5, 18].

3.1 Diffusion geometry and the diffusion distance

Let Ω be a set of n points. Then we define an **affinity** $k : \Omega \times \Omega \to \mathbb{R}$ as a kernel function with the following three properties:

- $k(x,y) = k(y,x)$ (symmetric)
- $k(x,y) \geq 0 \ \forall x, y$ (non-negative)
- $\sum_x \sum_y k(x,y) f(x) f(y) \geq 0$ (positive semidefinite)

Given any function of this type, we can immediately construct a graph G, where the nodes are the n points, and edges of weight $k(i,j)$ link points $i, j \in \Omega$. Now let K be an $n \times n$ matrix and let $K_{ij} = k(i,j)$. Now we define a random walk on the vertices of G as follows: Let $\omega(x)$ be sum of the row x in K and $\omega(y)$ be the sum of column y in K; that is,

$$\omega(x) = \sum_y k(x,y)$$

Let D be the diagonal matrix with entries $D(x,x) = \omega(x)$. Then $M = D^{-1}K$ is a Markov matrix which can be thought of as defining a random walk on the graph G, where $M(x,y)$ is the probability of transitioning in one time step from point x to point y. Likewise $(M^t)(x,y)$ is the probability of transitioning from point x to point y in t time steps. So now we can use this random walk as the basis for a distance between points. We define the **diffusion distance** between points x and y at a particular time t as:

$$D_t^2(x,y) = \sum_{z=1}^{n} \frac{1}{\pi(z)} \left(\left(M^t\right)(x,z) - \left(M^t\right)(y,z) \right)^2$$

where π is the stationary distribution of the Markov chain. This distance is the weighted Euclidean distance between the distributions induced by the random walk of t steps on x and y . It can be characterized as measuring the overall rate of connectivity between points of the data set. It will be small if the random walk contains many paths of length less than t between x and y, while it will be large if the number of such connections is small.

This diffusion distance is often signficantly more robust to noise than, say, the Euclidean distance in the original space. The reason for this robustness is that the effect of spurious or incoherent connections between points are dampened. Suppose point x and point y are strongly connected in the graph G, but this is in truth a product of some kind of noise and the points x and y are unrelated. When we look at the diffusion distance, we would see that point x diffuses partly to point y, but mostly to its neighbors who are not connected to y. Likewise y diffuses partly to x but mostly to its neighbors who are not connected to x. This continues as the diffusion time increases; essentially there is only one path between x and y, because the neighbors of x are far from the neighbors of y. So diffusion distance is a more **coherent** distance on the graph; the diffusion distance does not only consider the strength of the direct link between x and y, but the strength of all the paths from x to y.

3.2 The diffusion embedding

$M = D^{-1}K$ is not necessarily symmetric, but it is easily shown that it is similar to a symmetric matrix. Since K is symmetric by the symmetry of the kernel function and D is a non-singular diagonal matrix, we have that $D^{-\frac{1}{2}}KD^{-\frac{1}{2}}$ is symmetric. Then $M = D^{-1}K = D^{-\frac{1}{2}}\left(D^{-\frac{1}{2}}KD^{-\frac{1}{2}}\right)D^{\frac{1}{2}}$. Hence M and $D^{-\frac{1}{2}}KD^{-\frac{1}{2}}$ have the same eigenvalues and the eigenvectors of M can be readily obtained by a diagonal transformation of the eigenvectors of $D^{-\frac{1}{2}}KD^{-\frac{1}{2}}$.

Now take$\{\lambda_1, ..., \lambda_n\}$ to be the eigenvalues and $\{\psi_1, ..., \psi_n\}$ the corresponding eigenvectors of M. We define the **diffusion embedding** $\varphi_t : \Omega \to \mathbb{R}^n$ by

$$\varphi_t(x) = \left\{\lambda_1^t\psi_1(x), ..., \lambda_n^t\psi_n(x)\right\}$$

φ_t maps Ω into $\mathbb{R}^n$ but with point locations given by the diffusion distance

metric instead of the Euclidean metric. Now notice here that the eigenvalues λ_i are the eigenvalues of a Markov process; hence we have immediately that $|\lambda_i| \leq 1$ for all i, and further if the graph G is connected, $\lambda_1 = 1$ and the other eigenvalues decay in absolute value from there. Truncating φ_t by taking the first k eigenvalues and eigenvectors leads to a k-dimensional embedding within the original space.

Recall that the diffusion distance at time t was defined as:

$$D_t^2(x,y) = \sum_{z=1}^{n} \frac{1}{\pi(z)} \left(\left(M^t\right)(x,z) - \left(M^t\right)(y,z) \right)^2$$

It can be shown (see [5, 18]) that:

$$D_t^2(x,y) = \sum_{j \geq 0} \lambda_j^t \left\| \psi_j(x) - \psi_j(y) \right\|^2$$

So the Euclidean distance between the coordinates of two points in the diffusion map at time t is exactly the diffusion distance between those two points at time t. For more on the theory of diffusion maps and examples, consult [5, 7, 18, 21]. In our work, we will use diffusion maps for three main purposes:

1. To generate diffusion distances used in creation of initial and flexible trees, as discussed in 4.1.2.

2. As part of the process of generating eigenvector-cut binary trees, as discussed in section 4.1.1.

3. For visualization purposes after running the iterated questionnaire process discussed in section 5.3.

3.3 Affinity

In the previous sections, we described a method for utilizing diffusion maps to embed high-dimensional data into a low-dimensional manifold in the ambient space. This technique was based on building a graph structure on the data based on a given affinity between points. But different choices of affinity can lead to quite different results, and the choice of affinities will normally be driven by the characteristics of the data. For example, suppose that the data is of binary questionnaire type, like a psychological questionnaire where a broad population answers a long panel of questions. Then we can characterize a person's responses as a function supported on the set of questions. More colloquially, a person's responses form an overall response profile. Then the affinity we construct between two people is an attempt to characterize the similarity of the response profiles to the panel of questions.

A commonly used kernel is based on a metric:

$$k(i,j) = e^{-\frac{\|x_i - x_j\|^\alpha}{\epsilon}}$$

where α is some prespecified power and the parameter ϵ is data-dependent and causes the affinity to decay rapidly as the distance between points grows, thus limiting the affinity to (relatively) near neighbors. This helps to mitigate the difficulties in high dimension, because we ignore the problems in discrimination among objects that are far away; only local distances are preserved. Euclidean distance with a Gaussian($\alpha = 2$) is popular here, although other distances may also be utilized. In 5.2, we will suggest that an affinity based on a distance equivalent to Earth Mover's Distance will be useful.

[20]proposed a characterization of the affinity between two points i and j by making the assumption that there are some number of (hidden) equivalence

classes, around which the data is sampled with some unknown noise. Then the affinity between i and j might be the probability that i and j came from the same equivalence class. Other kernels not based on distances are likewise possible. For many data sets, affinities based on the inner product between two data points, such as correlation or cosine similarity, are natural, with a threshold added to set negative values to zero.

In any event, any of these might make useful affinities on appropriate datasets. The point is that the affinity is a proxy for the **global** closeness of two points – that is, people i and j are close (have high affinity) if their response profile against the entire panel of questions is similar in whatever metric we have chosen.

For questionnaire-type data, we often use a normalized correlation-like function, thresholded cosine similarity. Let X_i and X_j be the vectors of data associated with points i and j and let $t \geq 0$ is a threshold parameter discussed below. Then let

$$c_{ij} = \frac{\langle X_i, X_j \rangle}{\|X_i\| \, \|X_j\|}$$

The affinity would then be

$$A(i,j) = \begin{cases} c_{ij} & c_{ij} \geq t \\ 0 & c_{ij} < t \end{cases}$$

Unthresholded, this produces affinities between -1 (perfect anti-correlation between answers) and +1 (perfect correlation). In fact, when the data are coded as +1 (yes) and -1 (no), then this function is simply the number of matching answers minus the number of non-matching answers, divided by the total number of questions. We then want to at least threshold at 0, since one of the requirements on affinities is that they be non-negative.

It is often helpful to even further sparsify the graph by removing edges corre-

sponding to affinities below some threshold. With a carefully chosen threshold, this has the effect of helping to denoise the graph, from the assumption that weak affinities are more likely to be the product of noise than they are to represent something true about the dataset. A threshold chosen too low (or no threshold at all) may result in a graph which is highly connected with little differentiation between clusters. On the other hand, a high threshold will result in a disconnected graph with many components, a situation that is undesirable for the diffusion map, as mass will then be unable to diffuse from one component to a different one. In general, choosing the proper threshold is somewhat data-specific; one possible idea is to slowly increase the threshold until the graph becomes disconnected. Calculating this is a simple matter of monitoring the value of the second eigenvalue of the Markov matrix; as long as it is 1, the graph is disconnected. At the point when the graph does become connected, we might drop the threshold a little so as not to force the connectedness of the graph to rely on one single edge. We may also consider thresholding determined locally, based on some number of neighbors with the highest affinity instead of a global threshold. Again we must be careful to avoid disconnecting the graph. Any thresholding technique can be applied both to the affinity itself or to the Markov matrix – in the former case, edges are removed because of absolute weakness, while in the latter case edges are removed because they are relatively weak compared to other edges attached to a node.

4 Partition Trees

A key tool we will use in our data organization methods is the **partition tree**, defined in the next section. Partition trees serve three main purposes in our overall process. First, they serve as an encoding of similarity between rows or

columns. In our process for finding dual geometry–that is, a geometry which organizes the rows by exploiting the organization of the columns–we use the partition tree on the columns to define subsets of similar questions to refine our local geometry. Second, they serve as scaffolding for the definition of a basis for the space of matrices that we will use to estimate the probability field underlying the dataset. Third, they define a notion of smoothness.

4.1 Construction of partition trees

The concept of **level** is important to our method, and so we add some additional conditions for convenience to the standard definition of a tree as follows:

Define a **finite partition tree** T on a set of elements Ω as follows. T consists of a finite sequence of **partitions** P_l of Ω, with the following properties:

- $P_0 = \{\Omega\}$. (The root partition consists of all the elements in Ω).
- For $l > 0$, $P_l = \{p_1, p_2, ..., p_k\}$ where $p_i \subseteq \Omega$, the p_i are disjoint and nonempty, and $\bigcup_{i=1}^{k} p_i = \Omega$. (On each level, Ω is partitioned into disjoint subsets).
- For $l > 0$, if $s \in P_l$, $s \subseteq p$ for some $p \in P_{l-1}$. (Each set in a lower level partition is a subset of a unique set at every higher level).
- There exists a level k such that P_k consists of all the singleton elements of Ω.
- No two P_l are identical; we can simply elide the duplicate copies.

If we then identify each set in a partition with a vertex and call the set of all vertices V, and draw a directed edge $(p, q) \in E$ between $p \in P_{l-1}$ and $q \in P_l$ if $q \subseteq p$, then the resulting graph $G(E, V)$ is a partition tree. Under this definition, we introduce the following notations:

- If there is an edge (p, q), we call p the **parent** of q, and q the **child** of p.
- Each node except the root has a unique parent, which we also denote by $p = \overline{q}$.
- We denote the set of children of node p by $\underline{p}$, which could be $\emptyset$ (if $p \in P_k$), or any partition of p.
- We will sometimes want to denote **all** the ancestors of q, including the parent, the parent of the parent, and so on up to the root. We denote this set as $\overline{\overline{q}}$.
- Likewise we denote the recursive set of all children of p as $\underline{\underline{p}}$.

We use the notation $\#I$ for the number of elements in the folder I and $|I|$ to represent the fraction of the total set Ω which are contained in I. Hence $|I| = \frac{\#I}{\#\Omega}$.

At each level, each data element is associated with a particular node (we will often refer to these as **folders**), a feature which naturally leads to understanding the tree as enabling multiscale analysis. We further define a **binary tree** as a tree where each parent node has exactly two children unless it has only one element. Additionally, a **dyadic tree** on 2^j points is a tree under which each parent has exactly two children of equal size. Thus there are $j + 1$ levels and the tree has $2^{j+1} - 1$ nodes.

Of course, there are many possible trees of this type, and most of them are essentially useless for purposes of analysis. However, we can rely on previous theoretical results as well as on empirical performance to develop a notion of a "good" tree. Consider the **tree metric** between two elements of Ω defined as

follows:

$$D_{tree}(x,y) = \begin{cases} \min\{|p| : x,y \in p, p \in T\} & x \neq y \\ 0 & x = y \end{cases}$$

So the distance between two data points in this metric is equal to the size of the smallest folder which contains both points. We would like to find and construct trees such that functions on the data are smooth in this tree metric; that is, such that points which are close together in the tree take on function values that are close together. In the next two sections, we suggest methods for building trees that take special advantage of the features of diffusion maps.

4.1.1 Top-down approaches

One possible way of generating a tree is to apply a **top-down** approach: take the entire set S of data, break it into subclusters, then break each of those subclusters into pieces, until we reach the bottom level of the tree. The following algorithm for generating binary trees is inspired by Ravi Kannan's work on clustering in graphs [17].

We begin with a root node, containing the set of points to be clustered. We calculate some affinity on the points, and calculate the diffusion map induced by this affinity. We take ψ, the non-trivial eigenvector corresponding to the largest eigenvalue of the matrix that is not 1. Now each point x has a corresponding coordinate $\psi(x)$ in this eigenvector. Break the elements of the root node into two groups by separating them into a group consisting of those points where $\psi(x) > 0$ and those where $\psi(x) \leq 0$. This break defines the next partition level of the tree. Then for each node in the next level of the tree, repeat the process on the submatrix containing only the elements in that node, breaking it into subnodes as well. This process terminates when the number of subnodes drops

below some threshold. This algorithm can be modified to produce binary trees with different features as desired:

- Break each node at the median eigenvector coordinate to generate **dyadic** binary trees.
- Define a **balance constant** which limits the ratio of size between the two pieces of a node break. If we were to break a node into two pieces at zero, but this would cause the ratio of the size of the two pieces to exceed the balance constant, then we break instead at the closest value to zero such that the ratio does satisfy the balance constant.
- Define a **randomization constant** to produce trees with randomized borders: specify a balance constant as before, but when breaking each node, let the break point vary uniformly among acceptable ratios of folder sizes. So for example if there are 200 points in a particular node, then the dyadic split would be to split into groups of 100. A balance constant of 1.222 would allow for breaks as unbalanced as 105/95. So we could choose the breakpoint uniformly randomly between the 95th and 105th elements. This is especially useful for "spin cycling" (see 6.4).

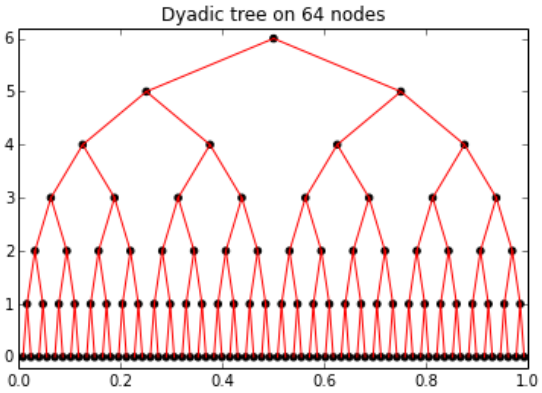
This method does have certain defects, however. The most important one is that it produces a binary tree, which may or may not accurately reflect the structure of the underlying data. It is true that any organization can be approximated by a binary tree by simply introducing additional splits; however, the strong notion of level incorporated into our tree-building process makes this especially distortionary for the methods we employ.

Building Binary Trees

In [1]:
```
from imports import *
np.random.seed(20090403)
```

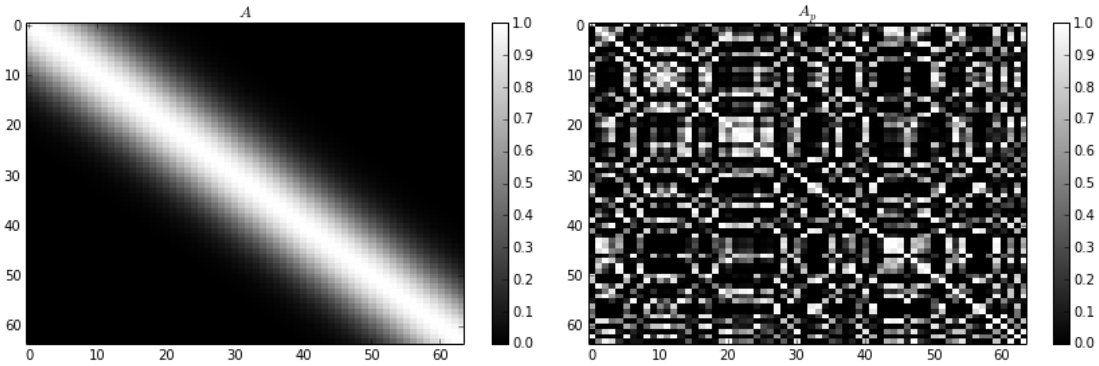
We first consider the dyadic binary tree on 64 nodes:

In [2]:
```
dyadic_64 = tree.dyadic_tree(6)
plot_tree(dyadic_64,title="Dyadic tree on 64 nodes")
```



Let each point of this tree be at a location x_i equal to its index (so 0 through 63). We generate a toy affinity for demonstration purposes by taking $A(i,j) \sim \exp\left(-\frac{|x_i - x_j|^2}{100}\right)$. We further consider a random permutation of the points, A_p, and plot the strength of the affinities A and A_p (white is stronger affinity).

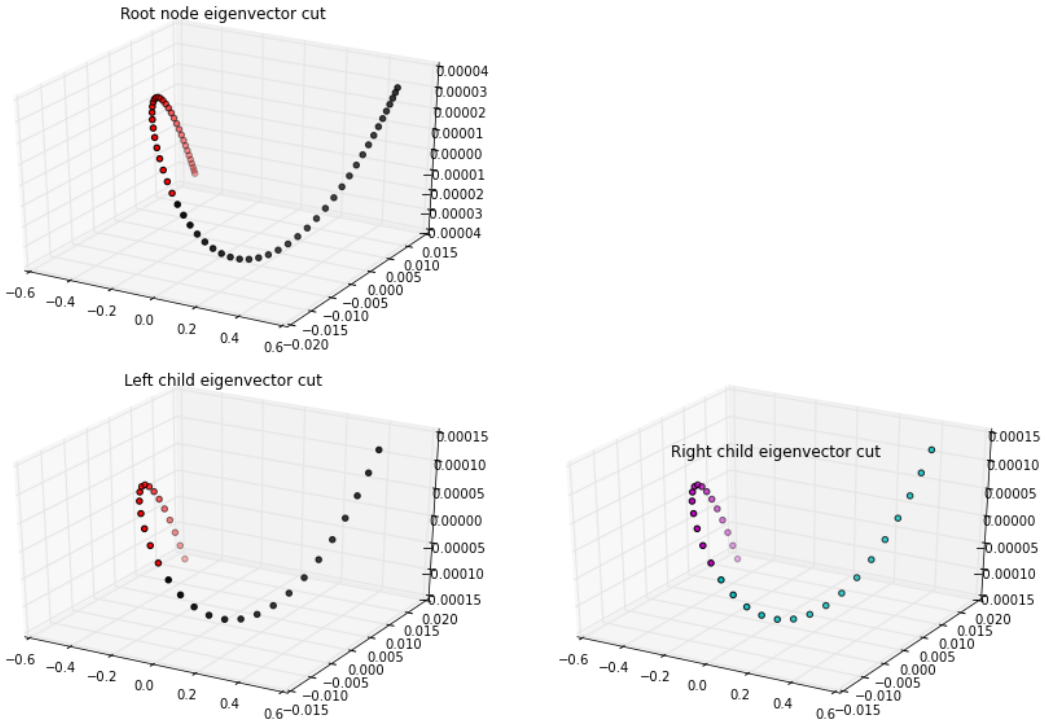
In [3]:
```
A = np.zeros([64,64])
for i in xrange(64):
    for j in xrange(64):
        d_ij = (i-j)
        A[i,j] = np.exp(-(d_ij**2.0/100.0))
row_order = np.random.rand(64).argsort()
A_p = A[row_order,:][:,row_order]
fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
bwplot(A,colorbar=True,title="$A$")
fig.add_subplot(122)
bwplot(A_p,colorbar=True,title="$A_p$")
plt.tight_layout()
plt.show()
```



Eigenvector Cuts

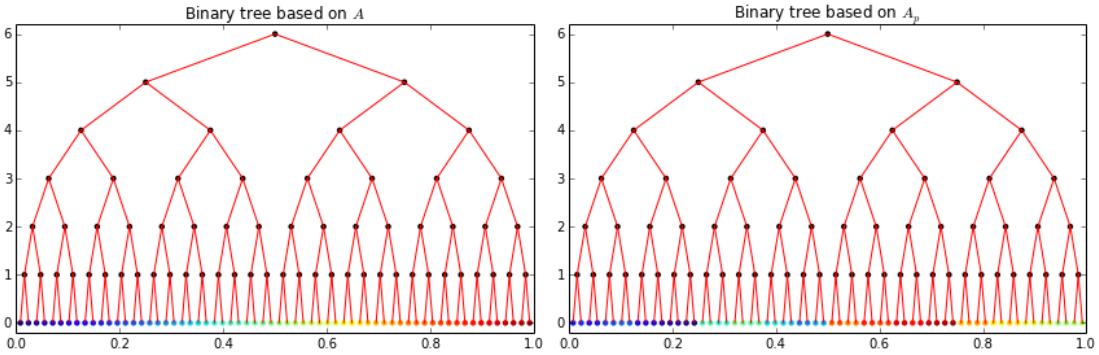
We next construct binary trees on A and A_p. To provide a sense of how the binary cutting process works, we look at the cuts made on the diffusion embedding at diffusion time 1 for the largest few nodes in the tree built on A. The nodes are colored by the split at that level.

In [4]:
```
bt1 = bin_tree_build.bin_tree_build(A,"r_dyadic",1.0)
bt2 = bin_tree_build.bin_tree_build(A_p,"r_dyadic",1.0)
lchild = bt1.children[0]
rchild = bt1.children[1]

fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(221,projection="3d")
ax2 = fig.add_subplot(223,projection="3d")
ax3 = fig.add_subplot(224,projection="3d")

#compute the top few eigenvectors.
vecs,vals = markov.markov_eigs(A,4)
partition = bt1.level_partition(2)
plot_embedding(vecs,vals,partition=partition,ax=ax1)
vecs,vals = markov.markov_eigs(A[lchild.elements,:][:,lchild.elements],4)
partition = np.array(bt1.level_partition(3))[lchild.elements]
plot_embedding(vecs,vals,partition=partition,ax=ax2)
vecs,vals = markov.markov_eigs(A[rchild.elements,:][:,rchild.elements],4)
partition = np.array(bt1.level_partition(3))[rchild.elements]
plot_embedding(vecs,vals,partition=partition,ax=ax3)

ax1.set_title("Root node eigenvector cut")
ax2.set_title("Left child eigenvector cut")
ax3.set_title("Right child eigenvector cut")

plt.tight_layout()
plt.show()
```



Next we display the trees. We color the leaf nodes on a gradient from 0 to 63 (from A) to show the organization of leaves present in the binary trees.

In [5]:
```
leafcolors = np.arange(-1.0,1.0,2.0/64.0)
fig=plt.figure(figsize=(12,4))
fig.add_subplot(121)
plot_tree(bt1,leafcolors=leafcolors,title="Binary tree based on $A$")
fig.add_subplot(122)
plot_tree(bt2,leafcolors=leafcolors[row_order],
          title="Binary tree based on $A_p$")
plt.tight_layout()
plt.show()
```



Notice that in the tree built on A_p, the precise gradient is lost. However, with regard to the tree metric, these trees are exactly equivalent, and can be rearranged to match the original tree without any loss of information. In the tree metric, once a split occurs, the internal organization of one of the child nodes is completely irrelevant to nodes outside that child. This can be seen by matching the distance matrix from one tree to the other (after undoing the permutation).

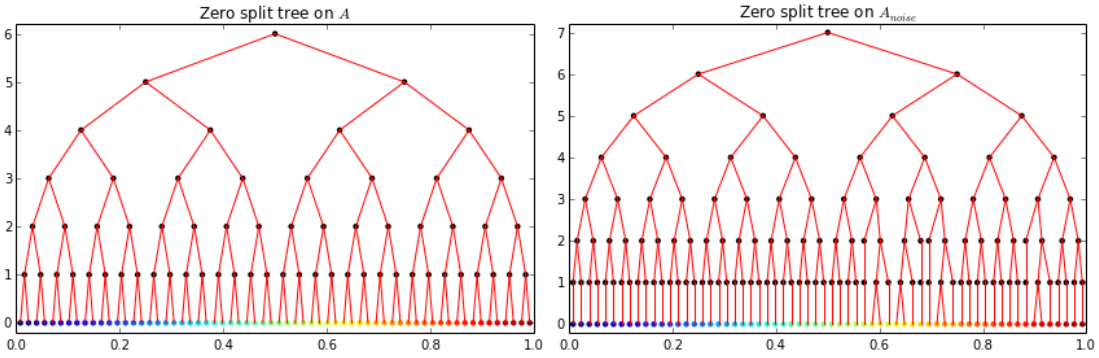
In [6]:
```
inverse_row_order = row_order.argsort()
tree_distances = np.zeros(A.shape)
tree_distances_p = np.zeros(A.shape)
for i in xrange(64):
    for j in xrange(64):
        tree_distances[i,j] = bt1.tree_distance(i,j)
        tree_distances_p[i,j] = bt2.tree_distance(inverse_row_order[i],
                                                  inverse_row_order[j])
np.allclose(tree_distances,tree_distances_p)
```

Out [6]: True

Perturbing the Trees For Spin Cycling

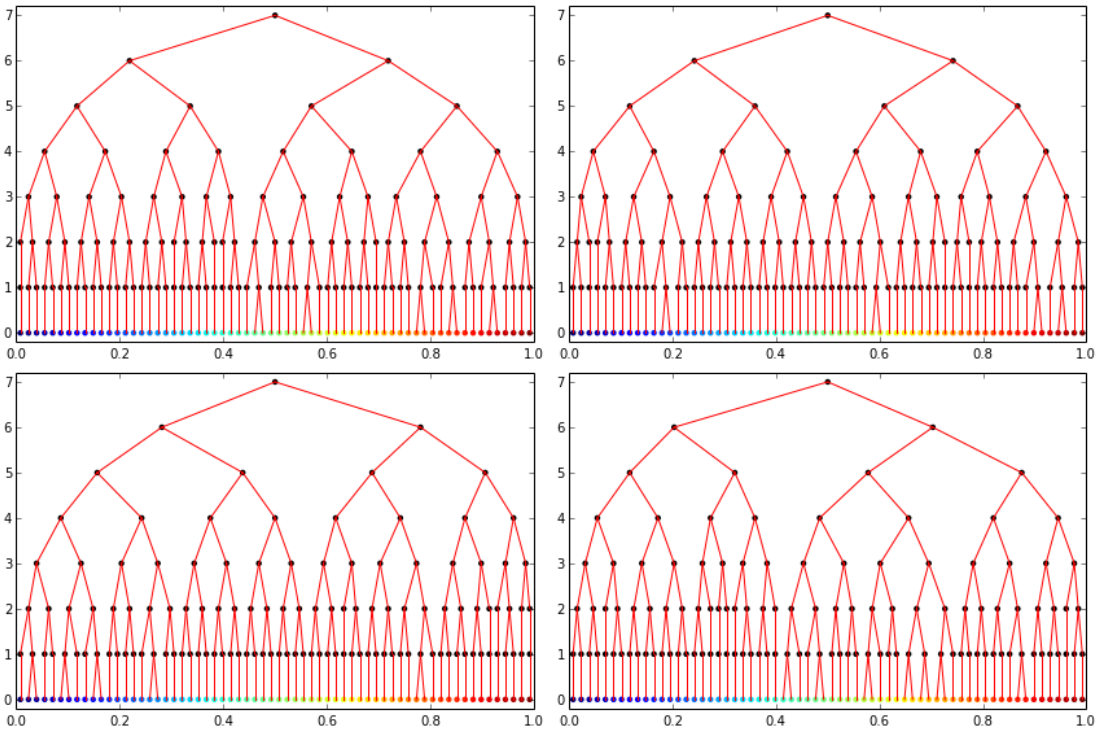
We next demonstrate the other trees discussed in this section. In the dyadic case, the binary splits occur at the median. We can also split the node into two groups at the point where the eigenvector coordinates change sign. Because of the neat symmetry of the original data, splitting A at zero reproduces the original tree exactly. Hence we introduce a slightly noisy version of A by $A_{noise} = (0.9)A + (0.1)U$, where the entries of U are uniform on $[0, 1]$. We then generate the zero-split tree on A_{noise}.

