In [1]:
```
from pylab import *
def ls(A):
    "Normalize a matrix to be a stochastic matrix."
    return A*1.0/maximum(1e-6,sum(A,0)[newaxis,:])
def vs(v):
    "Normalize a vector to sum to 1."
    return v*1.0/maximum(1e-6,sum(v))
def unary(i,n):
    "Generate a unary vector."
    assert i<n
    v = zeros(n)
    v[i] = 1
    return v
```

Hidden Markov Models

- speech recognition
- online handwriting recognition
- OCR
- DNA sequence analysis

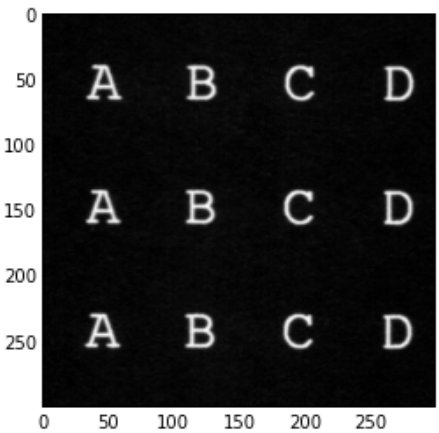
Training Data

Let's build an HMM model of a character. To do this, we need to construct training data that we can use in the rest of this worksheet. For that, we use a page of 30 rows of 26 upper case letters, arranged in a widely spaced grid.

In [19]:
```
letter = imread("letter.png")
letter = mean(letter,2)
letter = letter*1.0/amax(letter)
letter = 1.0-letter
```

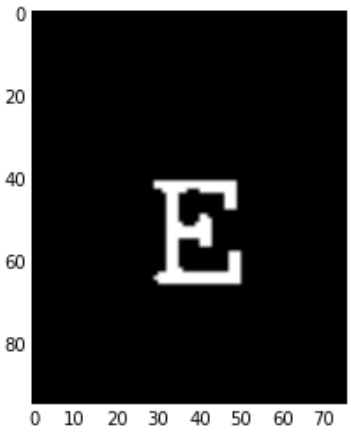
In [20]:
```
gray()
imshow(letter[:300,:300])
```

Out[20]: <matplotlib.image.AxesImage at 0x9a0fbd0>



In [21]:
```
h,w = letter.shape
ch,cw = h//30,w//26
lines = [letter[i*ch:(i+1)*ch,:] for i in range(30)]
chars = [[l[:,j*cw:(j+1)*cw] for j in range(26)] for l in lines]
chars = array(chars)
chars = array(chars>0.5*amax(chars),'B')
Es = chars[:,4,:,:]
gray()
imshow(Es[5])
```

Out[21]: <matplotlib.image.AxesImage at 0x9cbb590>



The individual letters aren't centered, so let's do that next.

In [22]:
```
from scipy.ndimage import measurements,interpolation

def centered(image,size=None):
    center = array(measurements.center_of_mass(image))
    tcenter = array(image.shape)/2
    delta = tcenter-center
    shifted = interpolation.shift(image,delta)
    if size is None: return shifted
    h,w = shifted.shape
    th,tw = size
    ch,cw = h//2-th//2,w//2-tw//2
    return shifted[ch:ch+th,cw:cw+tw]
```

We now have a stack of letters "E" in sequence. We put them together into a long row of letters.

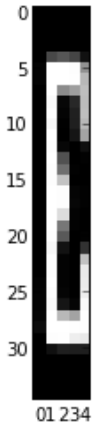
In [23]:
```
Es = [centered(E,size=(35,30)) for E in Es]
signal = hstack(Es).T
figsize(12,8)
imshow(signal.T)
```

Out[23]: <matplotlib.image.AxesImage at 0x9f25490>

Next, we are going to transform vertical slices through this input into tokens, namely by clustering.

In [25]:
```
from sklearn import cluster
km = cluster.KMeans(k=5)
km.fit(signal)
centers = km.cluster_centers_
figsize(4,4)
imshow(centers.T,interpolation='nearest')
```

Out[25]: <matplotlib.image.AxesImage at 0xa090750>



We can now represent the signal as a sequences of numbers, each indicating which cluster center represents that particular slice best.

In [26]:
```
outputs = km.predict(signal)
print outputs[:35]
```

```
[0 0 0 0 0 0 3 3 1 1 1 2 2 2 2 2 2 2 3 3 3 3 4 4 4 0 0 0 0 0 0 0 0 0 0]
```

We have a fairly reasonable number of examples for each slice type.

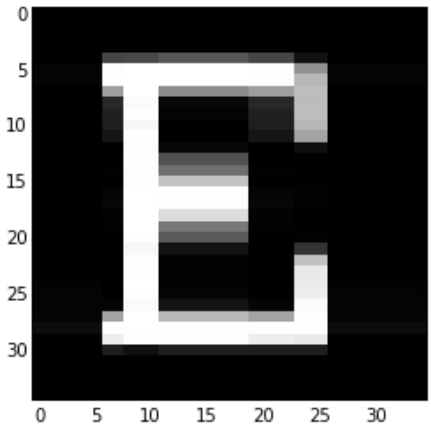
In [27]:
```
from collections import Counter
Counter(outputs)
```

Out[27]: Counter({0: 278, 2: 230, 3: 194, 4: 102, 1: 96})

We can also reconstruct the original signal in terms of these slices.

In [29]:
```
imshow(centers[outputs[:35]].T,interpolation='nearest')
```

Out[29]:
```
<matplotlib.image.AxesImage at 0xa5f95d0>
```



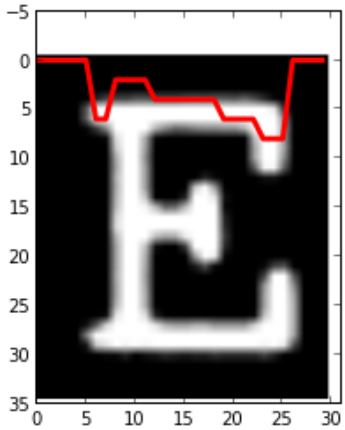
Manual Construction of a Model

We'd like some kind of model that represents characters fairly well. How can we construct it?

Let's start by constructing the "average character" and look at the sequence of slice codes it corresponds to.

In [31]:
```
Em = mean(array(Es),0)
figsize(4,4); imshow(Em)
s = km.predict(Em.T)
s
xlim(0,len(s)+1)
plot(2*s,color='r',linewidth=3)
```

Out[31]:
```
[<matplotlib.lines.Line2D at 0xa6aa950>]
```



Now we don't know how many repeats of these slices we actually need.

Well, we do approximately, but for the sake of argument, let's construct a generic model that just represents the sequences of slices.

In [33]:
```
# find where the VQ code changes
s0 = s[s!=roll(s,-1)]
print s0
figsize(4,4)
# display just the repeated slice sequence
subplot(121); imshow(tile(centers[s0],(10,1)).T,interpolation='nearest')
from itertools import chain
# repeat each slice four times
sr = array(list(chain(*[[x]*4 for x in s0])),'i')
subplot(122); imshow(tile(centers[sr],(5,1)).T,interpolation='nearest')
```

```
[0 3 1 2 3 4]
```

Out[33]: <matplotlib.image.AxesImage at 0xadbb450>

Durational Models