In [7]:
```
bt3 = bin_tree_build.bin_tree_build(A,"zero")
A_noise = A*0.90 + np.random.rand(64,64)*0.1
bt4 = bin_tree_build.bin_tree_build(A_noise,"zero")
fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
plot_tree(bt3,leafcolors=leafcolors,title='Zero split on $A$')
fig.add_subplot(122)
plot_tree(bt4,leafcolors=leafcolors,title='Zero split on $A_{noise}$')
plt.tight_layout()
plt.show()
```



This method (see the tree on the right) introduces a perturbation on the trees that can be desirable for cleaning up artifacts and smoothing boundaries. Finally, we can produce more of this slight randomization by building random dyadic trees. We choose a balance constant (here 1.5), and we can generate many trees which are slight perturbations of each other, but which preserve the overall structure quite well:

In [8]:
```
fig = plt.figure(figsize=(12,8))
for i in xrange(4):
    fig.add_subplot(2,2,i+1)
    bt = bin_tree_build.bin_tree_build(A,"r_dyadic",1.5)
    plot_tree(bt,leafcolors=leafcolors)
plt.tight_layout()
plt.show()
```



4.1.2 Bottom-up approaches

On the contrary, a **bottom-up** approach to building a tree builds the smallest, lowest-level folders first, and then proceeds by joining those folders into larger folders at higher levels, until at the top level of the tree, all the folders are joined at the root. Many algorithms exist for clustering of this type, notably the algorithm of Ward [28], as well as various other methods based on different metrics such as [25, 27]. For our purposes, however, we would like to build trees that have the following properties:

- The tree structure has a strong sense of level. Since we want the tree structure to induce a metric on the data, we want there to be relatively few levels (as compared with common clustering methods, in which there may be a level for each point), and we want the level at which folders are joined to be meaningful across the entire dataset.

- The tree structure is logically multiscale and follows the structure of the data. For example, consider a dataset that consists of three distinct balls of points equidistant from each other. If we have a strict binary tree, we would either cut one of the balls in two, which damages the tree metric, or take two of the balls against the other, and split the two balls at a higher level. But this damages the notion of level. Instead we would like to have the tree structure be simply organized into three folders.

- We would like to take advantage of the diffusion at different scales. Since we have diffusion distances for at all times t , it is logical to use the diffusion distance for higher t to cluster folders that are farther apart, because their relative distance is only well-realized after some time has passed in the diffusion.

We propose the following algorithm for building what we will call **flexible trees**, which attempt to provide all these properties.

1. At level 1, each point is in its own folder. Let the number of points to be turned into a tree be n. Then there are n folders, each containing only one point.

2. Define a penalty constant ϵ which will constrain the size of clusters as described below.

3. Define (using some other process) an affinity $A_{n\times n}$ on the n points. Calculate the diffusion map at time $t = 1$ on this affinity and the diffusion distances between all pairs of points.

4. Let p be the median distance between two points.

5. Find $d_t(i, j)$, the minimum distance between a point i, a singleton (at this level), and a point j which may or may not already be in a cluster.

   (a) If $d_t(i, j) > \frac{p}{\epsilon}$ and at least one join has already occurred at this level, then there are no more joins at this level. Continue with step 7.

   (b) If $d_t(i, j) < \frac{p}{\epsilon}$, or no joins have occurred at this level, then attempt to join points i and j as follows.

      i. If i and j are both singletons at this level, then form a new cluster containing both of them.

      ii. If j is already in a cluster, then join i to that cluster if $d_t(i, j) < \frac{p}{\epsilon}2^{(\#C_j-1)}$ where C_j is the cluster containing j. So for each element beyond the first in the cluster containing j, we divide the maximum allowable distance for joining the two by 2.

6. Repeat step 5 until there are no more joins at the current level. Note that two clusters containing two or more points are never joined in this process; any joins between multi-element clusters will occur at the next level.

7. The clusters formed by the repeated application of Step 5 are the folders at the next higher level of the tree.

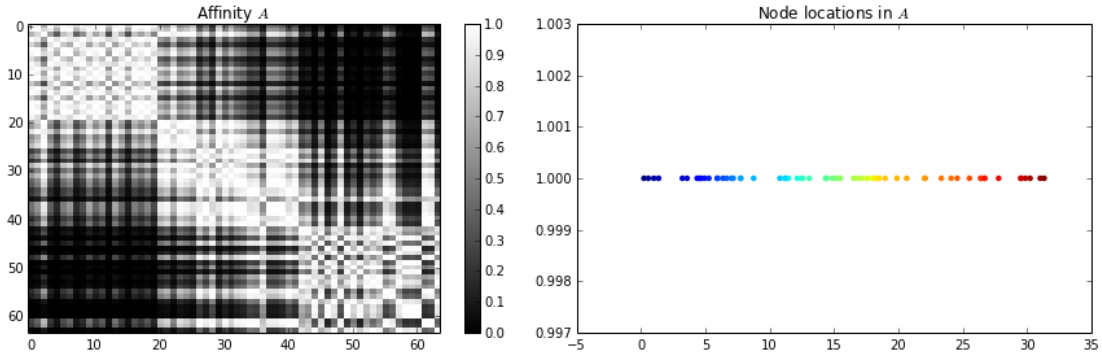
8. Now we want to repeat the process on the folders at the next level. First we double the diffusion time, and then repeat the process starting at step 3, except that we take the folders from this level as the singletons at the next level. We also take the average diffusion distance between all the points in a cluster at the new diffusion time to be the distance between the clusters at the new level. We repeat steps 3-7, doubling the diffusion time at each new level, until at the end all the levels are joined.

Building Flexible Trees

In [1]:
```
from imports import *
np.random.seed(20090403)
```

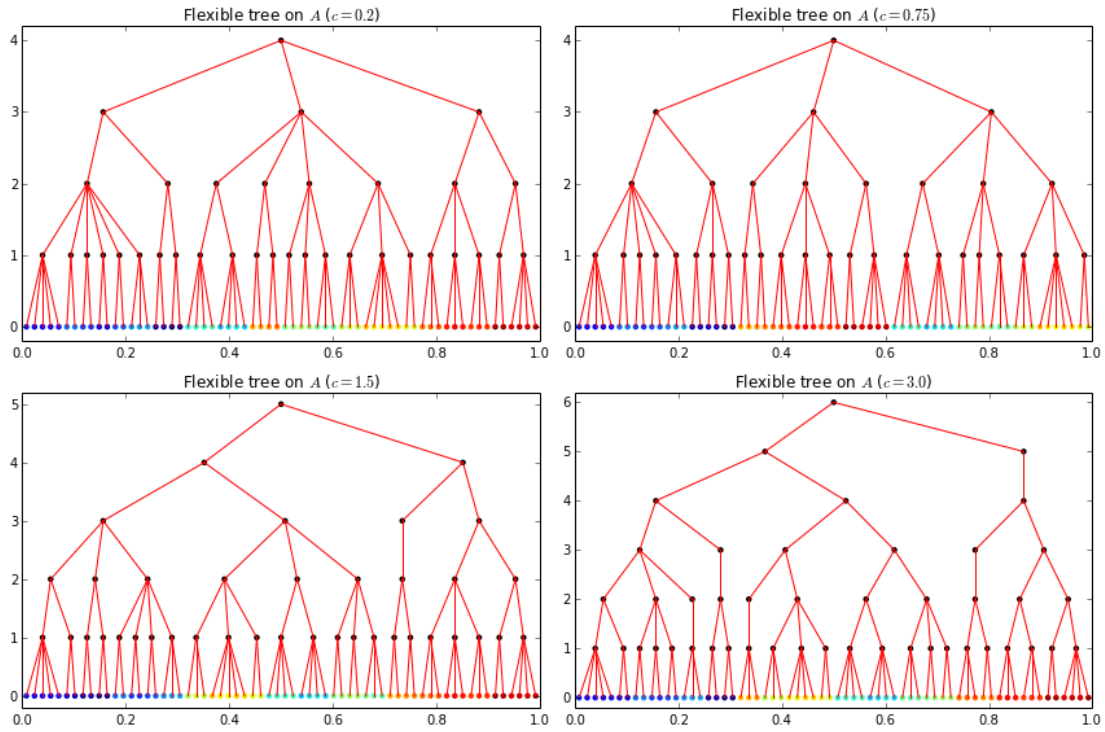
Again we consider a toy affinity for demonstration purposes. We take $A(i,j) \sim \exp\left(-\frac{|x_i - x_j|^2}{100}\right)$, where points are drawn from a Gaussian with mean 5 for nodes 0-20, 15 for nodes 21-41, and 25 for nodes 42-63, and standard deviation 3.

In [2]:
```
A = np.zeros([64,64])
means = [5.0,15.0,25.0]
sigma = 3.0
locs = np.zeros(64)
locs[0:21] = np.random.normal(means[0],sigma,21)
locs[21:42] = np.random.normal(means[1],sigma,21)
locs[42:] = np.random.normal(means[2],sigma,22)
for i in xrange(64):
    for j in xrange(64):
        d_ij = locs[i]-locs[j]
        A[i,j] = np.exp(-(d_ij**2.0)/100.0)
fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
bwplot(A,colorbar=True,title="Affinity $A$")
fig.add_subplot(122)
leafcolors = np.arange(-1.0,1.0,2.0/64.0)[locs.argsort().argsort()]
plt.scatter(locs,np.ones(len(locs),np.int),c=leafcolors,edgecolors='none')
plt.title("Node locations in $A$")
plt.tight_layout()
plt.show()
```



We next construct some flexible trees with different tree constants on A. We see that higher values of c give rise to trees which are taller and more hierarchical, while lower values lead to flatter trees, as we would expect.

In [3]:
```
fig = plt.figure(figsize=(12,8))
trees = []
for idx,tree_constant in enumerate([0.2,0.75,1.5,3.0]):
    trees.append(flex_tree_build.flex_tree_diffusion(A,tree_constant))
    fig.add_subplot(2,2,idx+1)
    plot_tree(trees[idx],leafcolors=leafcolors,
              title="Flexible tree on $A$ ($c={}$)".format(tree_constant))
plt.tight_layout()
plt.show()
```



4.1.3 Other approaches

Of course, there are many possible algorithms for building partition trees. Even slight modifications of the algorithms above may lead to better results for some datasets. For example, in the top-down algorithm for binary trees above, a simple modification would be to split the trees into 2^k pieces based on the signs of the coordinates of the first k nontrivial eigenvectors. Or perhaps the trees could be split into a small number of pieces by k-means clustering the points in a node based on the diffusion distances between points in the node. In practice, the optimal design of tree algorithms would be a somewhat supervised process, potentially reinforced by cross-validation or other measures of accuracy. The two algorithms presented here are simple and empirically seem to work well enough to enable the questionnaire process (to be described in 5.3) to work, and so we exhibit them here. But the literature contains many methods for building trees: [14] contains a method based on covering the space of points with balls of a given radius in diffusion distance, and then doubling the radius at higher levels; the algorithms of Ward and others previously mentioned contain methods for building hierarchical trees. Facilitating the concept of level for purposes of the Earth Mover's Distance (described in 5.2) is the only non-standard necessary feature of the trees we use here..

4.2 Discrete Haar and Haar-like bases on partition trees

Given a partition tree or trees on a dataset that expresses a geometry of the points inside the dataset, it is often useful to express the data in terms of local averages in the tree geometry. In this section, we discuss Haar and Haar-like bases for the space of functions in one dimension first on binary trees, and then on arbitarily-formed trees. Then we introduce the tensor Haar or tensor

Haar-like basis for two-dimensional data, which is our primary geometric tool in reconstructing the probability field underlying the data.

4.2.1 Binary Trees

We first consider the simplest case, that of a dyadic binary tree on 2^L nodes. The tree, then, consists of L levels, and at each level, the folders are exactly half the size of the folders in the level above. Then we can define a Haar basis for the space of functions defined on the 2^L nodes by reference to the tree.

- Let h_0 be the all-ones vector. Projecting a function onto this vector results in the mean of the function on all the nodes.
- For each non-singleton folder I in the tree, associate a Haar vector h_I with the folder. Choose a "left" child and a "right" child. Then set h_I to zero on nodes outside of I, to 1 on nodes in the left child, and to -1 on nodes in the right child. Now since there are equal numbers of nodes on left and right, the difference between the mean on the left child and the parent is offset exactly by the difference between the mean on the right child and the parent. Hence these vectors measure the differences in means between the left and right children of I.
- Then normalize each Haar vector to have l_2 norm 1 by dividing by $\sqrt{\#I}$.

Then the Haar vectors form an orthonormal basis for the space of functions on 2^L points. We might also consider the l_1-normalized version where the last step is omitted. If we do this, then the inverse transform has a factor of $\frac{1}{\#I}$ associated with each Haar vector.

Now suppose we have, instead of a dyadic tree, an arbitrary binary tree. Then the construction of the Haar vectors is unchanged except for the normalization.

Suppose that at node I, the children have m and n nodes, respectively. Apart from the children of I, every other vector is constant on the support of h_I. Hence we need to normalize the left and right nodes separately, so that their sum is 0. So if we multiply the m positive entries by n and the n negative entries by m, we achieve orthogonality. After doing this, the norm of the vector is $\sqrt{mn\,(m+n)}$. To obtain an orthonormal basis, we take the vector that is $+1$ on the left nodes and -1 on the right nodes, and modify it by multiplying the $+1$s by $\sqrt{\frac{n}{m(m+n)}}$ and the -1s by $\sqrt{\frac{m}{n(m+n)}}$. Therefore it is quite easy to extend the dyadic case to arbitrary binary trees.

4.2.2 Arbitrary Trees

When there are more than two subnodes, it is not as obvious what to do. As introduced in [14], we can create a **Haar-like** basis which is orthonormal on an arbitrary partition tree. Let Ω be the set of points, and let T be the partition tree. Then we do this as follows:

- Define h_0 to be the all-ones vector, just as above.
- Now if a node I has two children, then the Haar vector associated with that node is the vector from the binary case. But more generally, a node I with k children $\{J_1, J_2, ..., J_k\}$of sizes $\{n_1, n_2, ..., n_k\}$ has $k-1$ Haar-like vectors associated with it. An entire basis can be naturally obtained by performing Gram-Schmidt on the characteristic functions of all but one folder of every node – choosing different subsets of vectors will produce different bases. The basis that follows can be obtained in this manner.
- In lieu of performing Gram-Schmidt, we can also simply write down a particular Haar-like basis that is in some sense interpretable (vectors are positive on a particular folder and negative and reproducible as follows:

- Let I be any node of the tree. If I has one or fewer children, then h^I is empty (no basis vectors are associated with I).
- If I has $k \geq 2$ children, then there are $k-1$ vectors associated with I. Order the children of I from 1 to k (the particular Haar-like basis chosen will depend on the ordering), and construct the ith vector as follows (for $1 \leq i < k$):

$$h_i^I(x) = \begin{cases} +1 & x \in \underline{I}_i \\ -1 & x \in \underline{I}_k k > i \\ 0 & \text{otherwise} \end{cases}$$

- Then h^I is the set of all these vectors.
- So if for some particular node, $k = 4$ and the n_i are $\{1, 2, 3, 1\}$ (only the support of the parent is shown, everything else is zero), then we have:

$$h^I = \left\{ \begin{array}{ccc} +1 & 0 & 0 \\ -1 & +1 & 0 \\ -1 & +1 & 0 \\ -1 & -1 & +1 \\ -1 & -1 & +1 \\ -1 & -1 & +1 \\ -1 & -1 & -1 \end{array} \right\}$$

- Then we can write H, the unnormalized basis as the union of all these vectors and the all ones vector:

$$H = \left\{ h^I, I \in T \right\} \cup h_0$$

- Finally, we normalize each vector by the same process as for binary

trees, but for each vector, we treat the nodes marked +1 as one child, and the nodes marked -1 as the other child, ignoring that there may be multiple differently-sized nodes with a given sign.

– Hence in the example above, the final set of vectors associated with the 4-child node is

$$\psi^t = \left\{ \begin{array}{ccc} +\sqrt{\frac{6}{7}} & 0 & 0 \\ -\frac{1}{\sqrt{42}} & +\frac{1}{\sqrt{3}} & 0 \\ -\frac{1}{\sqrt{42}} & +\frac{1}{\sqrt{3}} & 0 \\ -\frac{1}{\sqrt{42}} & -\frac{1}{2\sqrt{3}} & +\frac{1}{2\sqrt{3}} \\ -\frac{1}{\sqrt{42}} & -\frac{1}{2\sqrt{3}} & +\frac{1}{2\sqrt{3}} \\ -\frac{1}{\sqrt{42}} & -\frac{1}{2\sqrt{3}} & +\frac{1}{2\sqrt{3}} \\ -\frac{1}{\sqrt{42}} & -\frac{1}{2\sqrt{3}} & -\frac{\sqrt{3}}{2} \end{array} \right\}$$

4.3 The Bi-Haar Basis

In our problem setting, in addition to bases for functions of one variable, we are often interested in expressing matrices, or functions of two variables, in alternate bases for the space of all such matrices. The primary decomposition of this space that we will concern ourselves with is formed by considering the product of two partition trees – a tree on the rows and a tree on the columns. Suppose that we have S, a tree on the rows, and T, a tree on the columns. In trying to express the elements of a matrix as a product of rows and columns, it is natural to consider submatrices formed by taking the product of two folders $I \in S$ and $J \in T$. Now consider a permutation of the rows which sorts the rows such that each folder is contiguous in the permutation. Since there is a strict hierarchial partition, this is immediately possible (in fact, there are many possible permutations that satisfy this condition). Likewise consider a permutation of the columns which causes column folders to be contiguous as well. Then for each folder $I \in S$ and

each folder $J \in T$, there is a submatrix $I \otimes J$ that is a contiguous rectangle under the row and column permutations previous described. There are $(\#S\#T)$ such rectangles in total.

Now consider the Haar vectors associated with S and T. Each vector in the Haar basis consists of values which are constant on a particular folder or set of folders (either positive or negative as appropriate).

Definition. Let $\{u_0, ..., u_{m-1}\} \subset \mathbb{R}^m$ be the vectors of a Haar or Haar-like basis for functions induced by a tree S on a set Ω_R where $\#\Omega_R = m$, and let $\{v_0, ..., v_{n-1}\} \subset \mathbb{R}^n$ be the vectors of a Haar or Haar-like basis for functions induced by a tree T on a set Ω_C where $\#\Omega_C = n$. Then a **bi-Haar basis** Ψ for the space of functions induced by the product tree $S \times T$ on $\Omega_R \times \Omega_C$ is defined as:

$$\Psi = \{u_i \otimes v_j, 0 \leq i < m, 0 \leq j < n\}$$

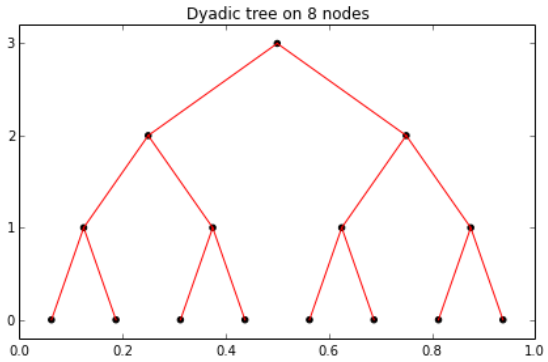
This **bi-Haar** basis implied by S and T forms an orthonormal basis for the set of all $m \times n$ matrices, and we can find the expansion in this basis by first expanding the rows in the Haar basis on the rows, and then expanding the columns of the result in the Haar basis on the columns.

The Haar and bi-Haar Bases

In [1]:
```
from imports import *
```

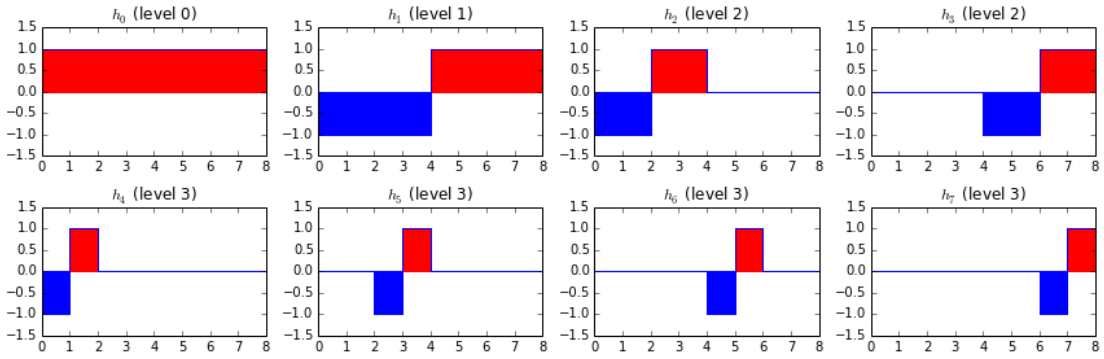
We consider first the standard dyadic Haar basis on 2^n points. Here we take $n = 3$ so we have a tree on eight points.

In [2]:
```
n = 3
t = tree.dyadic_tree(n)
plot_tree(t,title="Dyadic tree on {} nodes".format(2**n))
plt.tight_layout()
plt.show()
```



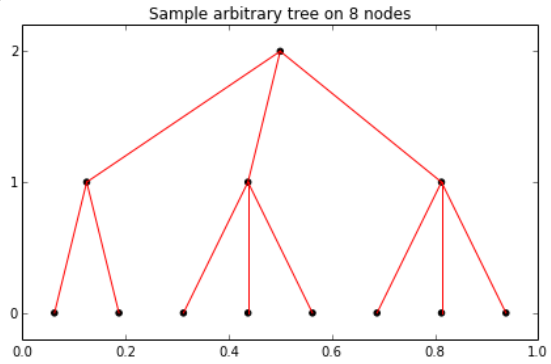
We calculate the standard Haar basis for this tree, which consists of eight vectors, pictured below. There is a single node h_0 which is constant on all nodes and so the projection of a function on the nodes onto that vector represents the average over all the nodes. Additionally, there is one vector corresponding to each node of the tree. These vectors are supported on the elements of that node and the projection of a function onto each of them represents the difference in averages between the children of each node.

In [3]:
```
hb,nodes = haar.compute_haar(t,True,"L1")
fig = plt.figure(figsize=(12,4))
for i in xrange(t.size):
    fig.add_subplot(2,4,i+1)
    x = np.arange(0,t.size+1)
    ypost = np.concatenate([hb[:,i],hb[-1:,i]])
    ypre = np.concatenate([hb[0:1,i],hb[:,i]])
    plt.step(x,ypost,where='post')
    y = np.vstack([ypre,ypost])
    plt.fill_between(x,np.max(y,axis=0),0,
                     where=np.max(y,axis=0)>0.0,color='r')
    plt.fill_between(x,np.min(y,axis=0),0,
                     where=np.min(y,axis=0)<0.0,color='b')
    plt.ylim(-1.5,1.5)
    plt.title("$h_{}$ (level {})".format(i,
                    0 if i==0 else t[nodes[i]].level))
plt.tight_layout()
plt.show()
```

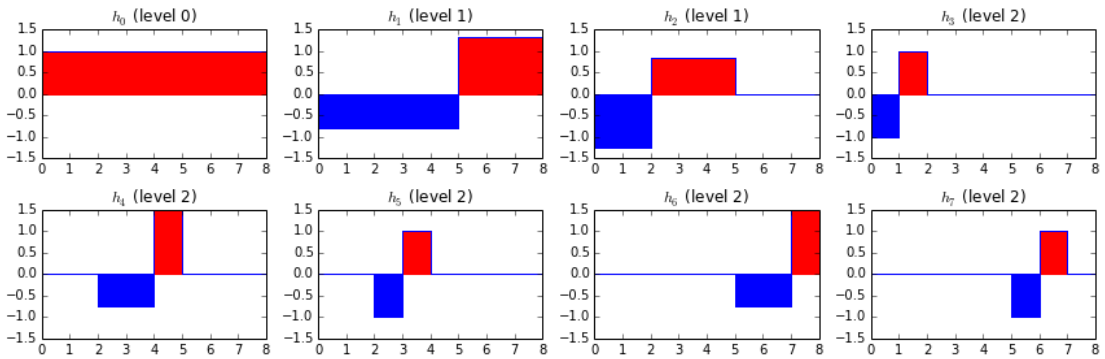


Next we consider an arbitrary tree, and generate the corresponding Haar-like basis for it.

In [4]:
```
ft = tree.ClusterTreeNode(range(8))
ft.create_subclusters([0,0,1,1,1,2,2,2])
for child in ft.children:
    child.create_subclusters(range(child.size))
ft.make_index()
plot_tree(ft,title="Sample arbitrary tree on {} nodes".format(ft.size))
plt.tight_layout()
plt.show()
```



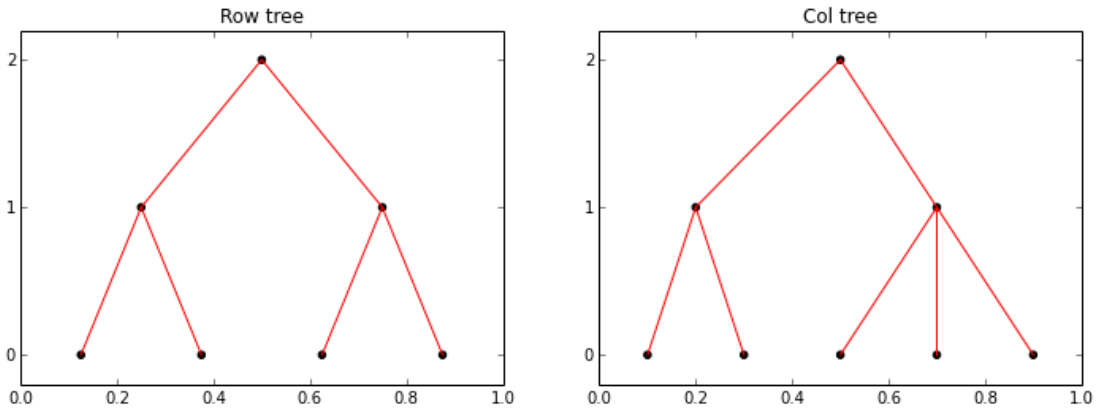
In [5]:
```
hb, nodes = haar.compute_haar(ft,True,"L1")
fig = plt.figure(figsize=(12,4))
for i in xrange(ft.size):
    fig.add_subplot(2,4,i+1)
    x = np.arange(0,ft.size+1)
    ypost = np.concatenate([hb[:,i],hb[-1:,i]])
    ypre = np.concatenate([hb[0:1,i],hb[:,i]])
    plt.step(x,ypost,where='post')
    y = np.vstack([ypre,ypost])
    plt.fill_between(x,np.max(y,axis=0),0,
                     where=np.max(y,axis=0)>0.0,color='r')
    plt.fill_between(x,np.min(y,axis=0),0,
                     where=np.min(y,axis=0)<0.0,color='b')
    plt.ylim(-1.5,1.5)
    plt.title("$h_{}$ (level {})".format(i,
                    0 if i==0 else ft[nodes[i]].level))
plt.tight_layout()
plt.show()
```



Notice that unlike in the dyadic case, the height of the vectors varies based on the ratio of the number of nodes in the $+$ (blue) group and the $-$ (red) group. Also notice that if a node has k children, there are $k-1$ Haar vectors that are supported on that node but no smaller nodes. For example in this sample tree, the root node has three children, and two Haar vectors: h_1 and h_2.

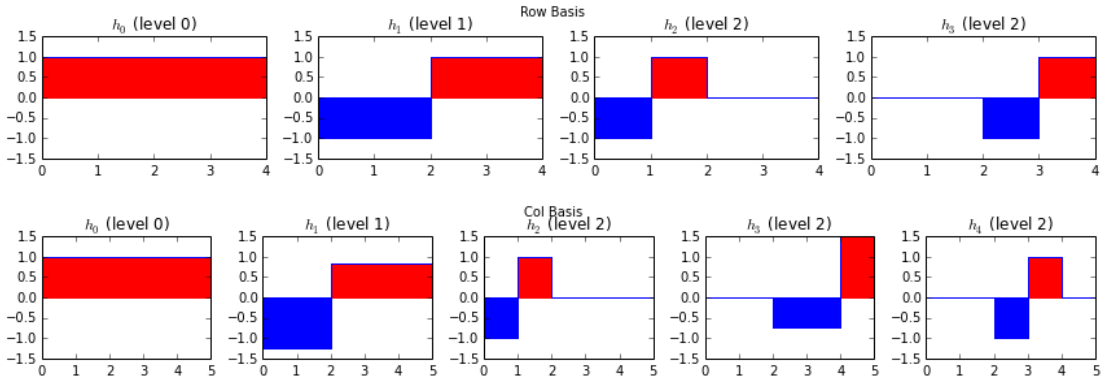
Next we turn attention to the bi-Haar basis. If there are m nodes on the rows and n nodes on the columns, there are mn vectors in the bi-Haar basis, so in the interests of brevity we will consider simpler trees of 4 and 5 nodes.

In [6]:
```
row_tree = tree.dyadic_tree(2)
col_tree = tree.ClusterTreeNode(range(5))
col_tree.create_subclusters([0,0,1,1,1])
for child in col_tree.children:
    child.create_subclusters(range(child.size))
col_tree.make_index()
fig = plt.figure(figsize=(12,4))
fig.add_subplot(1,2,1)
plot_tree(row_tree,title="Row tree")
fig.add_subplot(1,2,2)
plot_tree(col_tree,title="Col tree")
plt.show()
```



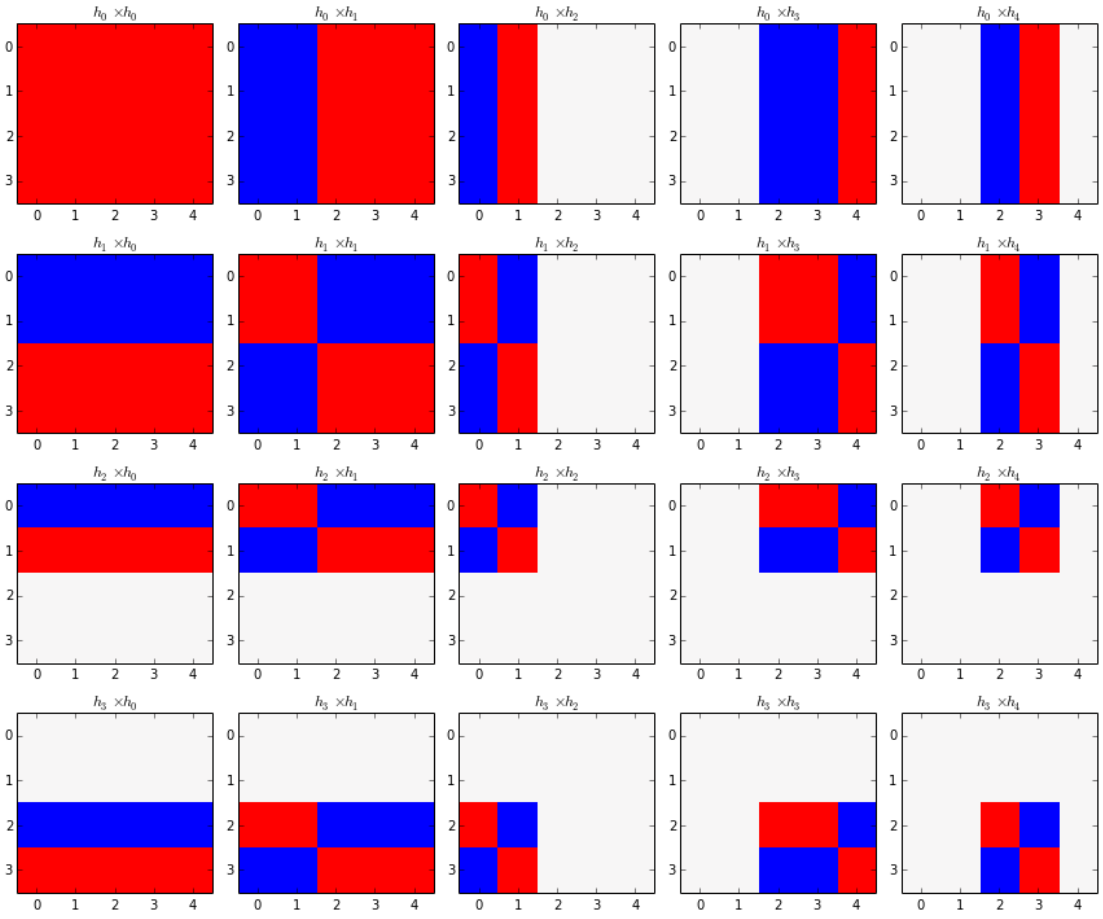
In [7]:

```
for t,title in [(row_tree,"Row Basis"),(col_tree,"Col Basis")]:
    hb, nodes = haar.compute_haar(t,True,"L1")
    fig = plt.figure(figsize=(12,2))
    fig.suptitle(title)
    for i in xrange(t.size):
        fig.add_subplot(1,t.size,i+1)
        x = np.arange(0,t.size+1,1)
        ypost = np.concatenate([hb[:,i],hb[-1:,i]])
        ypre = np.concatenate([hb[0:1,i],hb[:,i]])
        plt.step(x,ypost,where='post')
        y = np.vstack([ypre,ypost])
        plt.fill_between(x,np.max(y,axis=0),0,
                         where=np.max(y,axis=0)>0.0,color='r')
        plt.fill_between(x,np.min(y,axis=0),0,
                         where=np.min(y,axis=0)<0.0,color='b')
        plt.ylim(-1.5,1.5)
        plt.xlim(0,t.size)
        plt.xticks(range(t.size+1))
        plt.title("$h_{}$ (level {})".format(i,
                        0 if i==0 else t[nodes[i]].level))
    plt.tight_layout()
    plt.show()
```



The bi-Haar basis is made of the tensor products of the vectors in the row basis and the vectors in the column basis. Each one of these products forms an $m \times n$ matrix which is supported on the appropriate product folder. We demonstrate the 20 basis vectors that make up the bi-Haar basis for 4×5 matrices, given these row and column trees:

In [8]:
```
rows,cols = row_tree.size,col_tree.size
row_hb = haar.compute_haar(row_tree,False,"L1")
col_hb = haar.compute_haar(col_tree,False,"L1")
fig = plt.figure(figsize=(12,10))
for i in xrange(rows):
    for j in xrange(cols):
        fig.add_subplot(rows,cols,i*cols+j+1)
        plt.yticks(np.arange(0,rows+1))
        plt.title("$h_{} \\times h_{}$".format(i,j))
        cplot(5.0*np.outer(row_hb[:,i],col_hb[:,j]))
plt.tight_layout()
plt.show()
```



4.4 Function Smoothness in the bi-Haar Basis

In section 2, we introduced a generative model for binary questionnaire data. In that model, we made reference to a smoothness condition. We now make precise the notion of smoothness from that definition.

4.4.1 Smoothness in one dimension

First we consider smoothness in one dimension. Let Ω be a set of points, let T be a partition tree on Ω, and let Ψ be a Haarlike basis for functions on Ω induced by T. Recall that in 4.1 we defined the **tree metric** on Ω to be:

$$D_{tree}(x,y) = \begin{cases} \min\{|p| : x,y \in p, p \in q, q \in T\} & x \neq y \\ 0 & x = y \end{cases}$$

so that the distance between two distinct points is the size of the smallest folder containing both of the points. Let $f : \Omega \to \mathbb{R}$ and $\alpha > 0$.

Then we have that, up to constants that depend on the tree and α only, the following two conditions are equivalent:

1. There is a constant C_1 such that for any $\psi \in \Psi$associated with a folder $I \in T$:

$$|\langle f, \psi \rangle| \leq C_1 |I|^{\alpha + \frac{1}{2}}$$

2. f is α-Hölder with constant C_2 in the tree metric:

$$|f(x) - f(y)| \leq C_2 \cdot D_{tree}^{\alpha}(x,y)$$

for all $x, y \in \Omega$.

The intuitive meaning of this is that for functions, **being smooth in the tree metric $\Leftrightarrow$ Haar coefficients decay like a power of the folder size.** This was proved in [14].

4.4.2 Smoothness in two dimensions

In [6], Gavish and Coifman proved the natural extension of the above to the bi-Haar basis. Let Ω_1 and Ω_2 be two sets of points, and let $\Omega = \Omega_1 \times \Omega_2$. Let T_1 and T_2 be partition trees on Ω_1 and Ω_2, respectively. Then let Ψ be the bi-Haar basis for functions on Ω. Let $f : \Omega \to \mathbb{R}$ and let $\alpha > 0$. Then similarly to the one dimensional case, we have that the following two conditions are equivalent, up to constants that depend only on the trees and α:

1. There is a constant C_1 such that for any $\psi \in \Psi$which is associated with the rectangle $R \in T$:
$$|\langle f, \psi \rangle| \leq C_1 \, |R|^{\alpha + \frac{1}{2}}$$

2. f is α-bi-Hölder with constant C_2 in the tree metric, meaning that:
$$|f(x_1, y_1) - f(x_1, y_2) - f(x_2, y_1) + f(x_2, y_2)| \leq C_2 D^{\alpha}_{T_1}(x_1, x_2) D^{\alpha}_{T_2}(y_1, y_2)$$
for all $x, w \in \Omega_1$ and $y, z \in \Omega_2$. Here D_{T_j} is the tree metric induced by T.

Again we have the intuitive meaning of this: for functions, **being smooth in the sense of having small mixed difference quotient$\Leftrightarrow$having small bi-Haar coefficients on small rectangles**. So in this sense, the equivalence of these conditions allows us to precisely define what it means to be smooth with respect to the geometry induced by the bi-Haar basis: the bi-Hölder/mixed derivative condition is the relevant measure. Additionally, we have that the magnitude of the Haar coefficients on small folders overall serves as an easily

computable proxy for the constants in the bi-Hölder condition. We will exploit this in our reconstruction methods in 6.

4.5 Reconstruction on Known Geometry

We have the following theorem, also from Gavish and Coifman. [13] Let Ω and Ψ be as before, the product of two sets and the bi-Haar basis induced by two trees S and T on the product of the two sets. Let $f : \Omega \to \mathbb{R}$ be a function. Then define

$$g = \sum_{|R(\psi)|>\epsilon, \psi \in \Psi} \langle f, \psi \rangle \psi$$

Here $R(\psi)$ is the size of the smallest rectangle which contains the support of ψ. So g is the projection of f onto the subspace spanned by bi-Haar functions that are associated with rectangles of size greater than ϵ.

Then if f is $\alpha-$bi-Hölder with constant C, we have:

$$\|f - g\|_2 \leq \epsilon^{\alpha} \left(\frac{C}{\underline{B}} \right) \sqrt{\gamma \left(\log_{\overline{B}} \epsilon \right)^2 + \beta \log_{\overline{B}} \epsilon + \beta^2} \tag{1}$$

where

$$\gamma = \frac{1}{2} \left(1 - \left[\log_{\overline{B}} \left(\underline{B} \right) \right]^{-2} \right), \beta = \left(1 - \overline{B}^{2\alpha} \right)^{-1}$$

and $\overline{B}$ and $\underline{B}$ are tree balance constants such that for all folders I in S or T,

$$\underline{B} \leq \frac{\#I}{\#\overline{I}} \leq \overline{B}$$

Then γ measures the amount that the tree deviates from balance, while β measures the maximal growth rate of the tree. The proof of this is in [13].

Using this, we can obtain a pointwise approximation result.

Let $F : \Omega \to [0,1]$ be a probability field, and let X be an $M \times N$ sampling of that probability field, such that

$$X(i,j) \sim \text{Bernoulli}\left(F\left(i,j\right)\right)$$

Define

$$\hat{F} = \sum_{|R(\psi)|>\epsilon, \psi \in \Psi} \langle X, \psi \rangle \, \psi$$

and

$$G = \sum_{|R(\psi)|>\epsilon, \psi \in \Psi} \langle F, \psi \rangle \, \psi$$

We can decompose the $\mathcal{L}^2$ norm of the error:

$$\left\| F(i,j) - \hat{F}(i,j) \right\|_2 \lesssim \left\| F(i,j) - G(i,j) \right\|_2 + \left\| G(i,j) - \hat{F}(i,j) \right\|_2 \tag{2}$$

Then

$$\mathbb{E}\left[\left\| G(i,j) - \hat{F}(i,j) \right\|_2^2 \right] = \sum_{|R(\psi)|>\epsilon} \mathbb{E} \left\langle X(i,j) - F(i,j), \psi \right\rangle^2$$

Suppose that $R\left(\psi\right) > \epsilon$ is $m \times n$, and suppose that $\psi = u \otimes v$ where $u \in S$ and is supported on I and $v \in T$ and is supported on J. Then

$$\left\langle X(i,j) - F(i,j), \psi \right\rangle^2 = \left[\frac{1}{mn} \sum_{I' \in \underline{I}} \sum_{J' \in \underline{J}} u(I') v(J') \sum_{(i,j) \in I' \times J'} \left(X(i,j) - F(i,j) \right) \right]^2$$

$$\lesssim \frac{1}{m^2 n^2} \sum_{I' \in \underline{I}} \sum_{J' \in \underline{J}} \left(u(I') v(J') \right)^2 \left[\sum_{(i,j) \in I' \times J'} \left(X(i,j) - F(i,j) \right) \right]^2$$

Now for any Haarlike basis,$|u(I')| \lesssim \sqrt{\frac{m}{\#I'}}$ and $|v(J')| \lesssim \sqrt{\frac{n}{\#J'}}$, so:

$$\leq \frac{1}{mn} \sum_{I' \in \underline{I}} \sum_{J' \in \underline{J}} \frac{1}{\#I' \#J'} \left[\sum_{(i,j) \in I' \times J'} X(i,j) - \sum_{(i,j) \in I' \times J'} F(i,j) \right]^2$$

Taking expectations and using $\mathbb{E}X(i,j) = F(i,j)$ and the independence of the $X(i,j)$ draws:

$$\mathbb{E}\left[\langle X(i,j) - F(i,j), \psi \rangle^2 \right] \lesssim \frac{1}{mn} \sum_{I' \in \underline{I}} \sum_{J' \in \underline{J}} \frac{1}{\#I' \#J'} \mathbb{E} \left[\sum_{(i,j) \in I' \times J'} X(i,j) - \sum_{(i,j) \in I' \times J'} F(i,j) \right]^2$$

$$= \frac{1}{mn} \sum_{I' \in \underline{I}} \sum_{J' \in \underline{J}} \frac{1}{\#I' \#J'} \sum_{(i,j) \in I' \times J'} \mathrm{Var}\left(X(i,j)\right)$$

$$= \frac{1}{mn} \sum_{I' \in \underline{I}} \sum_{J' \in \underline{J}} \frac{1}{\#I' \#J'} \sum_{(i,j) \in I' \times J'} F(i,j)(1 - F(i,j))$$

Since $F(i,j)(1 - F(i,j)) \leq \frac{1}{4}$, we have

$$\mathbb{E}\left[\langle X(i,j) - F(i,j), \psi \rangle^2 \right] \lesssim \frac{1}{mn} \sum_{I' \in \underline{I}} \sum_{J' \in \underline{J}} \frac{1}{4}$$

The balancing constants of the tree give a constant bound on the number of subfolders of a given folder. So

$$\mathbb{E}\left[\langle X(i,j) - F(i,j), \psi \rangle^2 \right] \lesssim \frac{1}{\epsilon MN}$$

Now the number of rectangles that are larger than ϵ containing (i,j) is bounded above by $\mathcal{O}\left(\log\left(\frac{1}{\epsilon}\right)\right)$ (see [6]). So the total error in this part of the sum is

$$\mathbb{E}\left\| G(i,j) - \hat{F}(i,j) \right\|_2^2 \lesssim \log\left(\frac{1}{\epsilon}\right) \frac{1}{\epsilon MN}$$

Combining this with (1) and (2), we obtain:

$$\mathbb{E}\left[\left\|F(i,j)-\hat{F}(i,j)\right\|_2\right] \lesssim \epsilon^{\alpha}\left(\frac{C}{\underline{B}}\right)\left(\gamma\left(\log_{\overline{B}}\epsilon\right)^2+\beta\log_{\overline{B}}\epsilon+\beta^2\right)+\left(\sqrt{\log\frac{1}{\epsilon}}\right)\left(\sqrt{\frac{1}{\epsilon MN}}\right) \tag{3}$$

This result contains several logarithmic terms that are effectively constants for any practical matrix. Let us consider the implications, though, of this inequality. Suppose we have a function of known smoothness (a fixed α), and we want to approximate it by sampling some number of points MN. Up to constants and log factors, we know that if we had the true function F, we could approximate it with an error of by considering only bi-Haar coefficients on folders of size greater than ϵ. Then suppose that we also want to estimate the bi-Haar coefficients from the Bernoulli sampling process we have described. If we have chosen ϵ already, then the folders on which we will estimate the bi-Haar coefficients are precisely the folders of size ϵMN. But from the second part of (3), we see that that error is of order $\frac{1}{\sqrt{\epsilon MN}}$. Then setting these equal (to approximately equalize the order of the errors), we obtain:

$$\epsilon^{\alpha+\frac{1}{2}} \approx \frac{1}{\sqrt{MN}}$$

So what is the meaning of this? There are two sources of error in reconstructing a sampled function in this way. The first is the error inherent in estimating a smooth function by a truncated set of Haar coefficients, which is approximating the function by a function piecewise constant on rectangles. The second source of error is that we may not have enough samples to accurately estimate the Haar coefficients. So if we have a function of known smoothness, we can calculate the samples required to allow us to estimate that function to a particular accuracy using the above equation. Likewise if we have a fixed number of samples, we can calculate the size of rectangles on which we can approximate the Haar coeffi-

cients to within a certain accuracy. If we have a fixed ϵ and allow the number of samples MN to rise, the error in reconstructing a function of smoothness $\alpha = \frac{1}{2}$ is primarily the error that comes from having too few samples to accurately estimate the coefficients until we have approximately $\frac{1}{\epsilon^2}$ samples. Then we don't gain much more in the reconstruction because even though our estimates of the Haar coefficients get better, the error inherent in estimating from a truncated Haar expansion still dominates.

5 Dual network organization

In 3.3 we outlined some ways in which we could construct an affinity on (for example) columns of the matrix, calculate the diffusion map, the induced embedding, and build a partition tree. Then we have a discrete organization of the columns, which in the questionnaire model represent people. We view the folders of the partition tree as "demographics" – groups of people whose response profiles are close to one another. Many popular machine learning algorithms such as k-nearest neighbors or kernel smoothing make use of this information by using some averaging operator based on closeness to interpolate missing data, examine the data for coherence, identify outliers, and so on.

Instead let us turn to organizing the data in the opposite dimension – that is, given the organization of the people, let us organize the questions into groups of related questions. We could again use the previously described methods to construct an affinity between questions. A "question" is a function supported on the space of people; each question has its own response profile among the people. But instead of following our earlier procedure of constructing an affinity based on the global similarity of questions, treating each person as independent of each other, we should be able to improve the meaningfulness of that affinity

by incorporating our organization of the people into the process of constructing the affinity.

5.1 Dual geometry

In constructing the affinity that led to the organization of the people, we treated the questions as independent data points. If we use cosine similarity as the affinity, with a threshold at zero:

$$k(i,j) = \max\left(\frac{\langle X_i, X_j\rangle}{\|X_i\|\,\|X_j\|}, 0\right)$$

If each of the questions takes an answer "yes" (+1) or "no" (-1). Then the affinity is simply the number of matching answers minus the number of nonmatching answers, divided by the total number of questions. This essentially treats the questions as unstructured data; each question counts equally in the affinity, and each question is considered independently of the others. But suppose we are given some organization of the people in the form of a partition tree. Then we could potentially enhance our affinity calculation by incorporating the geometry of the partition tree into our calculation of affinity. One obvious way of doing this is to enhance the data by introducing new coordinates corresponding to the averages on the various folders in the partition tree.

As a very simple example, consider the following simplified "mini-questionnaire," which might be a subset of some other questionnaire.

	A	B	C
Q1	+1	-1	+1
Q2	+1	+1	+1
Q3	-1	+1	+1
Q4	+1	+1	-1
Affinity to A	1	0	0

Consider the relative affinities between the pairs (A, B) and (A, C). A agrees with both B and C on two of the questions and disagrees on two. Hence in cosine similarity both pairs have affinity 0; by Euclidean distance both pairs are equidistant, and so on. So from any global perspective, these two pairs have equal affinity to each other. Next suppose that we are externally given that there is some meaningful clustering of questions in which questions 1 and 2 are somehow related and questions 3 and 4 are also related. Then in addition to considering each question individually, we might consider the averages on the related questions as additional coordinates. Notice that in the generative model introduced in 2, the folder averages are approximations to the probability field averages on the folder. Since one purpose of the organization here is to identify people/question pairs whose probability field averages are close, the external knowledge that certain questions are already globally similar would lead us to want to put extra weight on the averages over those folders. So we take the natural partition tree induced by this organization of the questions, and consider the folder averages on every folder as additional coordinates:

Folder	A	B	C
$\{1\}$	+1	-1	+1
$\{2\}$	+1	+1	+1
$\{3\}$	-1	+1	+1
$\{4\}$	+1	+1	-1
$\{1,2\}$	+1	0	+1
$\{3,4\}$	0	+1	0
$\{1,2,3,4\}$	+0.5	+0.5	+0.5
affinity to A	1	$\frac{1}{21}$	$\frac{5}{21}$

In this multiscale model (A, C) has a significantly stronger affinity than (A, B). This occurs because even though the two pairs have equal numbers of differences at fine scales, when we consider the folder averages at coarser scales, A and C have matching folder averages, while B's folder averages deviate. Since our model is essentially a model of folder averages – which are our best estimates of the underlying probability field – it stands to reason that we want to emphasize those averages in our calculation of affinities. The organization of the people contains useful information that allows us to make the average response to folders of questions into a meaningful quantity in its own right and not just the sum of several independent variables. When we construct an affinity for the columns of a matrix based not only on the response profile against each row independently, but also including some organization of the rows, we call this a **dual affinity**. Once such a dual affinity is created, we can construct a graph, calculate the diffusion on the graph, and organize the columns. We refer to the resultant geometry consisting of the partition tree on the questions that we were given and the partition tree on the columns which we calculated, which can reasonably be said to be an organization on the rows and columns simultaneously, as a **dual geometry** on the data.

Naturally, the quality of the dual geometry is dependent on the quality of the partition tree on the opposite dimension used in its construction. If we simply took a random partition tree on the rows of the matrix, and used it in organizing the columns as above, we would not expect that process to yield a better affinity than simply treating each question independently. But if on the other hand, we have an organization of the rows which is of high quality; that is, which represents groupings of the data which are true in some sense, then our dual organization of the columns should be an improvement over an organization using no partition tree.

5.2 Earth Mover's Distance

Now the method of calculating a dual affinity in the previous section was quite crude; that is, adding folder averages as additional coordinates and calculating cosine similarity on the result is one possible way of incorporating the organizational data, but there is no particular reason to think it is a good one. In fact, the problem of measuring the similarity between two functions on some space of points that has a geometry associated with it is an important one in machine learning. One metric that is widely used is the Earth Mover's Distance (EMD), which is usually formulated as a distance between probability measures which measures the minimal cost (with a cost function defined by the geometry of the space) of transforming one distribution into another. Distributions which are small distortions of each other will have small EMD, and therefore EMD has the desirable property of being fairly insensitive to small perturbations.

In general, we can define an EMD-like distance between X_i and X_j, columns i

and j of the dataset X, given a partition tree S on the rows of X, to be:

$$EMD_S(i,j) = \sum_{I \in S} |m(X_i - X_j, I)| \, \omega(I)$$

where $m(A, I)$ is the mean value of vector A on the folder I and $\omega(I)$ is a weight function.

In [19], Leeb and Coifman defined several metrics and proved them equivalent to EMD with respect to the tree metric defined in 4.1 in settings such as ours; we will adopt a slight variation of the metric D_3 from that paper, which took

$$\omega(I) = |I|^{\alpha+1}$$

For our purposes, in order to have additional flexibility when dealing with flexible trees and more fully incorporate the notion of level into the distance, we add an additional parameter, and take

$$\omega(I) = 2^{-\alpha l(I)} \, |I|^{\beta}$$

where α is a constant related to level, β takes the place of $(\alpha+1)$ in the definition from [19], and $l(I)$ is the level at which the folder I is found in S (the root node is at level 0).

Then it remains to explain the roles of α and β in defining the metric. α is a constant related to the level of the folder in the tree. So increasing α results in higher weight being put on the differences in folders closer to the root of the tree; $\alpha = 0$ would correspond to all levels of the tree being equally weighted, while $\alpha < 0$ would put higher weights on folders closer to the leaves. In the binary questionnaire type setting, we typically choose $\alpha > 0$, because averages over many responses are likely to be closer to the true values of the probability field.

However, in applying methods like this to image processing or the like, $\alpha > 0$ would be analogous to putting high weights on low-frequency modes (patches of color, for example), while $\alpha < 0$ would be analogous to putting high weights on high-frequency modes (textures). The appropriate choice of α is driven by the application. β adjusts the weight by raising the size of the folder to a power directly. By the same principles, $\beta > 0$ puts higher weights on larger folders, while $\beta < 0$ puts higher weights on smaller folders.

In the dyadic binary tree case, these can be used interchangeably: $\alpha = 1, \beta = 0$ is the same as $\alpha = 0, \beta = 1$ in the case of a dyadic binary tree, since the size of each folder is half that of the folders at the next higher level. When dealing with arbitrary trees, β helps to normalize the weights on folders of widely differing sizes, as can occur when creating flexible trees.

Now supposing that we have calculated the EMD between all pairs of points, we can convert it to an affinity as was discussed in 3.3, by taking

$$k(x, y) = \exp\left(-\frac{\|x - y\|_{EMD}}{\epsilon}\right)$$

where ϵ is a parameter that restricts the affinity to nearby points; empirically we often use a constant multiple of the median of all the EMDs between pairs of points. From this affinity, we can construct a diffusion map and build a partition tree as before. But now let us consider the columns again. We already have an organization of the columns, but we can improve the quality of that organization by incorporating the geometry of the row partition tree using EMD. We can use the resulting column tree to in turn improve the row organization by utilizing the column tree, and so on. This iterative process leads directly to a general algorithm for analysis of data of this type.

5.3 Questionnaire Algorithm

1. Start by constructing some initial, global affinity on either the rows or columns of the matrix. (We will assume rows here).
2. Construct a graph with nodes of the rows and with edges whose weights are the affinities between nodes, as in 3.
3. Calculate the diffusion on the graph and use it to define the family of diffusion distances as in 3.2.
4. Use the diffusion distances to create a partition tree on the rows, as in 4.1
5. Using the partition tree from step 4 and EMD, construct an affinity on the columns of the matrix, as in 3.3.
6. Repeat steps 2-4 on this affinity to create a partition tree on the columns.
7. Use this partition tree in the same way to re-organize the rows.
8. Iterate this process as desired. See below for a principled stopping criterion.

There are many choices (which are largely problem specific choices) to be made in implementing this algorithm:

- Choose an initial direction (rows or columns). Some datasets are organized better in one direction than the other; iterations of EMD will be helped by the quality of the initial organization. Alternatively, we could choose to organize both the rows and columns independently. Then each iteration can use the tree on the opposite direction from the previous iteration. This treats the data more symmetrically, although the results will be quite similar to the process as described above.

- Choose a method of calculating the initial affinity, and any thresholding or parameters of that method.

- Choose a diffusion time to set a diffusion distance. A common choice is $\frac{1}{1-\lambda}$ where λ is the eigenvalue of the first non-trivial eigenvector. This is the diffusion time at which the entire graph is connected to itself [3].

- Choose a method for constructing trees. We discussed various methods in 4.

- Choose the α and β parameters for EMD. Discussion of the effects of different parameter choices is in 5.2.

- Choose when to stop the iteration. Based on the theorems in 4.4, we have the following interpretation of the iterative organization process: we are trying to find the bi-Haar basis which makes the probability field underlying the data most smooth. A good proxy for this smoothness is the relative smoothness of the **data** expressed in the bi-Haar basis. Since we know that the sum of the magnitudes of the Haar coefficients of a function is a measure of the smoothness of the function, we can use $\sum_{\psi \in \Psi} |\langle X, \psi \rangle|$, the sum of the absolute values of the Haar coefficients, as a measure of smoothness. So as long as our iterative process continues to reduce the sum, we continue to obtain better organizations, and should continue the iteration process. If the sum does not decrease, we may not be making further progress. In practice on example datasets, we have found that the iteration process usually offers only small marginal improvement after the first 1-3 iterations.

6 Reconstruction of the Probability Field

The product of the questionnaire process, then, is a pair of partition trees – one on the rows, one on the columns. So suppose we have S, a tree on the rows, T, a tree on the columns, and X, the observed data. Recall that in our model, X is the result of sampling a probability field F at each intersection of row and column. Now we are interested in reconstructing F. We construct the bi-Haar basis Ψ induced by the product of S and T. This basis consists of vectors which express the differences between the average value of X on a set of rectangles that cover the matrix at all scales. These rectangles include individual matrix elements, long thin rectangles which represent a small group's answers to many questions, or many people's answers to a small group of questions, all the way to the entire matrix. Given our smoothness condition, described in 4.4, and the data, our task is to reconstruct F, the probability field underlying the observed data. Now if the dual geometry of the questionnaire is "good", then the smoothness of F in this geometry will imply that we can improve our estimate of F by considering points that are close to each other in the geometry and combining in some manner the averages of rectangles at different scales.

In designing this process, two principles have to be balanced.

- Larger rectangles are more reliable because they are resilient to the quantization noise. Suppose that F is constant and equal to p on some large rectangle. Then averaging over that large rectangle will give us a better estimate of p than averaging over smaller rectangles.

- Smaller rectangles contain the information needed to produce a more accurate estimate. Our goal is to understand the structure of the data. Typically F will not be constant on a large rectangle, and so the information about subdividing it into smaller rectangles is where the structure

actually lies.

If the geometry of the space were given to us, then we would have good control of the first condition, because elementary statistics would imply that the accuracy of the average of a folder f would decay roughly as $\frac{1}{\sqrt{\#f}}$ because it is the sum of independent Bernoulli variables. However, here we have used the sampled data to create the geometry, so folder averages are not sums of iid samples from the underlying field. We can see this easily in one dimension. Suppose we have $F : \Omega \to [0,1]]$, a probability field on a one-dimension set. Then we let $f(x) \sim \text{Bernoulli}\,(F(x))$ be our observed data. Then it is true that $\frac{1}{\#\Omega}\sum_{x\in\Omega} f(x)$ as an estimator of $\frac{1}{\#\Omega}\sum_{x\in\Omega} F(x)$ will decay in this way as $\#\Omega$ grows. But now suppose we want to create a geometry relative to which F is as smooth as possible. There is essentially only one organization, which is to order Ω by the values of f with all the 0s in one folder and all the 1s in the other. But now if we call the interval containing zeros I, plainly $\frac{1}{\#I}\sum_{x\in I} f(x) = 0$. We cannot simply rely on the statistics of the folders to control the variance. Nevertheless, in the two-dimensional case, the organization just described is usually impossible, and the principle that folder averages are more reliable than individual values still stands.

So in general, however, we observe that the quantization noise leads us to shrink or suppress coefficients on small rectangles, while preserving coefficients on larger rectangles.

6.1 Thresholding by folder/coefficient size

We can, of course, reconstruct X exactly by taking

$$X = \sum_{\psi\in\Psi} \langle X, \psi\rangle\, \psi$$

Let $R(\psi)$ be the rectangle associated with the bi-Haar function ψ, and let $\mathcal{H}(X)$ represent the bi-Haar transform of X; ie $\mathcal{H}_\psi(X) = \langle X, \psi \rangle$.As we saw earlier in 4.4.2, decaying values of bi-Haar coefficients on smaller rectangles are equivalent to a smoothness condition in two variables. A natural way to exploit this is to suppress coefficients corresponding to small rectangles, which in a sense replaces the function defined by the original bi-Haar expansion with a smoother version. So we reconstruct F as a function of a threshold value ϵ, where coefficients which subdivide rectangles of size smaller than ϵ are set to zero.

More precisely, we apply the following transformation to the bi-Haar transform of X:

$$\mathcal{H}_\psi\left(\hat{F}\right) = \begin{cases} \mathcal{H}_\psi(X) & |R(\psi)| \geq \epsilon \\ 0 & |R(\psi)| < \epsilon \end{cases}$$

and then reconstruct the field as:

$$\hat{F} = \sum_{\psi \in \Psi} \mathcal{H}_\psi\left(\hat{F}\right)\psi$$

$\epsilon = 0$ corresponds to zero thresholding, in which case the matrix is reconstructed exactly. Lower ϵ results in higher fidelity to the observed data; higher ϵ results in a smoother function with more averaging across larger folders.

6.2 Shrinkage schemes

Another approach to the reconstruction process is to view the bi-Haar coefficients as a wavelet basis for the space of matrices. Then it becomes natural to apply common wavelet shrinkage schemes to the coefficients. Some of these schemes use a hard threshold as above, but others use a **soft** thresholding ap-

proach, where all coefficients are moved toward zero by a shrinkage value t :

$$\mathcal{H}_\psi\left(\hat{F}\right) = \begin{cases} \operatorname{sgn}\left(\mathcal{H}_\psi\left(X\right)\right)\left(\left|\mathcal{H}_\psi\left(X\right)\right| - t\right) & \left|\mathcal{H}_\psi\left(X\right)\right| > t \\ 0 & \left|\mathcal{H}_\psi\left(X\right)\right| \leq t \end{cases}$$

The most popular schemes of this type were introduced by Donoho and Johnstone and are standard in signal processing. We omit description of these methods, which can be found in [9, 10]. In one of the examples in section 8, we reconstruct with the scheme with a variable threshold that minimizes Stein's unbiased risk estimator (SURE).

6.3 Thresholding by hypothesis test

We can also consider thresholding based on the size of coefficients. Suppose we consider a single bi-Haar coefficient $\mathcal{H}_\psi\left(X\right)$ in the binary case, corresponding to a rectangle $R\left(\psi\right)$. Now the coefficient represents a rebalancing of the folder average μ_R among two subregions R_1 and R_2. A small coefficient corresponds to a small difference between the subregions, while a large coefficient represents a large difference. But even if the "true" coeffcient $\mathcal{H}_\psi\left(F\right)$ were zero–if the underlying probabilities for $R_1$and R_2 were actually equal–we would expect to observe nonzero values of $\mathcal{H}_\psi\left(X\right)$, due to the quantization noise. Deciding whether such values are below the level of noise can be characterized as a hypothesis test.

We propose the following model.

- Suppose R consists of R_+ "yes" answers and R_- "no" answers.
- Let the null hypothesis be H_0: The partition of R into R_1 and R_2 is random chosen from the set of all partitions of R that partition into sets of sizes R_1 and R_2.

- Hence under the null hypothesis, $R_{1+} \sim \text{hypergeometric}\left(r_+, \#R, \#R_1\right)$.
- Then we hypothesis test this at a suitable significance level:

$$\mathcal{H}_\psi\left(\hat{F}\right) = \begin{cases} \mathcal{H}_\psi\left(X\right) & \text{if we reject the null} \\ 0 & \text{otherwise} \end{cases}$$

We then repeat the test on each coefficient. This naturally tends to threshold away coefficients on small rectangles; note that if R contains one "yes" and one "no", then the bi-Haar coefficient corresponding to the split will be fairly large, but it will be set to zero by the hypothesis test for any reasonable p-value. To avoid standard problems with performing many hypothesis tests, one should apply false discovery rate control mechanisms like those suggested in [1, 2].

6.4 Spin Cycling

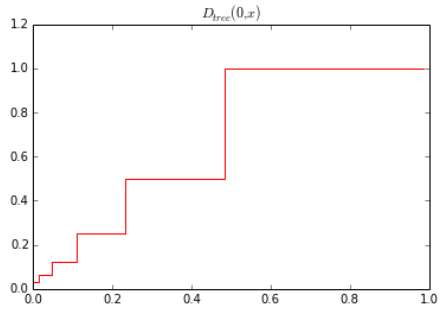
One of the most serious defects of using trees and the metrics they induce is that they are in some sense overspecific – in many cases, points are divided from each other somewhat arbitrarily, and there are artificially large distances in the tree metric. In the following notebook, we illustrate this and suggest a process of correction.

Spin Cycling of Partition Trees

In [1]:
```
import tree
import numpy as np
```

To illustrate this, consider the dyadic tree of n levels on the interval $[0, 1]$. This tree divides the interval into 2^n equally sized regions. Suppose we consider the distances between points in [0,1] in the tree metric. Points that lie within the same leaf folder have distance $\frac{1}{2^n}$; points that lie within the same folder at the next higher level have twice that distance, and so on. Now if we consider $D_{tree}\,(0, x)$ in this metric, it behaves a natural manner:

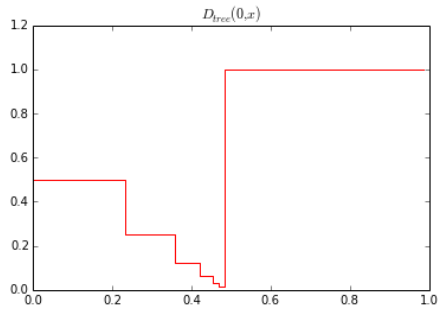
In [2]:
```
t = tree.dyadic_tree(6)
y = np.array([t.tree_distance(0,x) for x in t.elements])
plt.step(np.arange(0,t.size,1.0)/(1.0*t.size),y,'r')
plt.ylim(0.0,1.2)
plt.xlim(0.0,1.0)
plt.title("$D_{tree}(0,x)$")
plt.show()
```



However, now consider the point $\frac{1}{2} - \epsilon$. The distance function $D_{tree}\left(\frac{1}{2} - \epsilon, x\right)$ exhibits some strange behavior.

Points to the left of this point have a somewhat normal progression of tree distances. However, points to the right of $\frac{1}{2}$ all have distance 1 from $\frac{1}{2} - \epsilon$. This leads to the fact that $D_{tree}\left(\frac{1}{2} - \epsilon, \frac{1}{2} + \epsilon\right) = 1$ even for small ϵ.

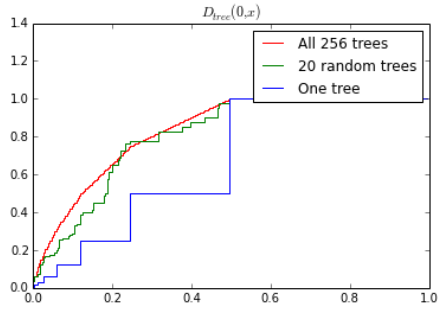
In [3]:
```
t = tree.dyadic_tree(6)
y = np.array([t.tree_distance(31,x) for x in t.elements])
plt.step(np.arange(0,t.size,1.0)/(1.0*t.size),y,'r')
plt.ylim(0.0,1.2)
plt.xlim(0.0,1.0)
plt.title("$D_{tree}(0,x)$")
plt.show()
```



This is a significant defect in the tree structure, which arises from what might be termed the basic arbitrariness of dividing points into folders. When the underlying data in the "true" space is neatly separated into a tree-like structure, this problem is minimized because points which are close in the true space will be close in the tree metric under all circumstances. But when the underlying data forms a continuum or the trees are restricted to folders of a particular size, situations where two points which are close in truth are far apart in the tree metric occur often. In some sense, what this means is that we cannot trust our trees, no matter how well they are constructed.

One good solution is to construct many trees, implement some process using them, and then combine the results of the processes to make a final estimate. In the setting of the tree metric as above, we can see that the average distance over 256 trees appears relatively smooth and natural. The removal of the artifacts arising from the use of discrete trees in our process is an important part of the reconstruction process that follows. Note that we can also achieve "better" smoothness by creating random trees.

In [4]:
```
n= 8
t = tree.dyadic_tree(n)
distances = np.zeros(2**n)
for i in xrange(2**n):
    y = np.array([1.0 if i > x else t.tree_distance(i,x)
                  for x in t.elements])
    distances += y.take(np.arange(i,2**n+i,1),
                        mode='wrap')/(2.0**n)
plt.step(np.arange(0,t.size,1.0)/(1.0*t.size),
         distances,'r',label="All {} trees".format(2**n))

iters=20
distances = np.zeros(2**n)
for i in np.random.randint(0,2**n-1,iters):
    y = np.array([1.0 if i > x else t.tree_distance(i,x)
                  for x in t.elements])
    distances += y.take(np.arange(i,2**n+i,1),
                        mode='wrap')/(iters)
plt.step(np.arange(0,t.size,1.0)/(1.0*t.size),
         distances,'g',label="{} random trees".format(iters))

y = np.array([t.tree_distance(0,x) for x in t.elements])
plt.step(np.arange(0,t.size,1.0)/(1.0*t.size),y,'b',
         label="One tree")

plt.title("$D_{}(0,x)$".format("{tree}",iters))
plt.ylim(0.0,1.4)
plt.xlim(0.0,1.0)
plt.legend()
plt.show()
```



We would like to exploit the desirable properties of partition trees without losing too much to this issue. One solution is to perfom what could be termed "spin cycling." [11] The process is to effectively distrust any particular pair of trees and instead use a somewhat randomized approach. Earlier 4.1.1 we considered an approach involving a randomized binary tree, where the cut between two subfolders, instead of being deterministic, was randomized over a range. Suppose there were some small set of points near the border of the cuts between the two subnodes. Any single tree would assign a border point to one side or the other. But if we sampled many trees, then border points would be assigned with some probability roughly proportional to their closeness to each side. In reconstructing function estimates based on trees, we will use this technique to improve our outcomes. A reconstruction based on a single pair of trees will contain artifacts like the ones just discussed. But by generating many trees, reconstructing functions based on each of them, and averaging the results, we can significantly reduce the impact of these artifacts while still taking advantage of the tree construct for organizational purposes.

7 The Adaptive Questionnaire

The questionnaire process described previously is a method for estimating an underlying probability field from empirically observed data. While in some circumstances recovering the underlying function that gave rise to empirical data might be the goal, another application that might be quite useful in practice is the construction of adaptive presentation of questionnaires in a way that reduces the data collection requirements. In this section, we propose an algorithm for reducing the size of a questionnaire by asking targeted questions to locate new individuals within the tree and infer their answers to unanswered questions.

7.1 Problem Setting

Consider the case of a psychological questionnaire like the MMPI, which consists of 567 questions. Suppose that some set of these **full questionnaires** has been given to various test-takers. Answering 500+ questions might be uncomfortable or expensive for the test-takers, and the test-givers would like to have a scheme whereby they could ask some reduced set of questions and infer the responses to all the questions with high accuracy. Suppose further that the test-givers can ask questions in an arbitrary order. We define an algorithm here which consists of the following steps:

- Training the algorithm on the training set to discover the probability field underlying the observed data.
- Defining a **question tree** by a modification of the tree algorithms in 4.1.1 on the training set.
- Adaptively asking a subset of questions from the question tree in order to place the new test-taker in the tree structure and expressing the probability vector for the new test-taker as an average of nearest neighbors in the question tree.

This algorithm has an additional practical advantage that the questionnaire process (which is somewhat computationally costly) need only be run once, and the work to be done for new test-takers is extremely efficient, and fits into standard paradigms of statistics (linear models especially). As such, it might be significantly more palatable to practitioners unfamiliar with advanced machine learning techniques. We describe each of these steps in detail in the following sections.

7.2 Training the algorithm

We begin by processing the training set of full questionnaires using the algorithm described previously, with the goal of reconstructing the underlying probability field from the data. We accomplish this by the following steps:

- Generate some number of tree-pairs consisting of a row tree and a column tree, using different randomized tree algorithms, different constants, etc.
- Reconstruct the probability field from the data and each tree-pair taken from the product of the set of row trees and the set of column trees. (So if we generated 15 pairs of trees using different parameters or randomizations, we have 225 tree-pairs from which to reconstruct).
- Average all these reconstructions to obtain $\overline{F}$, the mean of the estimates of F, with artifacts removed or minimized by the spin cycling process.

7.3 Defining a question tree

In this stage, we will define a (separate) question tree based on a process similar to the questionnaire algorithm, using the training data. We describe the process here for a binary tree, which is a modification of the binary tree algorithm in 4.1.1; later we will discuss modifications for more flexibly-sized trees.

1. Take an arbitrary row tree S from the row trees constructed in 7.2. Begin with a root node containing all the n points. Construct an $n \times n$ affinity on the columns using the EMD with respect to the tree S.
2. Build a graph on the training points contained in this node using this affinity, and calculate the first non-trivial eigenvector $\varphi(x)$ of the Markov process on the graph.

3. Note that the binary tree algorithm would suggest that we should split a node into children according to a rule such as the sign of $\varphi(x)$, or whether or not $\varphi(x)$ is above or below its median. Further, if $|\varphi(x)|$ is large, this indicates a stronger preference for being assigned to the corresponding child.

4. We specify k, a number of questions that we will ask of a new individual who has been located in this node. Then we find a small multiple regression on k questions to predict $\varphi(x)$. To find the k regressors, we can use the LARS algorithm of Efron et al [12], which adds regressors from a pool of regressors by increasing estimated parameters in the direction of highest correlation with the residual. Once we have found the k regressors, we calculate the corresponding β for the simple ordinary least squares problem $X_k\beta + \beta_0 \approx \varphi(x)$, and associate this linear model with this node.

5. Now we use the training data to calculate$\hat{\varphi}(x) = X_k\beta + \beta_0$, the predicted values of the eigenvector. Then we split the node into children based on the **predicted** values $\hat{\varphi}$, using the median or zero as desired.

6. Now we remove the working node and add the new children to a queue of nodes to process, and repeat steps 2-5 on each node, generating a small linear model on each, until the child nodes become too small (we used a size of 10 points in experiments).

We call the resulting structure a **question tree**, because it is a splitting of the data based on the answers to particular questions.

7.4 The Adaptive Questionnaire

Now suppose we are faced with a new individual Y who has not completed the questionnaire, and we have the question tree Q and associated linear models from the last section. We proceed by asking the k_{root} questions associated with the root node. After the test-taker answers, we use the linear model associated with the root node to find $\hat{\varphi}$ for the new test-taker. We then assign him to a child node C based on the value of $\hat{\varphi}$. Then we ask the k_C questions included in the linear model associated with C, calculate the $\hat{\varphi}$ value, and assign him to another child. We repeat this process until he has been assigned to a leaf node η. Then we define:

$$\hat{F}_Y = \frac{1}{\#\eta} \sum_{i \in \eta} \overline{F}_i$$

that is, the estimate of the probability field for person Y is the average of the estimates of probability fields for all the individuals (his nearest neighbors in the question tree metric) in the final folder to which he is assigned. Now the total number of questions he has answered is equal to $\sum_{Y \in C} k_C$; if k is constant from level to level, this will be approximately $k \log_2 \frac{n}{m}$ where m is the typical size of the leaf folders of Q. So if there are, for example, 2000 points in the training set and leaf nodes are of size approximately 10, then we will typically have to ask only $k \log_2 200 \approx 8k$ questions. In the examples section, we will present a full example of probability field recovery on a psychological questionnaire dataset.

8 Examples

8.1 Artificial Questionnaire Data

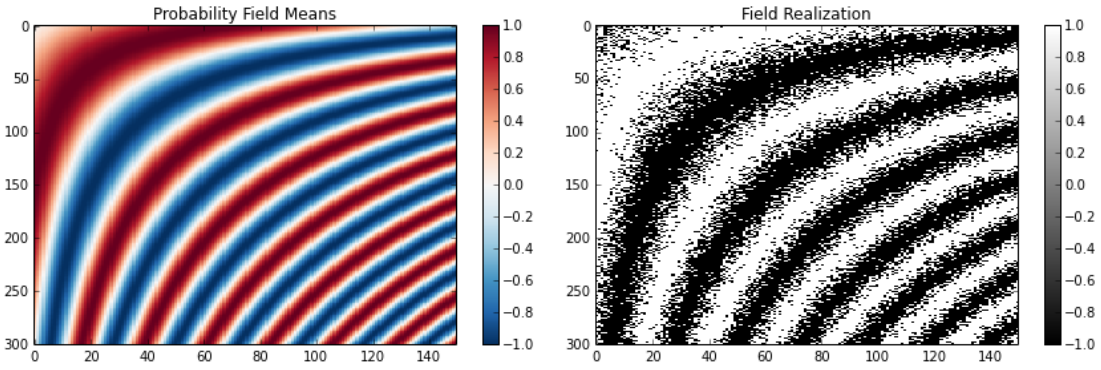
The Questionnaire Process, Recovery, and Reconstruction

In this example, we generate a sample probability field with intrinsic geometry in both dimensions. We sample the probability field, permute the rows and columns, producing a set of empirical data that looks like noise to the eye. We then apply the questionnaire methods to reconstruct and recover the original probability field.

In [17]:
```
from imports import *
bal_constant = 1.2
flex_constant = 0.5
```

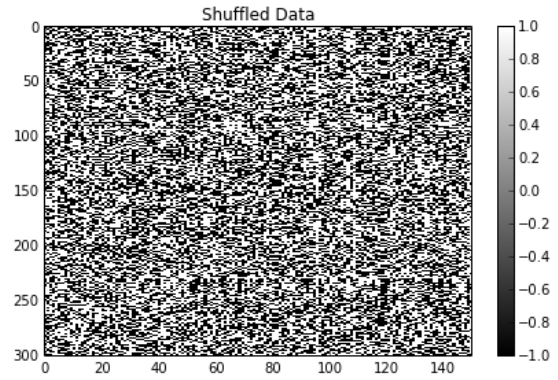
We define the sample probability field to be a 300×150 matrix P. We let i and j vary uniformly on $[0, 2\pi]$ over the rows and columns of the matrix respectively. Then we let the entries of the matrix be $P(i,j) = \frac{1}{2}\left(1 + \sin\left(\frac{i+j+2ij}{2}\right)\right)$. We then sample from the field, taking successes to be $+1$ and failures to be -1.

In [18]:
```
n_rows,n_columns = 300,150
means_matrix = np.zeros([n_rows,n_columns])
for i in xrange(n_rows):
    for j in xrange(n_columns):
        ii = i*2.0*np.pi/n_rows
        jj = j*2.0*np.pi/n_columns
        means_matrix[i,j] = np.sin((ii+jj+2*ii*jj)/2.0)*1.0

np.random.seed(20140301)
pf = artificial_data.ProbabilityField(means_matrix/2.0+0.5)
orig_data = pf.realize()

fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
cplot(means_matrix,colorbar=True,title="Probability Field Means")
fig.add_subplot(122)
bwplot2(orig_data,colorbar=True,title="Field Realization")
plt.tight_layout()
plt.show()
```



Next we generate a random permutation of the rows and a random permutation of the columns, and take the result of that to be the empirical data with which we are presented.

In [19]:
```
row_shuffle_order = np.random.rand(n_rows).argsort()
col_shuffle_order = np.random.rand(n_columns).argsort()
shuffled_means_matrix = means_matrix[row_shuffle_order,:][:,col_shuffle_order]
data = orig_data[row_shuffle_order,:][:,col_shuffle_order]
inverse_row_order = row_shuffle_order.argsort()
inverse_col_order = col_shuffle_order.argsort()
fig = plt.figure(figsize=(6,4))
bwplot2(data,colorbar=True,title="Shuffled Data")
plt.tight_layout()
plt.show()
```



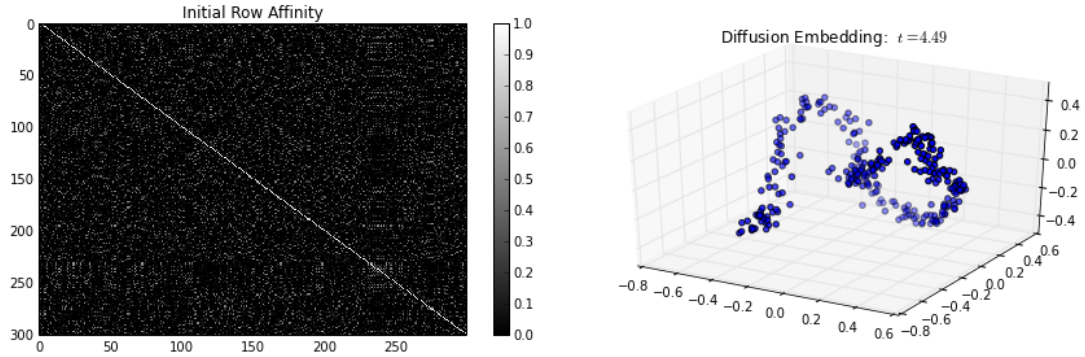
So our task, then, is to take this dataset of 45,000 “yes” or “no” answers, and using only the knowledge that the underlying probability field from which it was sampled is smooth with respect to some geometry on the rows and columns jointly, to recover the probability field and the permutations of rows and columns from which it came.

The Questionnaire Process

We begin by computing an affinity between rows. We will use cosine similarity as the initial affinity, thresholding at 0.05. The bright white diagonal line is the perfect affinity of each row with itself, while the gray represent varying degrees of affinity between different vectors.

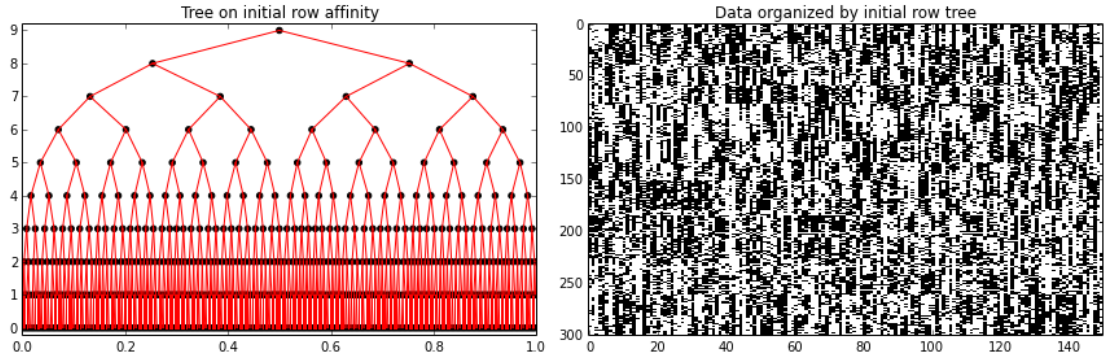
We take this initial affinity and calculate the eigenvectors and eigenvalues of the approximate bistochastic Markov matrix, producing a diffusion embedding of the rows.

In [20]:
```
init_row_aff = affinity.mutual_cosine_similarity(data.T,threshold=0.05)
init_row_vecs,init_row_vals = markov.markov_eigs(init_row_aff,12)
fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
bwplot(init_row_aff,colorbar=True,title="Initial Row Affinity")
ax = fig.add_subplot(122,projection="3d")
plot_embedding(init_row_vecs,init_row_vals,ax=ax)
plt.tight_layout()
plt.show()
```



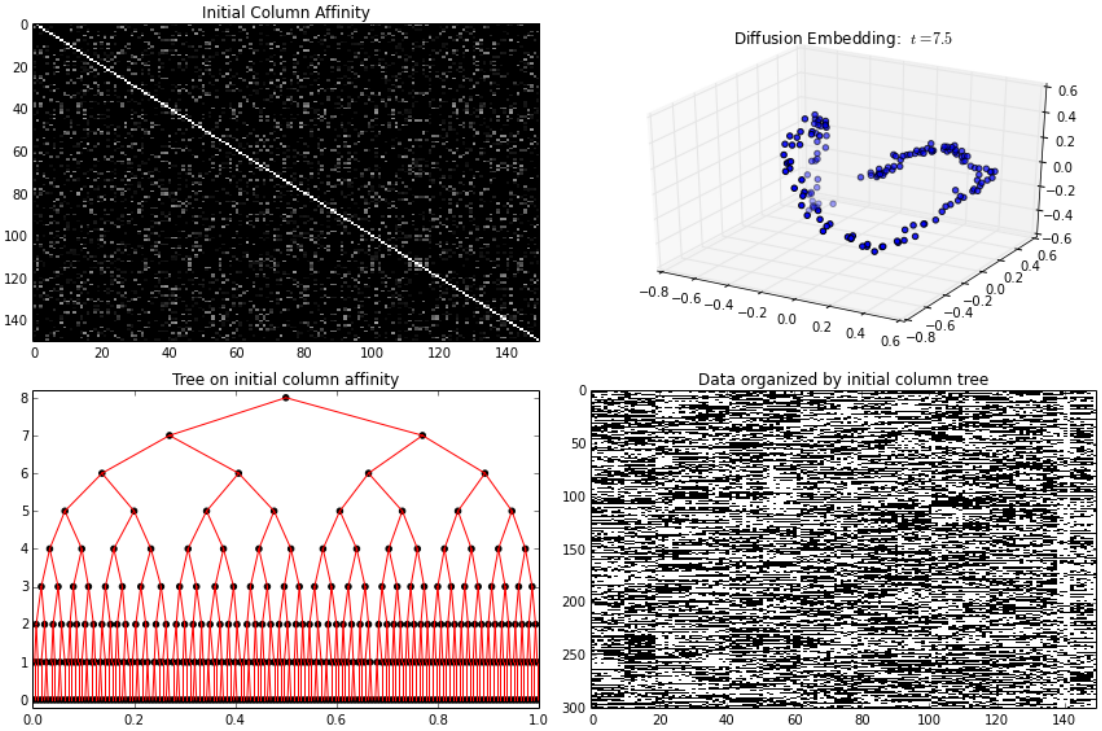
Now based on this diffusion embedding, we will construct a tree on the rows. We organize the data by rows and plot the result; despite the noisiness of the resulting plot, some structure can be seen.

In [21]:
```
init_row_tree = bin_tree_build.bin_tree_build(init_row_aff,
                                               'r_dyadic',bal_constant)
fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
plot_tree(init_row_tree,title="Tree on initial row affinity")
fig.add_subplot(122)
bwplot2(barcode.organize_rows(init_row_tree,data),
        title="Data organized by initial row tree")
plt.tight_layout()
plt.show()
```



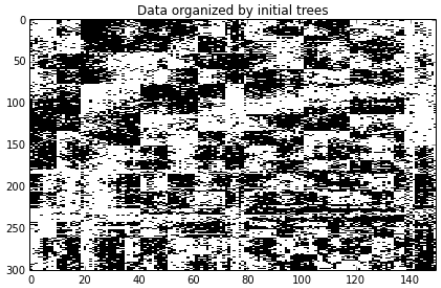
Next we repeat the identical process for the columns.

In [22]:
```
init_col_aff = affinity.mutual_cosine_similarity(data,threshold=0.05)
init_col_vecs,init_col_vals = markov.markov_eigs(init_col_aff,12)
fig = plt.figure(figsize=(12,8))
fig.add_subplot(221)
bwplot(init_col_aff,colobar=True,title="Initial Column Affinity")
ax = fig.add_subplot(222,projection="3d")
plot_embedding(init_col_vecs,init_col_vals,ax=ax)
init_col_tree = bin_tree_build.bin_tree_build(init_col_aff,
                                              'r_dyadic',bal_constant)
fig.add_subplot(223)
plot_tree(init_col_tree,title="Tree on initial column affinity")
fig.add_subplot(224)
bwplot2(barcode.organize_cols(init_col_tree,data),
        title="Data organized by initial column tree")
plt.tight_layout()
plt.show()
```



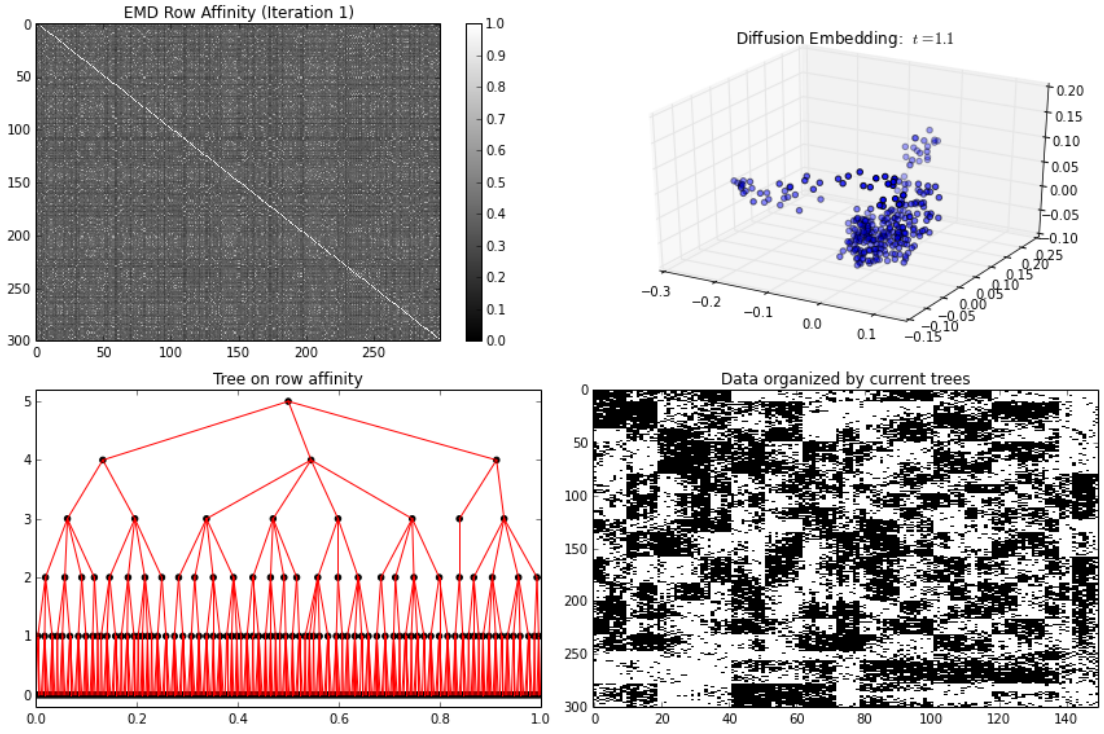
We can combine these initial trees by organizing the rows according to the row trees and the columns according to the column tree. Already in this we can see that the partition trees are creating submatrices of near-constant value.

In [23]:
```
bwplot2(barcode.organize_folders(init_row_tree,init_col_tree,data),
        title="Data organized by initial trees")
plt.tight_layout()
plt.show()
```



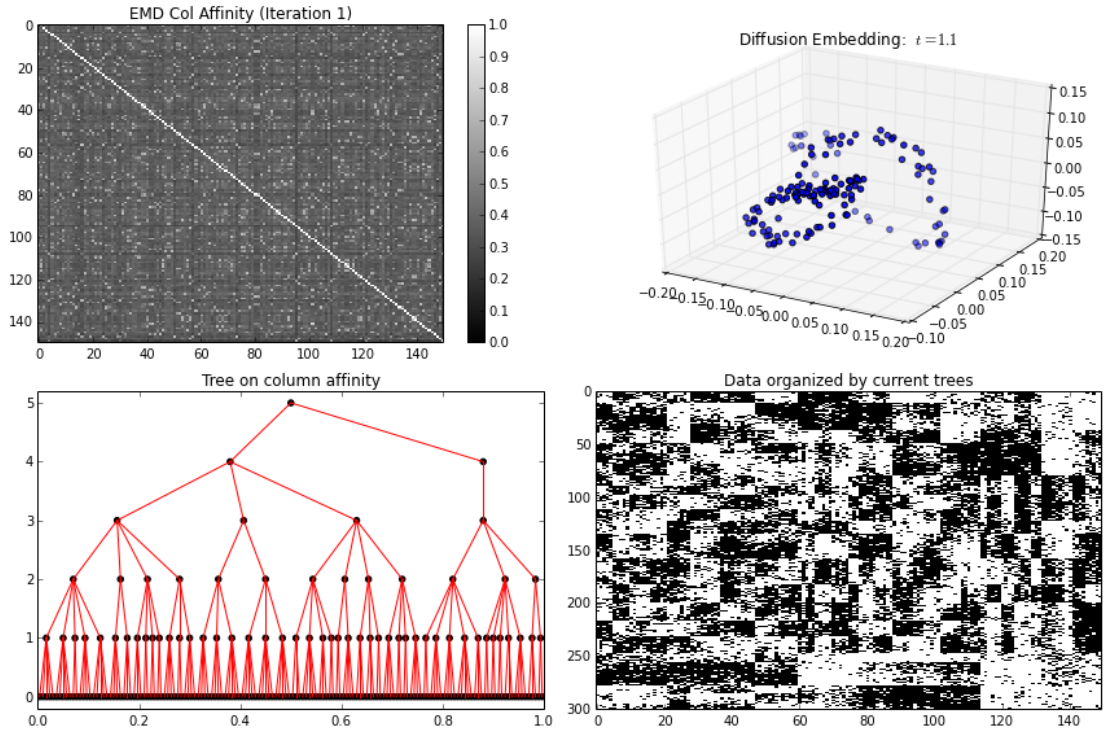
We proceed by calculating the Earth Mover's Distance (EMD) with constant $\beta = 1.0$ for the rows based on the organization of the columns.

In [24]:
```
row_emd = dual_affinity.calc_emd(data.T,init_col_tree,alpha=0.0,beta=1.0)
row_affinity = affinity.threshold(dual_affinity.emd_dual_aff(row_emd),0.0)
row_vecs,row_vals = markov.markov_eigs(row_affinity,12)
fig = plt.figure(figsize=(12,8))
fig.add_subplot(221)
bwplot(row_affinity,colorbar=True,title="EMD Row Affinity (Iteration 1)")
ax = fig.add_subplot(222,projection="3d")
plot_embedding(row_vecs,row_vals,ax=ax)
row_tree = flex_tree_build.flex_tree(row_affinity,flex_constant)
fig.add_subplot(223)
plot_tree(row_tree,title="Tree on row affinity")
fig.add_subplot(224)
bwplot2(barcode.organize_folders(row_tree,init_col_tree,data),
        title="Data organized by current trees")
plt.tight_layout()
plt.show()
```



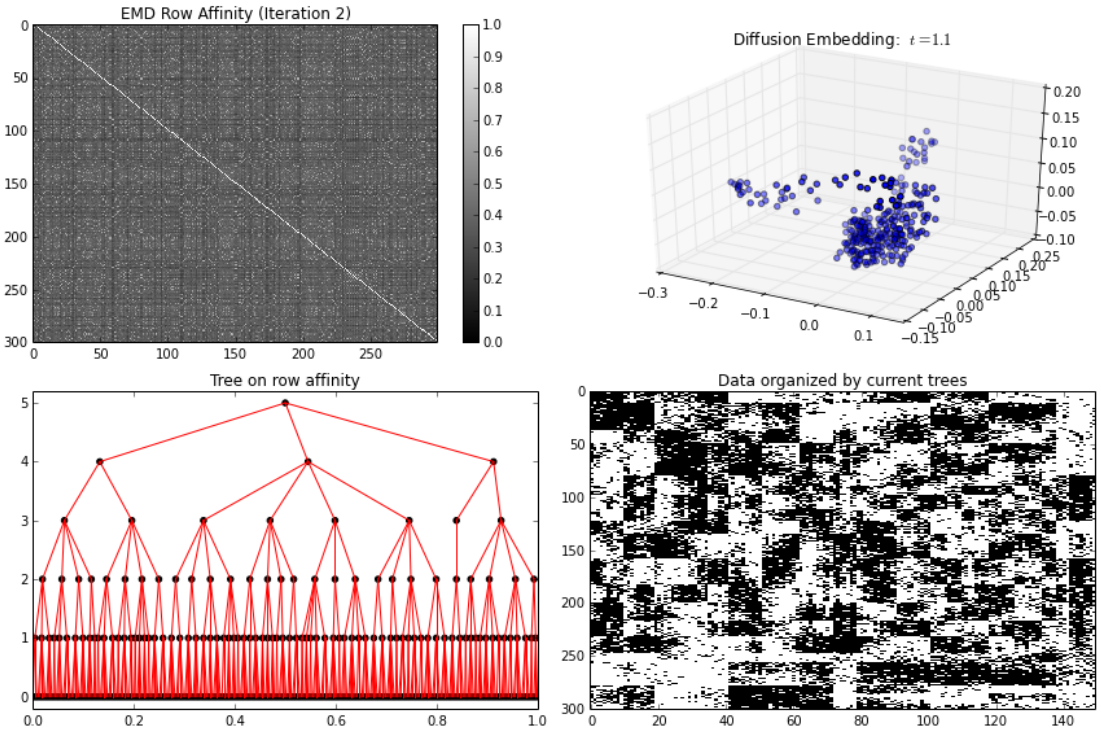
And we can now calculate the EMD and associated diffusion map and tree on the columns, using the newly generated row tree. We repeat this process for a total of two iterations below.

In [25]:
```
col_emd = dual_affinity.calc_emd(data,row_tree,alpha=0.0,beta=1.0)
col_affinity = affinity.threshold(dual_affinity.emd_dual_aff(col_emd),0.0)
col_vecs,col_vals = markov.markov_eigs(col_affinity,12)
fig = plt.figure(figsize=(12,8))
fig.add_subplot(221)
bwplot(col_affinity,colorbar=True,title="EMD Col Affinity (Iteration 1)")
ax = fig.add_subplot(222,projection="3d")
plot_embedding(col_vecs,col_vals,ax=ax)
col_tree = flex_tree_build.flex_tree(col_affinity,flex_constant)
fig.add_subplot(223)
plot_tree(col_tree,title="Tree on column affinity")
fig.add_subplot(224)
bwplot2(barcode.organize_folders(row_tree,col_tree,data),
        title="Data organized by current trees")
plt.tight_layout()
plt.show()
```



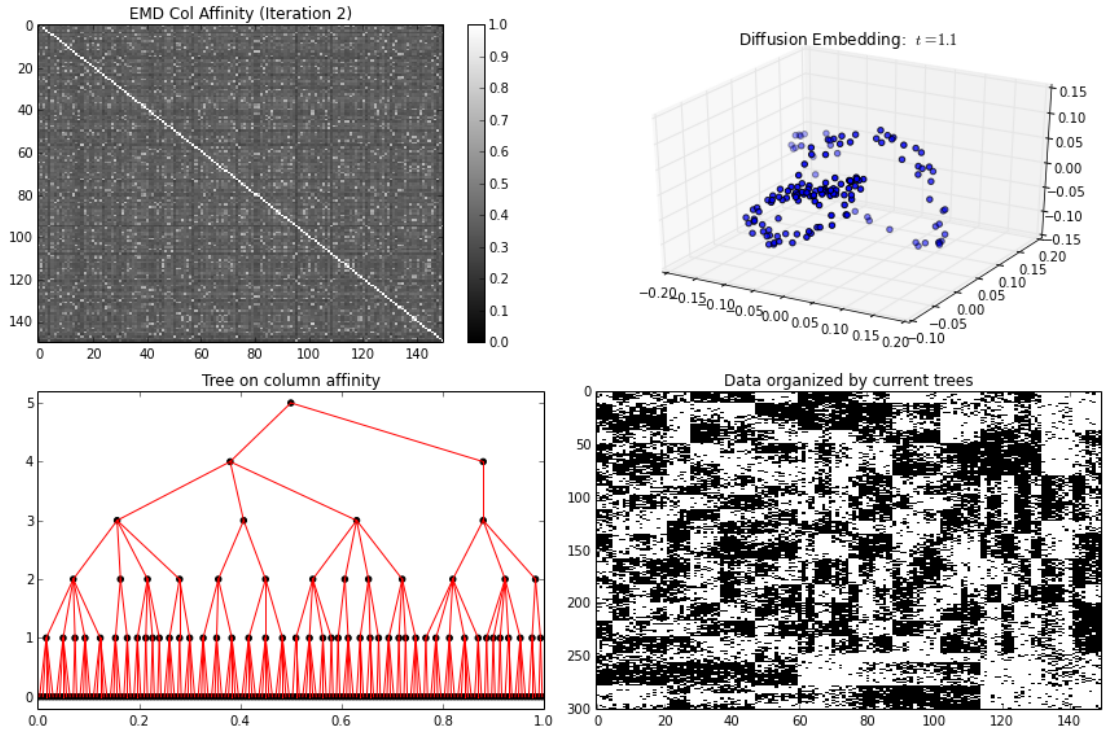
In [26]:

```
row_emd = dual_affinity.calc_emd(data.T,init_col_tree,alpha=0.0,beta=1.0)
row_affinity = affinity.threshold(dual_affinity.emd_dual_aff(row_emd),0.0)
row_vecs,row_vals = markov.markov_eigs(row_affinity,12)
fig = plt.figure(figsize=(12,8))
fig.add_subplot(221)
bwplot(row_affinity,colorbar=True,title="EMD Row Affinity (Iteration 2)")
ax = fig.add_subplot(222,projection="3d")
plot_embedding(row_vecs,row_vals,ax=ax)
row_tree = flex_tree_build.flex_tree(row_affinity,flex_constant)
fig.add_subplot(223)
plot_tree(row_tree,title="Tree on row affinity")
fig.add_subplot(224)
bwplot2(barcode.organize_folders(row_tree,init_col_tree,data),
        title="Data organized by current trees")
plt.tight_layout()
plt.show()
```



In [27]:

```
col_emd = dual_affinity.calc_emd(data,row_tree,alpha=0.0,beta=1.0)
col_affinity = affinity.threshold(dual_affinity.emd_dual_aff(col_emd),0.0)
col_vecs,col_vals = markov.markov_eigs(col_affinity,12)
fig = plt.figure(figsize=(12,8))
fig.add_subplot(221)
bwplot(col_affinity,colorbar=True,title="EMD Col Affinity (Iteration 2)")
ax = fig.add_subplot(222,projection="3d")
plot_embedding(col_vecs,col_vals,ax=ax,title="Diffusion Embedding: ")
col_tree = flex_tree_build.flex_tree(col_affinity,flex_constant)
fig.add_subplot(223)
plot_tree(col_tree,title="Tree on column affinity")
fig.add_subplot(224)
bwplot2(barcode.organize_folders(row_tree,col_tree,data),
        title="Data organized by current trees")
plt.tight_layout()
plt.show()
```



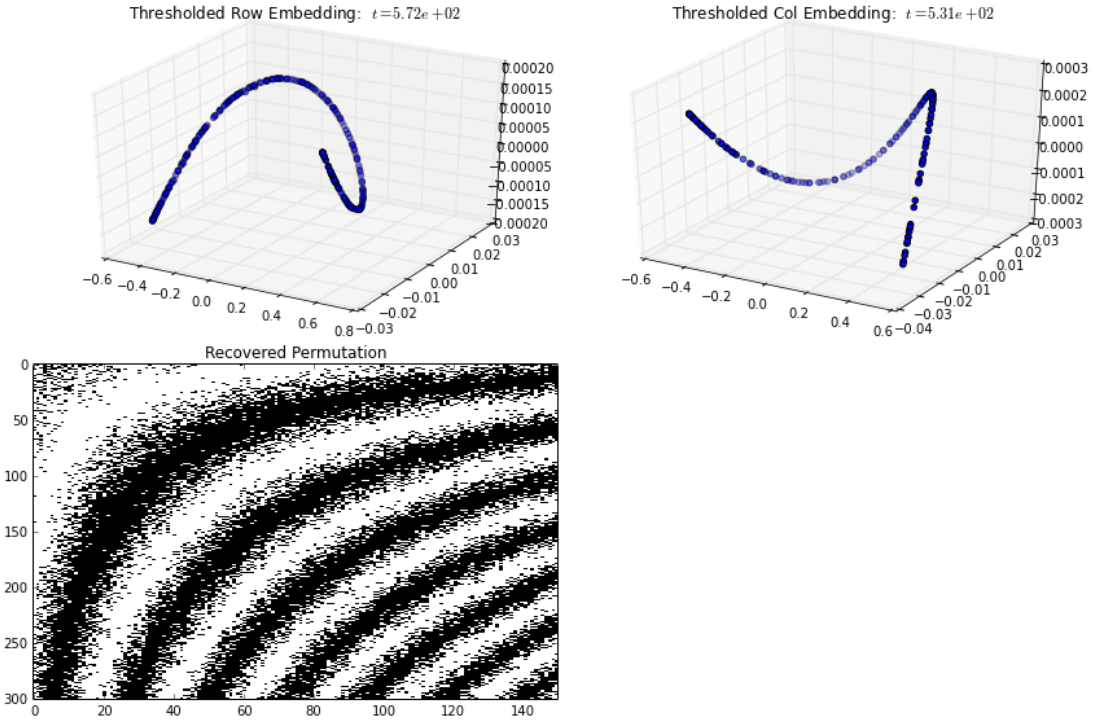
Recovering the Row and Column Permutations

Now in the lower right above we have organized the data by the trees, and obviously in this organization, we see that there are many submatrices of near-constant value. But as was pointed out in the section on tree-building, there is no single permutation of rows and columns that correspond to a particular pair of trees, so we cannot expect to recover exactly the original permutation of the rows and columns. Instead, many trees are equivalent. But we can often recover the original permutation in the following manner.

We begin by calculating affinities on the rows and columns with fairly high thresholds (choosing a precise one is data-dependent). Here we choose 0.53. This collapses the diffusion maps to a curve. We calculate those diffusion maps. Then we choose a random starting point along the curve (say x). The next point of our ordering is the nearest neighbor of x (in diffusion distance) that is not already in our ordering. By doing this, we pick a direction and walk along the curve in a parametric manner.

Now we may pick a starting point in the middle of the curve, in which case our ordering will have a large jump in it as we reach the end of the parameteric curve in diffusion space. However, we can easily detect this by reorganizing the data according to the permutation we have just obtained. We then choose a new starting point x to coincide with the largest distance between any two consecutive points in our ordering, which will generally coincide with such a jump. In this way, we can recover an appproximation of the original permutation of the data.

In [28]:
```
row_emd = dual_affinity.calc_emd(data.T,col_tree,alpha=0.0,beta=1.0)
col_emd = dual_affinity.calc_emd(data,row_tree,alpha=0.0,beta=1.0)
row_affinity = affinity.threshold(dual_affinity.emd_dual_aff(row_emd),0.53)
row_vecs,row_vals = markov.markov_eigs(row_affinity,12)
col_affinity =  affinity.threshold(dual_affinity.emd_dual_aff(col_emd),0.53)
col_vecs,col_vals = markov.markov_eigs(col_affinity,12)

row_order,col_order = barcode.organize_diffusion(data,
                        row_vecs.dot(np.diag(row_vals)),
                        col_vecs.dot(np.diag(col_vals)))
#reversed the orders here
#if we don't do this, then the permutation is a mirror image of the original.
row_order = [x for x in reversed(row_order)]
#col_order = [x for x in reversed(col_order)]
fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(221,projection="3d")
plot_embedding(row_vecs,row_vals,ax=ax,title="Thresholded Row Embedding: ")
ax = fig.add_subplot(222,projection="3d")
plot_embedding(col_vecs,col_vals,ax=ax,title="Thresholded Col Embedding: ")
fig.add_subplot(223)
bwplot2(data[row_order,:][:,col_order],title="Recovered Permutation")
plt.tight_layout()
plt.show()
```

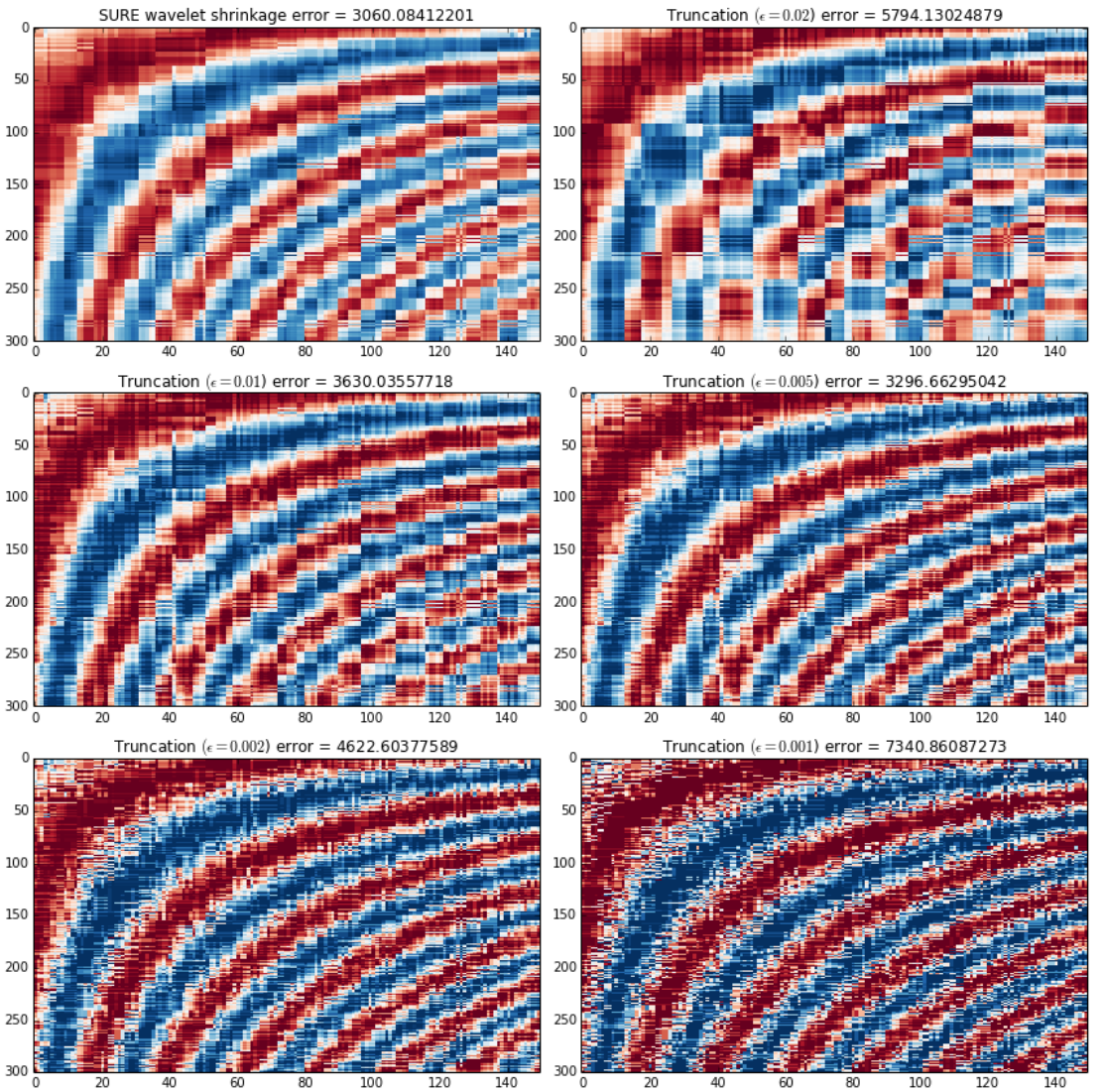


Reconstructing the Probability Field

Now we turn to reconstructing the underlying field. We begin by using the row and column trees generated above to reconstruct the field. We do this in this case using two methods: the wavelet shrinkage method of Donoho and Johnstone, utilizing the Stein unbiased risk estimator (SURE), and a simpler method of expressing the data in bi-Haar coefficients and truncating the coefficients supported on folders smaller than a certain specified threshold ϵ. We present plots for the SURE shrinkage as well as several different values of ϵ. The error in the title of each plot is the Frobenius norm $\left||\hat{X} - X|\right|_F$, where X is the data matrix and $\hat{X}$ is the reconstruction.

In [29]:

```
fig = plt.figure(figsize=(12,12))
fig.add_subplot(321)
recon = tree_recon.recon_2d_sure(data,row_tree,col_tree)
tree_recon.threshold_recon(recon,-1.0,1.0)
err = np.sum((recon-shuffled_means_matrix)**2)
cplot(recon[row_order,:][:,col_order])
plt.title("SURE wavelet shrinkage err = {}".format(err))
for idx,epsilon in enumerate([0.02,0.01,0.005,0.002,0.001]):
    fig.add_subplot(3,2,idx+2)
    recon = tree_recon.recon_2d_haar_folder_size(data,
                                    row_tree,col_tree,epsilon)
    tree_recon.threshold_recon(recon,-1.0,1.0)
    cplot(recon[row_order,:][:,col_order])
    err = np.sum((recon-shuffled_means_matrix)**2)
    plt.title("Truncation $(\epsilon = {}$) err = {}".format(epsilon,err))
plt.tight_layout()
plt.show()
```

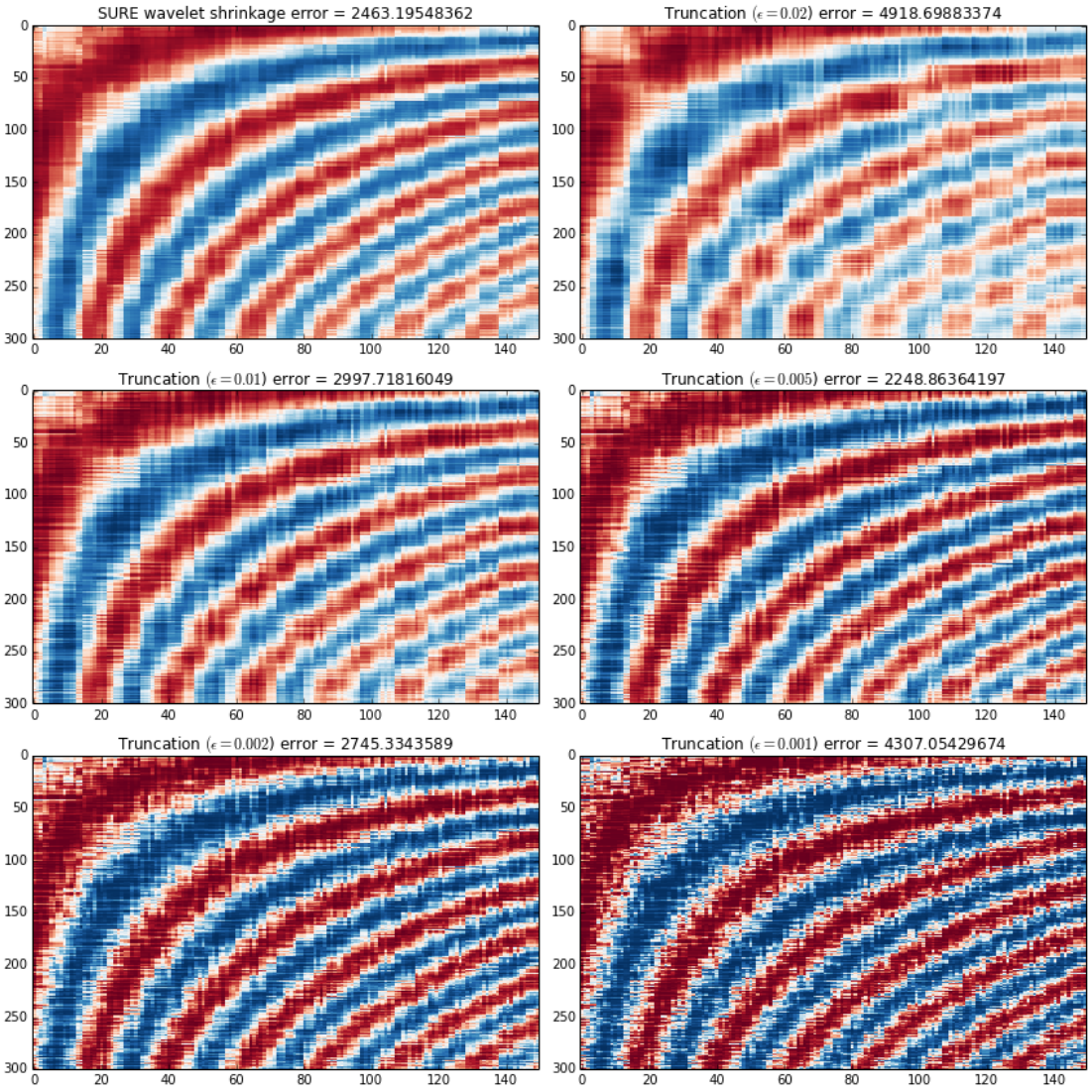


So we see that SURE performs best in this error metric, while the truncation reconstructions have to balance inaccuracy, where ϵ is too large, so that detail is lost, with overfitting, where ϵ is too small, so that we are coming closer to reconstructing the data than reconstructing the probability field.

Improvement through Spin Cycling

We can improve both of these methods by spin cycling. Here we run the questionnaire process seven times, with different tree constants each time. Then we cross match the row and column trees from the seven runs, generating 49 tree-pairs with which we reconstruct the probability field. We compare the average of the 49 reconstructions on SURE and on the values of ϵ above.

In [30]:
```
tree_constants = [0.1,0.3,0.5,0.75,1.0,1.5,2.0]
params_list = []
for tree_constant in tree_constants:
    kwargs = {}
    kwargs["n_iters"] = 2
    kwargs["threshold"] = 0.0
    kwargs["row_alpha"] = 0.0
    kwargs["col_alpha"] = 0.0
    kwargs["row_beta"] = 1.0
    kwargs["col_beta"] = 1.0
    kwargs["tree_constant"] = tree_constant
    params = questionnaire.PyQuestParams(questionnaire.INIT_AFF_COS_SIM,
                                         questionnaire.TREE_TYPE_FLEXIBLE,
                                         questionnaire.DUAL_EMD,
                                         questionnaire.DUAL_EMD,**kwargs)
    params_list.append(params)

qruns = []
for params in params_list:
    qruns.append(questionnaire.pyquest(data,params))

sure_recon = np.zeros(data.shape)
trunc_recons = []
for idx,epsilon in enumerate([0.02,0.01,0.005,0.002,0.001]):
    trunc_recons.append(np.zeros(data.shape))

for qrun in qruns:
    for qrun2 in qruns:
        row_tree = qrun.row_trees[-1]
        col_tree = qrun2.col_trees[-1]
        sr = tree_recon.recon_2d_sure(data,row_tree,col_tree)
        tree_recon.threshold_recon(sr,-1.0,1.0)
        sure_recon += sr
        for idx,epsilon in enumerate([0.02,0.01,0.005,0.002,0.001]):
            tr = tree_recon.recon_2d_haar_folder_size(data,
                                row_tree,col_tree,epsilon)
            tree_recon.threshold_recon(tr,-1.0,1.0)
            trunc_recons[idx] += tr
sure_recon /= (len(qruns)**2)
for idx,epsilon in enumerate([0.02,0.01,0.005,0.002,0.001]):
    trunc_recons[idx] /= (len(qruns)**2)

fig = plt.figure(figsize=(12,12))
fig.add_subplot(321)
cplot(sure_recon[row_order,:][:,col_order])
err = np.sum( (sure_recon-shuffled_means_matrix)**2 )
plt.title("SURE wavelet shrinkage err = {}".format(err))
for idx,epsilon in enumerate([0.02,0.01,0.005,0.002,0.001]):
    fig.add_subplot(3,2,idx+2)
    cplot(trunc_recons[idx][row_order,:][:,col_order])
    err = np.sum( (trunc_recons[idx]-shuffled_means_matrix)**2 )
    plt.title("Truncation $(\epsilon = {}$) err = {}".format(epsilon,err))
plt.tight_layout()
plt.show()
```



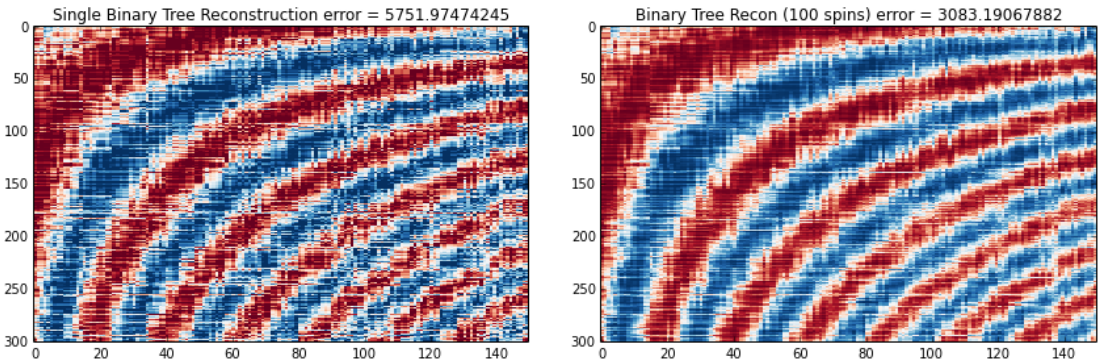
So we see here that spin cycling can remove many artifacts and improve the performance of partition tree-based estimators and the estimator based on SURE alike, reducing the error in this case by almost a third or more from a single run.

Binary Trees (and spin cycling)

We finish by presenting the outcomes of processing the entire questionnaire process using randomized binary trees. The overall reconstruction performance for these trees is poorer because the binary tree structure limits the accuracy of the model. However, the reconstructions still benefit significantly from spinning, with a near 50% reduction in error from 100 row-column tree pairs.

In [31]:
```
bal_constants = [1.0,1.3,1.5,1.8,2.1,1.6,1.5,2.1,1.6,1.8]
params_list = []
for bal_constant in bal_constants:
    kwargs = {}
    kwargs["n_iters"] = 2
    kwargs["threshold"] = 0.0
    kwargs["row_alpha"] = 1.0
    kwargs["col_alpha"] = 1.0
    kwargs["row_beta"] = 0.0
    kwargs["col_beta"] = 0.0
    kwargs["tree_constant"] = bal_constant
    params = questionnaire.PyQuestParams(questionnaire.INIT_AFF_COS_SIM,
                                         questionnaire.TREE_TYPE_BINARY,
                                         questionnaire.DUAL_EMD,
                                         questionnaire.DUAL_EMD,**kwargs)
    params_list.append(params)
qruns = []
for params in params_list:
    qruns.append(questionnaire.pyquest(data,params))
epsilon = 0.0015
binary_recons = np.zeros([len(qruns)**2] + list(data.shape))
idx = 0
for qrun in qruns:
    for qrun2 in qruns:
        row_tree = qrun.row_trees[-1]
        col_tree = qrun2.col_trees[-1]
        tr = tree_recon.recon_2d_haar_folder_size(data,
                                  row_tree,col_tree,epsilon)
        tree_recon.threshold_recon(tr,-1.0,1.0)
        binary_recons[idx] = tr
        idx += 1
br_spun = np.mean(binary_recons,axis=0)
```

In [32]:
```
fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
cplot(binary_recons[50][row_order,:][:,col_order])
err = np.sum( (binary_recons[50]-shuffled_means_matrix)**2 )
plt.title("Single Binary Tree Reconstruction error = {}".format(err))
fig.add_subplot(122)
cplot(br_spun[row_order,:][:,col_order])
err = np.sum( (br_spun-shuffled_means_matrix)**2 )
plt.title("Binary Tree Recon (100 spins) error = {}".format(err))
plt.tight_layout()
plt.show()
```



8.2 Science News

Organizing the Science News Dataset

In [1]:
```
from imports import *
np.random.seed(20140101)
```

The Science News dataset consists of a 1153×1047 matrix, representing the abstracts of a set of documents published in Science News. The columns of the matrix are 1047 documents on a variety of topics. The rows of the matrix are each a word that appears in one or more of the documents, and the data in each column of the matrix represents an empirical probability distribution on the words for a particular document. We would like to organize the documents and words jointly using the questionnaire algorithm; the intuition is that the words belong to related categories or topics, such that the EMD defined on those words will help to organize the documents. Further, some words are substitutable for each other; one author might prefer to use one synonym while another prefers a different one. If we can group such words into folders, we will find affinities between documents that might not be available naively.

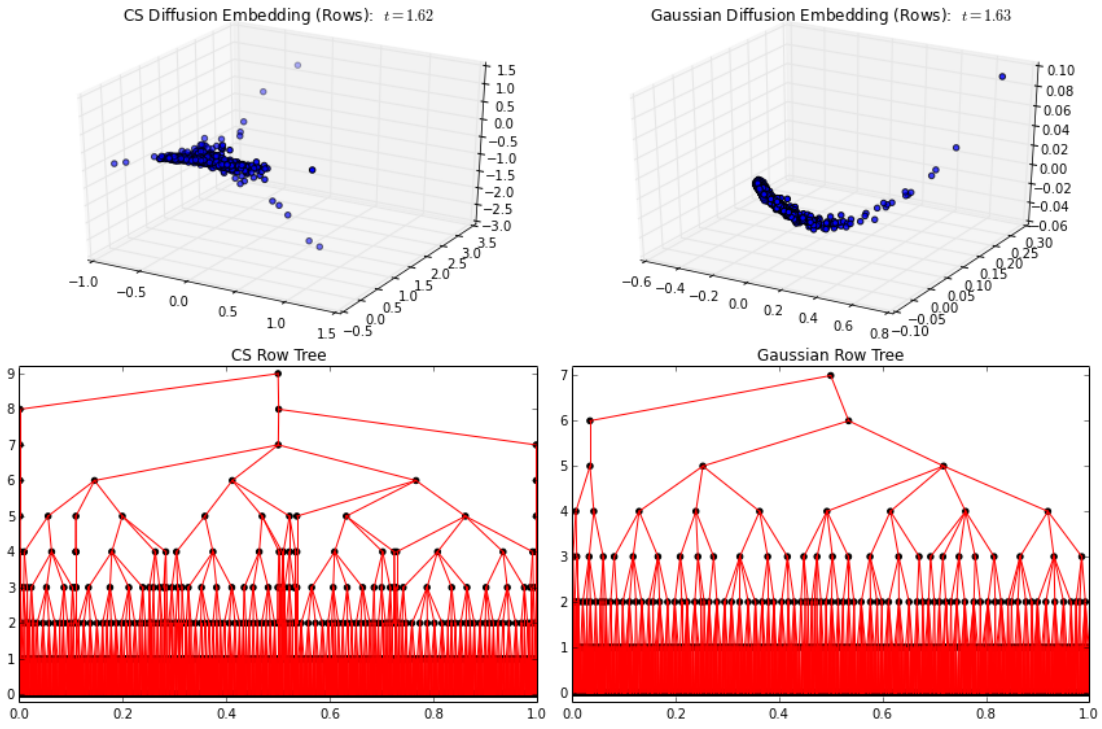
The words are preprocessed to remove common words like "the", etc, and words which appear in only one document, such as names. However, as we will detect, there are still quite a number of words which remain which are not terribly meaningful. Further, we will purposefully partially cripple the dataset by ignoring the probability distribution information and simply replacing any positive probability with a +1, while leaving zeros alone. Hence we will change the data into binary questionnaire type, but we will still be able to first of all detect outlier words/documents and then organize the documents such that in the embedding we will see reasonable structure among externally validated document types.

In [2]:
```
%run py_load_data.py sn
binary_data = np.ones(data.shape)*(data > 0.0)
m,n = data.shape
def remove_data(indices,direction):
    global data, binary_data
    if direction == "words":
        global words
        keep_indices = [x for x in xrange(m) if x not in indices]
        data = data[keep_indices,:]
        words = words[keep_indices]
        print "removed {} words".format(len(indices))
    elif direction == "docs":
        keep_indices = [x for x in xrange(n) if x not in indices]
        global doc_titles, doc_class
        data = data[:,keep_indices]
        doc_titles = doc_titles[keep_indices]
        doc_class = doc_class[keep_indices]
        print "removed {} docs".format(len(indices))
    binary_data = np.ones(data.shape)*(data > 0.0)
```

Outlier Detection

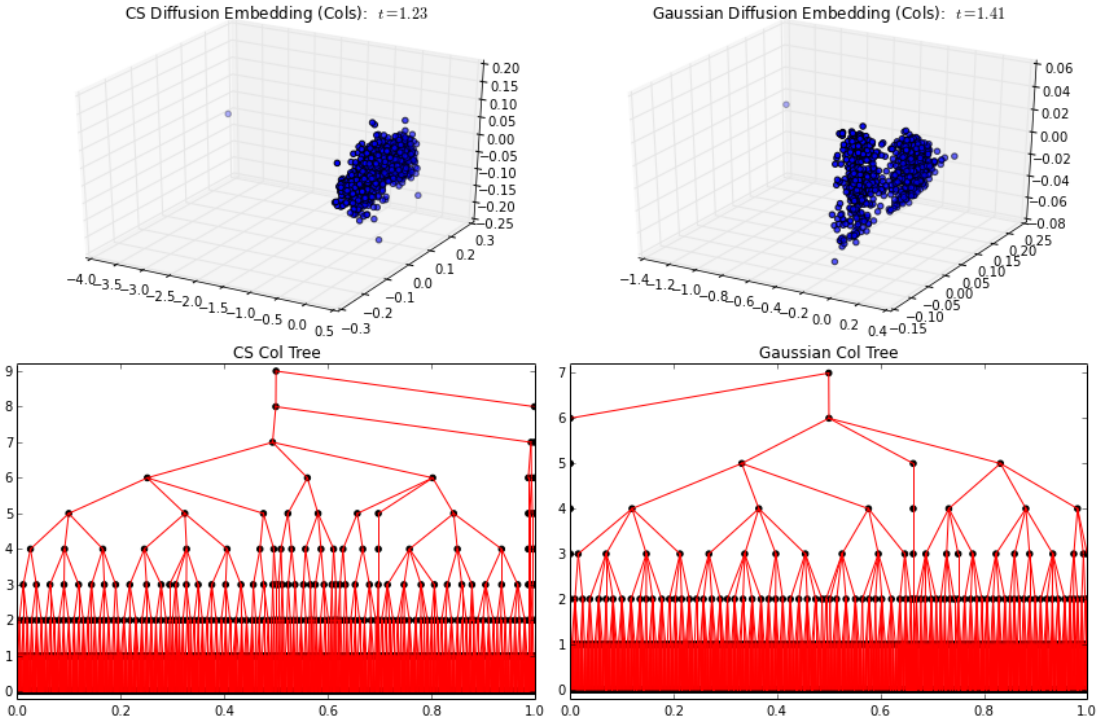
We begin by calculating an initial affinity on the rows using two methods: cosine similarity and a Gaussian kernel on the Euclidean distance between rows. We see that the Gaussian kernel organizes the data into a slightly nicer curve. Both embeddings, however, have points that are seemingly outliers. We will investigate this in a moment.

In [3]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(221,projection="3d")
cs_row_aff = affinity.mutual_cosine_similarity(binary_data.T)
cs_vecs, cs_vals = markov.markov_eigs(cs_row_aff,4)
plot_embedding(cs_vecs,cs_vals,ax=ax1,
               title="CS Diffusion Embedding (Rows): ")
ax2 = fig.add_subplot(222,projection="3d")
g_row_aff = affinity.gaussian_euclidean(binary_data.T,knn=10)
g_vecs, g_vals = markov.markov_eigs(g_row_aff,4)
plot_embedding(g_vecs,g_vals,ax=ax2,
               title="Gaussian Diffusion Embedding (Rows): ")
fig.add_subplot(223)
cs_row_tree = flex_tree_build.flex_tree_diffusion(cs_row_aff,1.0)
plot_tree(cs_row_tree,title="CS Row Tree")
fig.add_subplot(224)
g_row_tree = flex_tree_build.flex_tree_diffusion(g_row_aff,1.0)
plot_tree(g_row_tree,title="Gaussian Row Tree")
plt.tight_layout()
plt.show()
```



We do the same thing for the columns. Notice that both of the column embeddings have a single outlier which is quite far away from the main body of points.

In [4]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(221,projection="3d")
cs_col_aff = affinity.mutual_cosine_similarity(binary_data)
cs_vecs, cs_vals = markov.markov_eigs(cs_col_aff,4)
plot_embedding(cs_vecs,cs_vals,ax=ax1,
               title="CS Diffusion Embedding (Cols): ")
ax2 = fig.add_subplot(222,projection="3d")
g_col_aff = affinity.gaussian_euclidean(binary_data,knn=10)
g_vecs, g_vals = markov.markov_eigs(g_col_aff,4)
plot_embedding(g_vecs,g_vals,ax=ax2,
               title="Gaussian Diffusion Embedding (Cols): ")
fig.add_subplot(223)
cs_col_tree = flex_tree_build.flex_tree_diffusion(cs_col_aff,1.0)
plot_tree(cs_col_tree,title="CS Col Tree")
fig.add_subplot(224)
g_col_tree = flex_tree_build.flex_tree_diffusion(g_col_aff,1.0)
plot_tree(g_col_tree,title="Gaussian Col Tree")
plt.tight_layout()
plt.show()
```



The title of the outlier document is "Science News of the Year 2001 Compiled." It's not that surprising that such a document would be an outlier in our dataset, as it is likely to contain a wide variety of words from all different disciplines, and so be unlike any other document. We will remove this document from the dataset to facilitate a cleaner analysis.

In [5]:
```
print np.array([x.size for x in g_col_tree.children])
print doc_titles[g_col_tree.children[0].elements]
```

```
[   1 1046]
[u'10393. :  , Dec. 22, 2001 > Science News of the Year 2001 Compiled']
```

We can repeat this analysis on the row trees, and identify the words which are outliers to the main group of words.

In [6]:
```
print np.array([x.size for x in g_row_tree.children])
print ",".join(words[g_row_tree.children[0].elements])
```

```
[  80 1073]
year,university,study,scientist,people,cell,time,team,science,
group,system,animal,percent,work,found,note,human,state,
data,million,sn,week,effect,world,test,problem,colleague,
during,finding,body,life,suggest,form,number,month,evidence,
ago,result,using,type,national,day,change,different,known,
produce,part,cause,case,laboratory,find,point,center,early,
remain,re,down,don,california,around,without,get,similar,
provide,institute,past,nature,help,amount,too,need,look,
little,example,think,although,add,recent,explain,come
```

These words are quite vague in meaning and not very discriminative of different documents from each other. Most are likely to be found in almost any document, such as "university" or "science". Words like "california" may not appear in many documents, but will be unhelpful in distinguishing the meanings of various documents. So we will remove these words from the dataset as well.

In [7]:
```
remove_data(g_row_tree.children[0].elements,"words")
remove_data(g_col_tree.children[0].elements,"docs")
print binary_data.shape
```

```
removed 80 words
removed 1 docs
(1073L, 1046L)
```

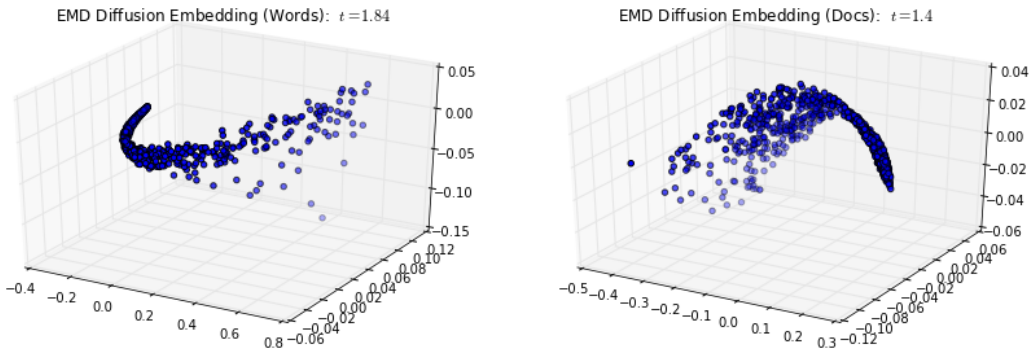
Organization

Now let us run the questionnaire process on the remaining data.

In [8]:
```
kwargs = {}
kwargs["threshold"] = 0.0
kwargs["row_alpha"] = 0.0
kwargs["col_alpha"] = 0.0
kwargs["row_beta"] = 1.0
kwargs["col_beta"] = 1.0
kwargs["tree_constant"] = 1.0
kwargs["n_iters"] = 3
params = questionnaire.PyQuestParams(questionnaire.INIT_AFF_GAUSSIAN,
                                     questionnaire.TREE_TYPE_FLEXIBLE,
                                     questionnaire.DUAL_EMD,
                                     questionnaire.DUAL_EMD,**kwargs)
```

In [9]:
```
qrun = questionnaire.pyquest(binary_data,params)
row_tree = qrun.row_trees[-1]
col_tree = qrun.col_trees[-1]
row_emd = dual_affinity.calc_emd(binary_data.T,col_tree,
                                 params.row_alpha,params.row_beta)
col_emd = dual_affinity.calc_emd(binary_data,row_tree,
                                 params.col_alpha,params.col_beta)
row_aff = dual_affinity.emd_dual_aff(row_emd)
col_aff = dual_affinity.emd_dual_aff(col_emd)
row_vecs,row_vals = markov.markov_eigs(row_aff,12)
col_vecs,col_vals = markov.markov_eigs(col_aff,12)
```

In [10]:
```
fig = plt.figure(figsize=(12,4))
ax1 = fig.add_subplot(121,projection="3d")
plot_embedding(row_vecs,row_vals,ax=ax1,
                            title="EMD Diffusion Embedding (Words): ")
ax2 = fig.add_subplot(122,projection="3d")
plot_embedding(col_vecs,col_vals,ax=ax2,
                            title="EMD Diffusion Embedding (Docs): ")
plt.tight_layout()
plt.show()
```



Compared to embeddings based on the raw intial affinities we obtained earlier, the EMD iterated embeddings are much smoother and “nicer” looking.

The following code will open an external application which will allow the user to browse groups of words or documents, a process hard to simulate on the page but useful for subjectively evaluating the quality of word or document folders.

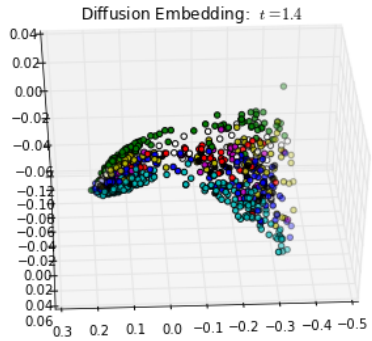
In [11]:
```
#viewer_files.write_tree_viewer("words.vw",row_tree,row_vecs,row_vals,words)
#viewer_files.write_tree_viewer("docs.vw",col_tree,col_vecs,col_vals,doc_titles)
#%run tree_viewer2.py words.vw
#%run tree_viewer2.py docs.vw
```

We can also color the nodes of the diffusion embedding with eight categories of document supplied externally. Because the classification of document types is somewhat inherently murky and we weren't working to predict this variable, the eight categories, while very loosely running in stripes along the embedding, aren't really cleanly organized.

In [12]:
```
COLORS = "mgkybrcw"
nodecolors = [COLORS[x-1] for x in doc_class]
for x,y in zip(COLORS,score_titles):
    print x,y
```

```
m Anth
g Space
k Behav
y Env
b Life
r MathCS
c Med
w PhysTech
```

In [13]:
```
plot_embedding(col_vecs,col_vals,nodecolors=nodecolors,azim=87,elev=43)
plt.tight_layout()
plt.show()
```

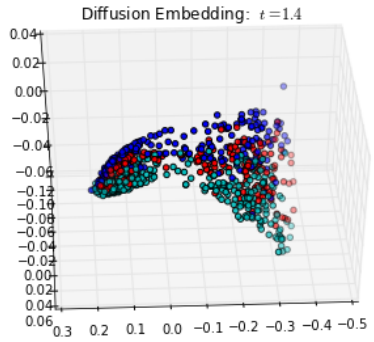


However, if we combine the eight categories into three larger categories: (Space,MathCS,PhysTech), (Anth,Behav,Env), and (Life,Med), and paint the embedding in that way, we can see that the larger categories (which probably overlap each other significantly in reality) are painted in recognizable stripes across the embedding.

In [14]:
```
COLORS = "rbrrcbcb"
nodecolors = [COLORS[x-1] for x in doc_class]
for x,y in zip(COLORS,score_titles):
    print x,y
```

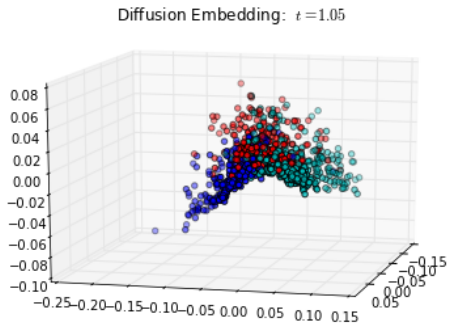
```
r Anth
b Space
r Behav
r Env
c Life
b MathCS
c Med
b PhysTech
```

In [15]:
```
plot_embedding(col_vecs,col_vals,nodecolors=nodecolors,azim=87,elev=43)
plt.tight_layout()
plt.show()
```



We can additionally look at embeddings that are the result of different parameters. For example, suppose that we let β on the columns be -1 instead of 1. Then we are putting a lot more weight in EMD on individual words and smaller groups of words. The resulting embedding is a lot less smooth, but the category painting reveals some of the same structure in a different shape:

In [16]:
```
params.col_beta = -1.0
qrun = questionnaire.pyquest(binary_data,params)
row_tree = qrun.row_trees[-1]
col_tree = qrun.col_trees[-1]
row_emd = dual_affinity.calc_emd(binary_data.T,col_tree,
                                 params.row_alpha,params.row_beta)
col_emd = dual_affinity.calc_emd(binary_data,row_tree,
                                 params.col_alpha,params.col_beta)
row_aff = dual_affinity.emd_dual_aff(row_emd)
col_aff = dual_affinity.emd_dual_aff(col_emd)
row_vecs,row_vals = markov.markov_eigs(row_aff,12)
col_vecs,col_vals = markov.markov_eigs(col_aff,12)
plot_embedding(col_vecs,col_vals,nodecolors=nodecolors,
                          azim=14,elev=13)
plt.tight_layout()
plt.show()
```



8.3 MMPI2

The next dataset we will consider as an example is the Minnesota Multiphasic Psychological Inventory-2 (MMPI2), which is the most widely used standardized psychometric test of adult personality and psychopathology. The dataset consists of a sample of 2,428 people's answers to 567 questions with binary responses in $\{1, -1\}$. The questions are a series of statements, and the answers are the test taker's indication of agreement with the statements. Some example statements are "Evil spirit possess me at times." or "It is hard for me to accept compliments." Certain parts of the questionnaire algorithm from 5.3 involve operations on "folder averages" or averages on clusters of points. From the example questions above, it is clear that at least in some cases, the questions have some notion of **polarity**; that is, we could easily ask the question in a different way and reverse the sign of the answers. For example, we could ask "It is easy for me to accept compliments." and probably obtain nearly the same set of answers, with just the sign flipped. But it might be the case that questions which point in opposite directions might have strong predictive power or affinity to each other that we might lose. Additionally, because we are using folder averages, especially in EMD, we might be limiting ourselves to building associations between questions which have the same polarity. Or we might hamstring our ability to take meaning from folder averages in cases where the folders could contain questions of different polarity. If two questions point in opposite directions, it is important to be able to distinguish between the answer set (+1,-1) and (-1,+1), for obvious reasons.

Now it would be a desirable feature of any data organization scheme to be able to positively associate questions which are quite similar in meaning to each other. But in this example, the questions are practically opposites. Hence we would like to find a way to "reverse the polarity" of some of the questions in

order to facilitate this. We did the following:

- "Doubled" the questionnaire by appending a copy of the data matrix with the signs reversed ("anti-questions").
- Calculated an affinity A on the rows by taking their cosine similarity. Naturally the rows had perfect anti-similarity to their anti-questions.
- Calculated the diffusion on A.
- We took the first nontrivial eigenvector of the diffusion. The coefficients corresponding to each row of this vector had the following structure:
  - The coefficient values were mirrored across zero.
  - The set of values that was positive contained exactly one of each question/anti-question pair.

The set of questions with positive eigenvector coefficients consisted of 416 of the original questions, and 151 anti-questions. Based on visual inspection of the types of questions that were reversed, we concluded that in so far as it was possible, this process had in some sense "depolarized" the MMPI questionnaire; when we process the modified data set, a positive answer to each question each essentially points in the same direction. Now there is no guarantee that this process will work on all datasets – in fact it is easy to construct data where this will be impossible. But empirically, depolarizing the data in this way removed a significant difficulty that had been faced in processing the data previously. In the following notebook, we explore the reconstruction process, and demonstrate the process of building question trees on the MMPI2 dataset.

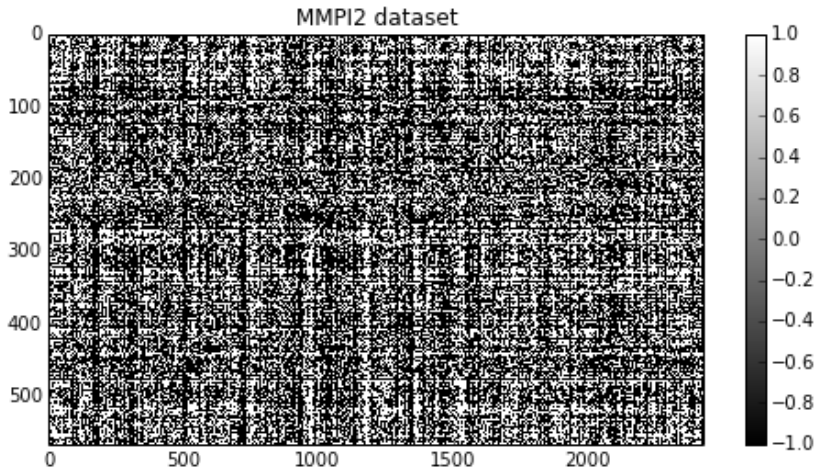
Reconstructing the MMPI and the Adaptive Questionnaire

The MMPI2, as described in the main text, consists of 567 yes/no questions. Our dataset, which unfortunately for legal reasons cannot be distributed, contains the responses of 2428 individuals to the questionnaire. In this notebook we will process the questionnaire, reconstruct a probability field underlying the data, and then demonstrate the use of adaptive question trees to classify and predict the probability field of a new person using a small subset of the questions.

In [1]:
```
from imports import *
%run py_load_data.py de
```

We begin with the following dataset: 567×2428 yes/no answers (+1 and -1, with zeroes for questions that were not answered).

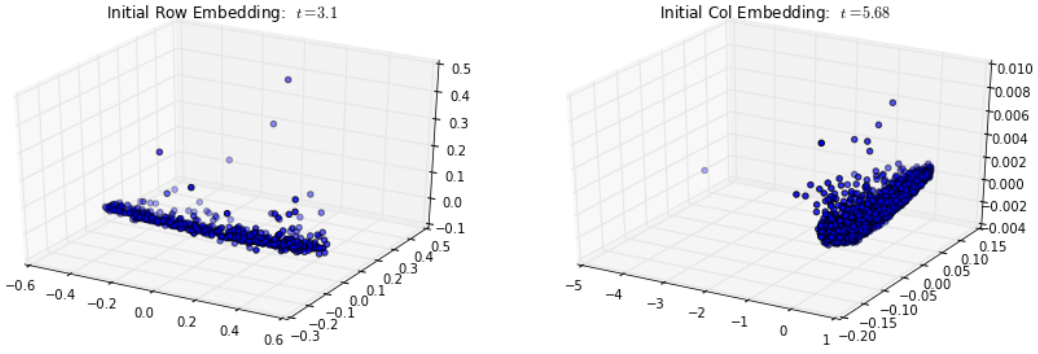
In [2]:
```
fig = plt.figure(figsize = (8,4))
bwplot2(data)
plt.colorbar()
plt.title("MMPI2 dataset")
plt.show()
```



Organization

We begin by calculating affinities on the rows and columns, using cosine similarity. Then we calculate the diffusion maps, embed the data, and generate trees on the rows and columns. We then run the entire questionnaire process for two iterations, and look at the embeddings for each iteration:

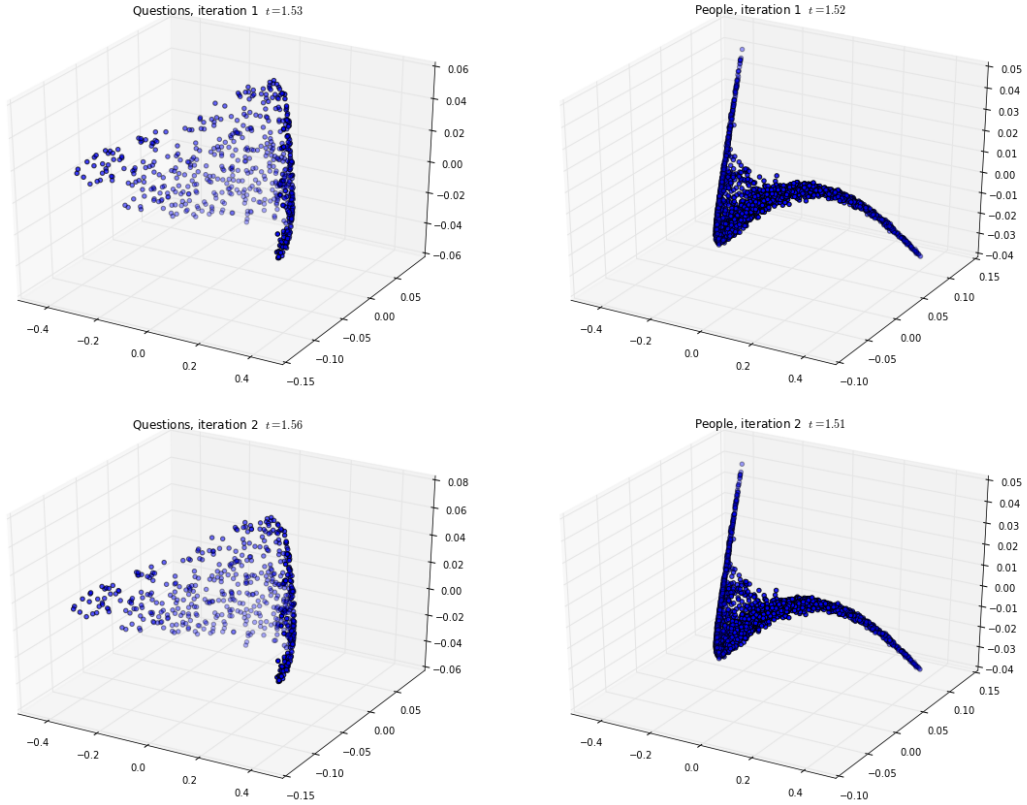
In [3]:
```
init_row_aff = affinity.mutual_cosine_similarity(data.T,threshold=0.0)
init_col_aff = affinity.mutual_cosine_similarity(data,threshold=0.0)
init_row_vecs,init_row_vals = markov.markov_eigs(init_row_aff,12)
init_col_vecs,init_col_vals = markov.markov_eigs(init_col_aff,12)
fig = plt.figure(figsize=(12,4))
ax = fig.add_subplot(121,projection="3d")
plot_embedding(init_row_vecs,init_row_vals,ax=ax,
               title="Initial Row Embedding: ")
ax2 = fig.add_subplot(122,projection="3d")
plot_embedding(init_col_vecs,init_col_vals,ax=ax2,
               title="Initial Col Embedding: ")
plt.tight_layout()
plt.show()
```



In [4]:
```
kwargs = {}
kwargs["threshold"] = 0.0
kwargs["row_alpha"] = 0.0
kwargs["col_alpha"] = 0.0
kwargs["row_beta"] = 1.0
kwargs["col_beta"] = 1.0
kwargs["tree_constant"] = 1.0
kwargs["n_iters"] = 2
params = questionnaire.PyQuestParams(questionnaire.INIT_AFF_COS_SIM,
                                     questionnaire.TREE_TYPE_FLEXIBLE,
                                     questionnaire.DUAL_EMD,
                                     questionnaire.DUAL_EMD,**kwargs)
qrun = questionnaire.pyquest(data,params)
```

In [5]:

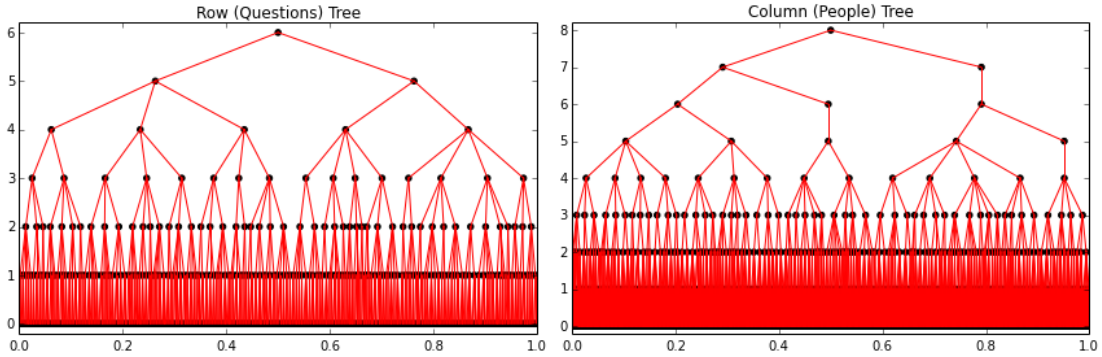
```
fig = plt.figure(figsize=(16,12))
m=len(qrun.col_trees)
axes = {}
for idx,col_tree in enumerate(qrun.col_trees):
    axes[2*idx+1] = fig.add_subplot(m,2,2*idx+1,projection="3d")
    row_emd = dual_affinity.calc_emd(data.T,col_tree,params.row_alpha,
                                     params.row_beta)
    row_aff = dual_affinity.emd_dual_aff(row_emd)
    row_vecs,row_vals = markov.markov_eigs(row_aff,4)
    plot_embedding(row_vecs,row_vals,ax=axes[2*idx+1],
                   title="Questions, iteration {} ".format(idx+1))
for idx,row_tree in enumerate(qrun.row_trees[1:]):
    axes[2*idx] = fig.add_subplot(m,2,2*idx+2,projection="3d")
    col_emd = dual_affinity.calc_emd(data,row_tree,params.col_alpha,
                                     params.col_beta)
    col_aff = dual_affinity.emd_dual_aff(col_emd)
    col_vecs,col_vals = markov.markov_eigs(col_aff,4)
    plot_embedding(col_vecs,col_vals,ax=axes[2*idx],
                   title="People, iteration {} ".format(idx+1))
plt.tight_layout()
plt.show()
```



The iteration process converges pretty rapidly to a consistent set of embeddings. Notice that the people embedding takes an interesting form; that of a long curve that is stretched/elongated in a different direction toward the middle, but is relatively thin at either end.

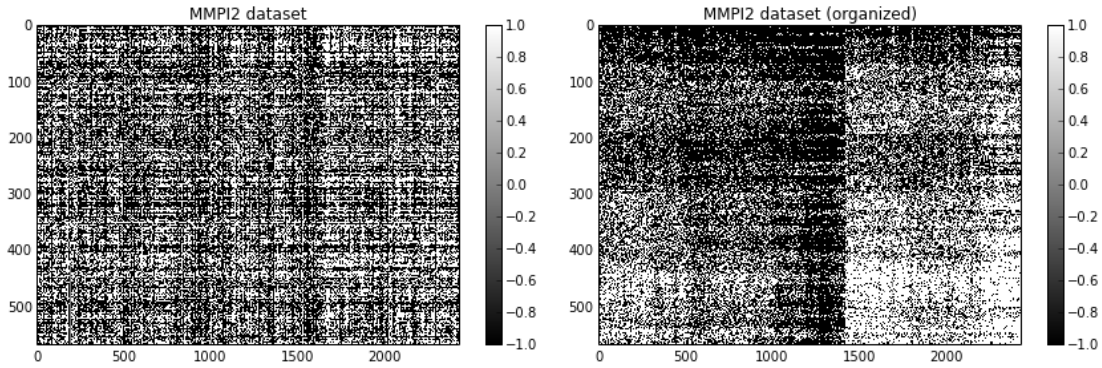
We plot the final trees:

In [6]:
```
row_tree = qrun.row_trees[-1]
col_tree = qrun.col_trees[-1]
fig = plt.figure(figsize=(12,4))
fig.add_subplot(121)
plot_tree(row_tree,title="Row (Questions) Tree")
fig.add_subplot(122)
plot_tree(col_tree,title="Column (People) Tree")
plt.tight_layout()
plt.show()
```



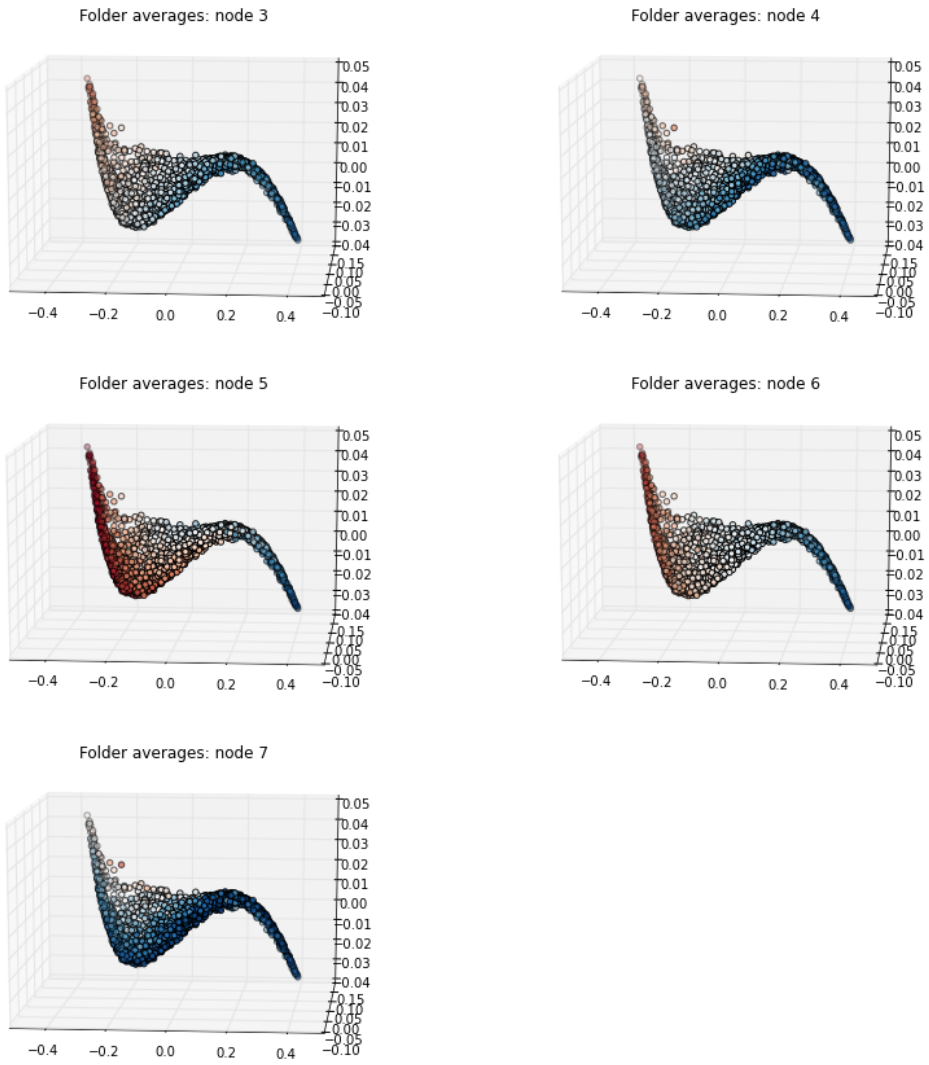
We plot the original data set, organized by these trees:

In [7]:
```
fig = plt.figure(figsize = (12,4))
fig.add_subplot(121)
bwplot2(data,colorbar=True,title="MMPI2 dataset")
fig.add_subplot(122)
bwplot2(barcode.organize_folders(row_tree,col_tree,data),
        colorbar=True,title="MMPI2 dataset (organized)")
plt.tight_layout()
plt.show()
```



We can see that the organization on the right exhibits quite a bit more structure than the raw data on the left. We consider the five groups at level 4 on the row tree. We paint the people embedding by the average response to each of those five groups (on a scale from red (yes) to blue (no)).

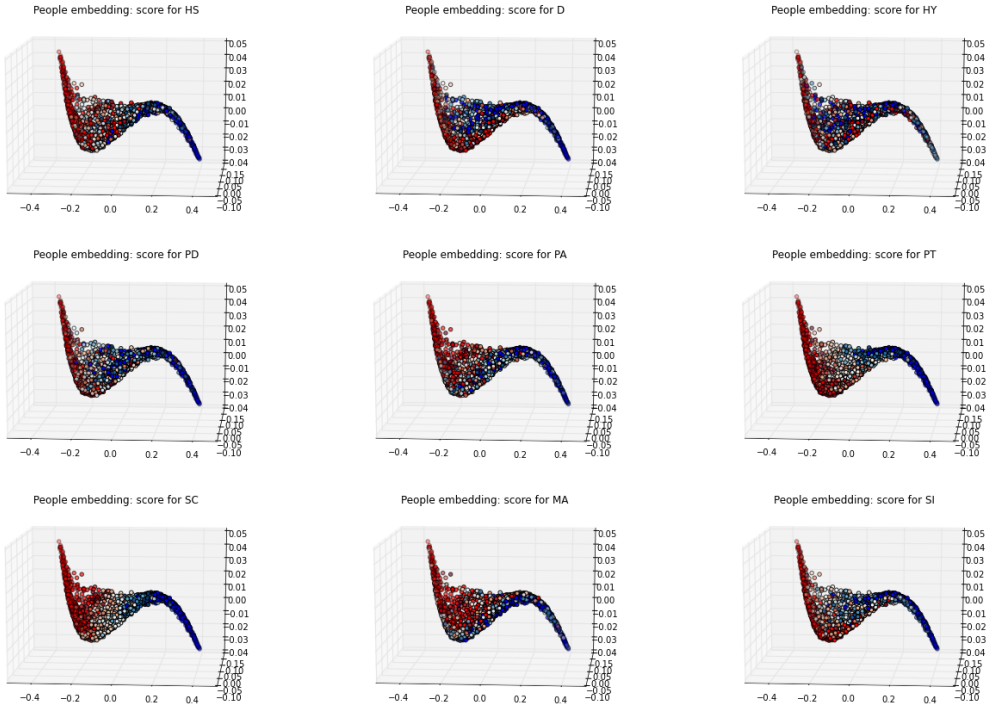
In [8]:
```
fig = plt.figure(figsize=(12,12))
avgs = tree_util.tree_averages(data,row_tree)
axes = {}
for idx,node in enumerate(sorted(row_tree.dfs_level(3),
                                 key=lambda x: x.idx)):
    axes[idx] = fig.add_subplot(3,2,idx+1,projection="3d")
    nodecolors = avgs[node.idx,:]
    plot_embedding(col_vecs,col_vals,nodecolors=nodecolors,
                   title="Folder averages: node {}".format(node.idx),
                   nodt=True,azim=-85,elev=12,ax=axes[idx])
plt.tight_layout()
plt.show()
```



The average response varies from question group to question group, but a clear pattern of decreasing “yes” answers can been seen from tail to tail.

We have an additional piece of external data, which are scores on each of nine different psychological disorders that the MMPI2 is designed to measure. We z-score the raw scores, and color the embedding by each of the scores in turn:

In [9]:
```
fig = plt.figure(figsize=(18,12))
avgs = tree_util.tree_averages(data,row_tree)
axes = {}
s=p_score_descs
for idx in xrange(9):
    axes[idx] = fig.add_subplot(3,3,idx+1,projection="3d")
    nodecolors = avgs[node.idx,:]
    plot_embedding(col_vecs,col_vals,nodecolors=p_scores[idx,:],
                   title="People embedding: score for {}".format(s[idx]),
                   nodt=True,azim=-85,elev=12,ax=axes[idx])
plt.tight_layout()
plt.show()
```



While the embedding does not neatly divide the disorder scores into bands, there is still a clear pattern of high scores on one tail of the embedding and low scores on the other, with varying degrees of mixture in the larger ball in the center.

As one final example that the organization we obtain from the questionnaire process is at least reasonable, we consider two groups of questions that are grouped together as folders in the row tree.

In [10]:
```
group1 = [x.elements for x in
          row_tree.dfs_level(row_tree.tree_depth-1) if 23 in x.elements]
for x in [q_descs[y] for y in group1]:
    print "\n".join(x)
print "****"
group2 = [x.elements for x in
          row_tree.dfs_level(row_tree.tree_depth-2) if 36 in x.elements]
for x in [q_descs[y] for y in group2]:
    print "\n".join(x)
```

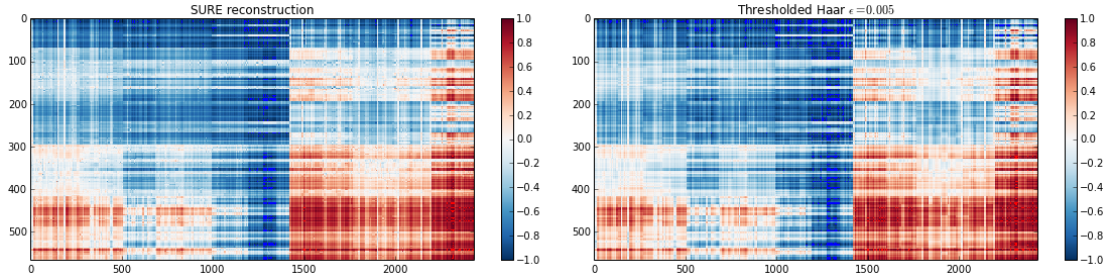
```
24. Evil spirits possess me at times.
72. My soul sometimes leaves my body.
96. I see things or animals or people around me that others do not see.
228. There are persons who are trying to steal my thoughts and ideas.
468. I am afraid of being alone in a wide-open place.
****
32. I have had very peculiar and strange experiences.
37. At times I feel like smashing things.
309. I usually have to stop and think before I act even in small matters.
517. I find it difficult to hold down a job.
562. It is hard for me to accept compliments.
```

By inspection, it certainly seems that these questions are quite reasonably clustered together: the first group of questions seems to broadly relate to the perception of supernatural events, while the second group seems to relate to difficulties with self-awareness.

Reconstruction

Next we will try to reconstruct the underlying probability field that gave rise to the MMPI2 dataset. We begin by considering two main methods: the wavelet shrinkage reconstruction based on SURE, and the reconstruction based on truncation of coefficients corresponding to folders of less than a certain size.

In [11]:
```
sure_recon = tree_recon.recon_2d_sure(data,row_tree,col_tree)
recon = tree_recon.recon_2d_haar_folder_size(data,row_tree,col_tree,0.005)
fig = plt.figure(figsize=(16,4))
fig.add_subplot(121)
cplot(barcode.organize_folders(row_tree,col_tree,sure_recon),
      colorbar=True,title="SURE reconstruction")
fig.add_subplot(122)
cplot(barcode.organize_folders(row_tree,col_tree,recon),
      colorbar=True,title="Thresholded Haar $\epsilon = 0.005$")
plt.tight_layout()
plt.show()
```

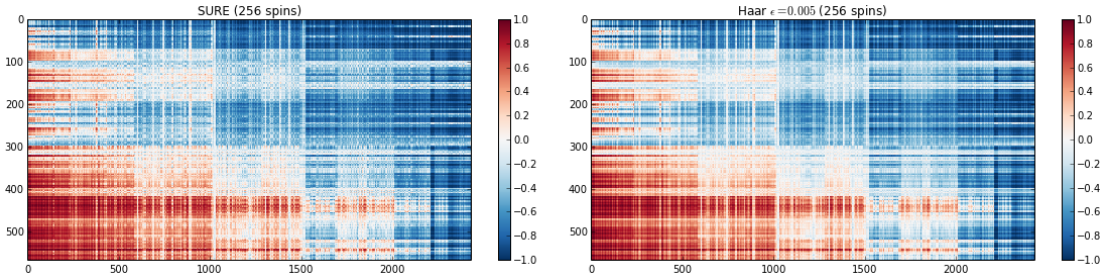


Next we spin cycle. We will generate sets of trees, using different methods. We will generate flexible trees using eight different tree constants. We will also generate randomized binary trees, using four different balance constants, sampling each twice. In all this will constitute 16 sets of trees. Then we will cross-match these trees to produce 256 pairs of trees, and reconstruct using each method, taking the average over all 256 tree-pairs as the probability field.

In [12]:
```
np.random.seed(20140401)
tree_constants = [0.2,0.4,0.6,0.8,1.0,1.2,1.4,1.5]
balance_constants = [1.1,1.5,2.0,2.5]
params_list = []
for tree_constant in tree_constants:
    kwargs = {}
    kwargs["n_iters"] = 2
    kwargs["threshold"] = 0.0
    kwargs["row_alpha"] = 0.0
    kwargs["col_alpha"] = 0.0
    kwargs["row_beta"] = 1.0
    kwargs["col_beta"] = 1.0
    kwargs["tree_constant"] = tree_constant
    params = questionnaire.PyQuestParams(questionnaire.INIT_AFF_COS_SIM,
                                         questionnaire.TREE_TYPE_FLEXIBLE,
                                         questionnaire.DUAL_EMD,
                                         questionnaire.DUAL_EMD,**kwargs)
    params_list.append(params)
for balance_constant in balance_constants:
    kwargs["bal_constant"] = balance_constant
    params = questionnaire.PyQuestParams(questionnaire.INIT_AFF_COS_SIM,
                                         questionnaire.TREE_TYPE_BINARY,
                                         questionnaire.DUAL_EMD,
                                         questionnaire.DUAL_EMD,**kwargs)
    params_list.append(params)
    params_list.append(params)
qruns = []
for params in params_list:
    qruns.append(questionnaire.pyquest(data,params))
```

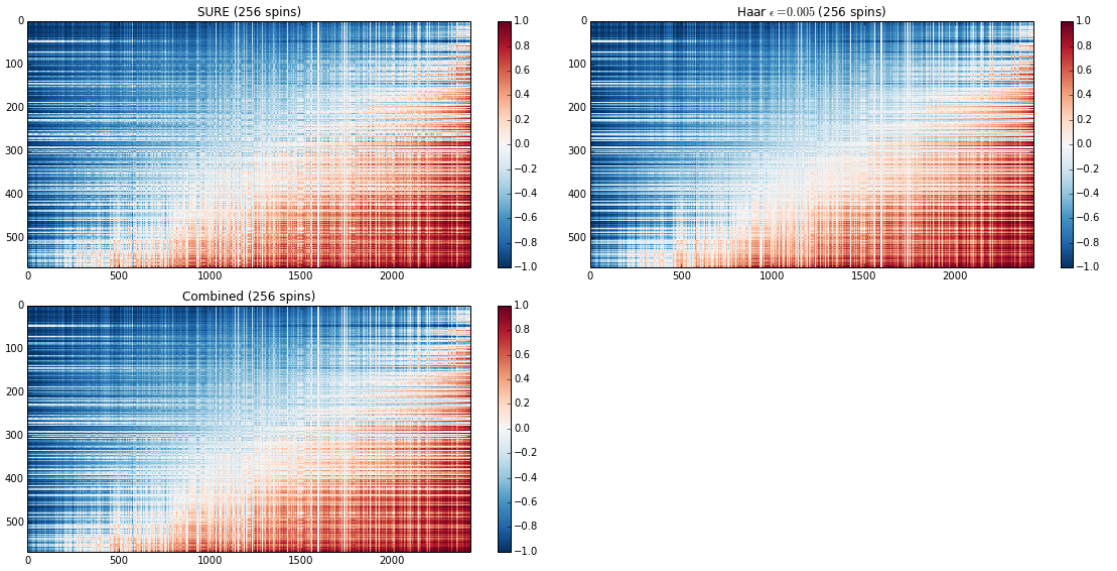
In [13]:
```
total_sure_recon = np.zeros(data.shape)
total_haar_recon = np.zeros(data.shape)
for qrun in qruns:
    for qrun2 in qruns:
        row_tree = qrun.row_trees[-1]
        col_tree = qrun2.col_trees[-1]
        sure_recon = tree_recon.recon_2d_sure(data,row_tree,col_tree)
        tree_recon.threshold_recon(sure_recon,-1.0,1.0)
        total_sure_recon += sure_recon
        haar_recon = tree_recon.recon_2d_haar_folder_size(
                            data,row_tree,col_tree,0.005)
        tree_recon.threshold_recon(haar_recon,-1.0,1.0)
        total_haar_recon += haar_recon
total_sure_recon /= 1.0*(len(qruns)**2)
total_haar_recon /= 1.0*(len(qruns)**2)
total_recon = (total_sure_recon + total_haar_recon)/2.0
```

In [14]:
```
row_tree = qruns[4].row_trees[-1]
col_tree = qruns[4].col_trees[-1]
fig = plt.figure(figsize=(16,4))
fig.add_subplot(121)
cplot(barcode.organize_folders(row_tree,col_tree,total_sure_recon),
      colorbar=True,title="SURE ({} spins)".format(len(qruns)**2))
fig.add_subplot(122)
cplot(barcode.organize_folders(row_tree,col_tree,total_haar_recon),
      colorbar=True,
      title="Haar $\epsilon = 0.005$ ({} spins)".format(len(qruns)**2))
plt.tight_layout()
plt.show()
```



Because the suborganizations of the trees are somewhat arbitrary, we permute the rows and columns to coincide with the empirical mean response by each person and on each question:

In [15]:
```
row_order = np.sum(data,axis=1).argsort()
col_order = np.sum(data,axis=0).argsort()

fig = plt.figure(figsize=(16,8))
fig.add_subplot(221)
cplot(total_sure_recon[row_order,:][:,col_order],colorbar=True,
      title="SURE ({} spins)".format(len(qruns)**2))
fig.add_subplot(222)
cplot(total_haar_recon[row_order,:][:,col_order],colorbar=True,
      title="Haar $\epsilon = 0.005$ ({} spins)".format(len(qruns)**2))
fig.add_subplot(223)
cplot(total_recon[row_order,:][:,col_order],colorbar=True,
      title="Combined ({} spins)".format(len(qruns)**2))
plt.tight_layout()
plt.show()
```



The Adaptive Questionnaire

Next we turn to the following problem. Suppose we have a training set of data, and the model assumptions of a smooth probability field are assumed to hold. Then we want to take a new person who has not answered any of the questions, and predict her probability field by asking an adaptively chosen small subset of the questions. The first step is to define a training set.

In [16]:
```
indices = np.random.rand(data.shape[1]).argsort()
train_data = data[:,indices[0:2000]]
test_data = data[:,indices[2000:]]
```

First we calculate the probability field for the training set, using the same methodology as above. Here we combine the two methods (SURE and Haar truncation) to provide even more artifact smoothing.

In [17]:
```
train_qruns = []
for params in params_list:
    train_qruns.append(questionnaire.pyquest(train_data,params))
```

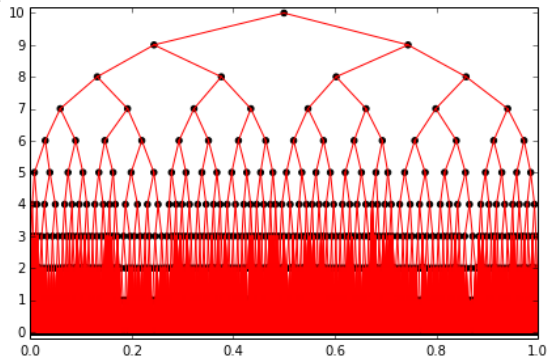
In [18]:
```
train_recon = np.zeros(train_data.shape)
for qrun in train_qruns:
    for qrun2 in train_qruns:
        row_tree = qrun.row_trees[-1]
        col_tree = qrun2.col_trees[-1]
        sure_recon = tree_recon.recon_2d_sure(
                        train_data,row_tree,col_tree)
        tree_recon.threshold_recon(sure_recon,-1.0,1.0)
        train_recon += sure_recon
        haar_recon = tree_recon.recon_2d_haar_folder_size(
                        train_data,row_tree,col_tree,0.005)
        tree_recon.threshold_recon(haar_recon,-1.0,1.0)
        train_recon += haar_recon
train_recon /= 2.0*(len(qruns)**2)
```

Next we build a **question tree** on the people, by analyzing the training set. The method by which this is computed is fully described in the main text, but the main idea is this. At each node, starting with the root, we calculate the diffusion on a graph built on the people at that node. We take the first non-trivial eigenvector of that node and then we use the LASSO method to identify a small subset of questions we can ask in order to linearly predict the eigenvector coordinates for each column. Then we take the original data, predict the eigenvector coordinates, and split the node into two children based on the sign of the predicted eigenvector coordinates.

In [19]:
```
row_tree = qruns[4].row_trees[-1]
col_tree = qruns[4].col_trees[-1]
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    q_tree = question_tree.mtree(train_data,row_tree)
```

Here is the question tree.

In [20]:
```
plot_tree(q_tree)
plt.tight_layout()
plt.show()
```



Now we take the test set. We pretend that we do not have any answers from these people. Then we locate them within the tree by asking them the appropriate questions at each node, calculating the predicted value of the linear model at that node, and then proceeding down the indicated branch of the tree. When we reach a small folder near the bottom of the tree, we take their predicted probability field values to be the mean of the folder in which they have been located.

In [21]:
```
total_recon = (total_sure_recon + total_haar_recon)/2.0
pred_data = np.zeros(test_data.shape)
folders = np.zeros(test_data.shape[1],np.int)
for cidx,colidx in enumerate(indices[2000:]):
    v = data[:,colidx]
    cur_node = q_tree
    while hasattr(cur_node,"lm"):
        peig = cur_node.lm.predict(v[cur_node.active])
        if peig > 0.0:
            cur_node = cur_node.children[1]
        else:
            cur_node = cur_node.children[0]
    pred_data[:,cidx] = np.mean(train_recon[:,cur_node.elements],axis=1)
    folders[cidx] = cur_node.idx
```

Now we can compare these predicted probability field values to the fields calculated in the probability field for the entire dataset. Taking the absolute l_1 error in the probabilities, we find that our method, which asks only at most 50 (and usually fewer) questions, can reconstruct the probability field for an unknown person with an average error per question of only 0.063. Note that this is not like a classifier or a standard statistical prediction task, where we are trying to use a set of independent data to predict a single value or a single class membership. Instead, we are asking less than 10% of the questions on the questionnaire, and inferring the probability field underlying the answers to the other 90+% of the questions all at once.

In [22]:
```
diffs = pred_data - total_recon[:,indices[2000:]]
print "{:3.2%}".format(np.mean(np.abs(diffs))/2.0)
```

```
6.32%
```

References

[1] F. Abramovich, Y. Benjamini, D. Donoho, I. Johnstone, Adapting to unknown sparsity by controlling the false discovery rate, The Annals of Statistics 34, (2), pp. 584-653, 2006.

[2] Y. Benjamini, Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing". Journal of the Royal Statistical Society, Series B 57 (1): pp 289–300. 1995.

[3] F. Chung, Spectral Graph Theory, in: CBNS-AMS, vol. 92, Amer. Math. Soc., Providence, RI, 1992.

[4] R. Coifman et al, Geometric diffusion as a tool for harmonic analysis and structure definition of data, parts I and II. Proc. Nat. Acad. Sci., 102 (21), pp.7426–7437, 2005.

[5] R. Coifman, S. Lafon, Diffusion maps, Appl. Comput. Harmon. Anal. 21 (1) , pp.5–30, 2006.

[6] R. Coifman, M. Gavish, Harmonic analysis of digital data bases, in: J. Cohen, A.I. Zayed (Eds.), Wavelets and Multiscale Analysis, Birkhäuser, pp. 161–197, 2011.

[7] R. Coifman, M. Maggioni, S.W. Zucker, I.G. Kevrekidis, Geometric diffusions for the analysis of data from sensor networks, Curr. Opin. Neurobiol. 15, 2005.

[8] G. David, A. Averbuch, Hierarchical data organization, clustering and denoising via localized diffusion folders, Appl. Comput. Harmon. Anal. 33 pp1–23, 2012.

[9] D. Donoho, I. Johnstone, Adapting to unknown smoothness via wavelet shrinkage. J. Amer. Statist. Assoc. 90 pp. 1200–1224, 1995.

[10] D. Donoho, I. Johnstone, Ideal Spatial Adaptation by Wavelet Shrinkage, Biometrika, 81, pp 425-55, 1994.

[11] D. Donoho, R. Coifman, Translation-invariant de-noising, in: A. Antoniadis, G. Oppenheim (Eds.), Wavelets and Statistics, Springer-Verlag, New York, 1995.

[12] B. Efron, T. Hastie, I. Johnstone, R. Tibshirani, Least Angle Regression, Annals of Statistics 32 (2) pp. 407–499, 2004

[13] M. Gavish, R. Coifman, Sampling, denoising and compression of matrices by coherent matrix organization, Applied and Computational Harmonic Analysis, 33 (3), pp. 354-369, 2012.

[14] M. Gavish, B. Nadler, R. Coifman, Multiscale wavelets on trees graphs and high dimensional data, in: Proceedings of ICML, 2010.

[15] J. Hunter, Matplotlib: A 2D Graphics Environment, Computing in Science and Engineering 9 (90), 2007.

[16] E. Jones et al, Scipy: Open Source Scientific Tools for Python, 2001.

[17] R. Kannan, S. Vempala, A. Vetta. On clusterings: Good, bad and spectral. J. ACM 51, 3, pp. 497-515, 2004.

[18] S. Lafon, Diffusion maps and geometric harmonics, Ph.D. dissertation, Yale University, 2004.

[19] W. Leeb, R. Coifman, Earth Mover's Distance and Equivalent Metrics for Spaces with Hierarchical Partition Trees, Technical Report YALEU/DCS/TR-1482, 2013.

[20] B. Nadler, S. Lafon, R. Coifman, I. Kevrekidis, Diffusion Maps - a Probabilistic Interpretation for Spectral Embedding and Clustering Algorithms, in: Principal Manifolds for Data Visualization and Dimension Reduction, Springer Berlin Heidelberg, pp.238-260, 2008.

[21] B. Nadler, S. Lafon, R. Coifman, I. Kevrekidis, Diffusion Maps, Spectral Clustering and Eigenfunctions of Fokker–Planck Operators, in Advances in Neural Information Processing Systems 18, 2005.

[22] T. Oliphant, Python for Scientific Computing, Computing in Science and Engineering 9 (90), 2007.

[23] F. Pedregosa, et al. Scikit-learn: Machine Learning in Python, JMLR 12, pp. 2825-2830, 2011.

[24] F. Pérez, B. Granger, IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering 9 (90), 2007.

[25] R. Sibson, SLINK: an optimally efficient algorithm for the single-link cluster method, The Computer Journal (British Computer Society) 16 (1): pp 30–34, 1973.

[26] J.O. Strömberg, Wavelets in higher dimensions, Documenta Mathematica Extra Volume ICM-1998(3), pp. 523–532, 1998.

[27] G. Székely, M. Rizzo, Hierarchical clustering via Joint Between-Within Distances: Extending Ward's Minimum Variance Method, Journal of Classification 22, pp. 151-183, 2005.

[28] J. Ward, Hierarchical Grouping to Optimize an Objective Function. Journal of the American Statistical Association 58 (301): pp. 236–244, 1963.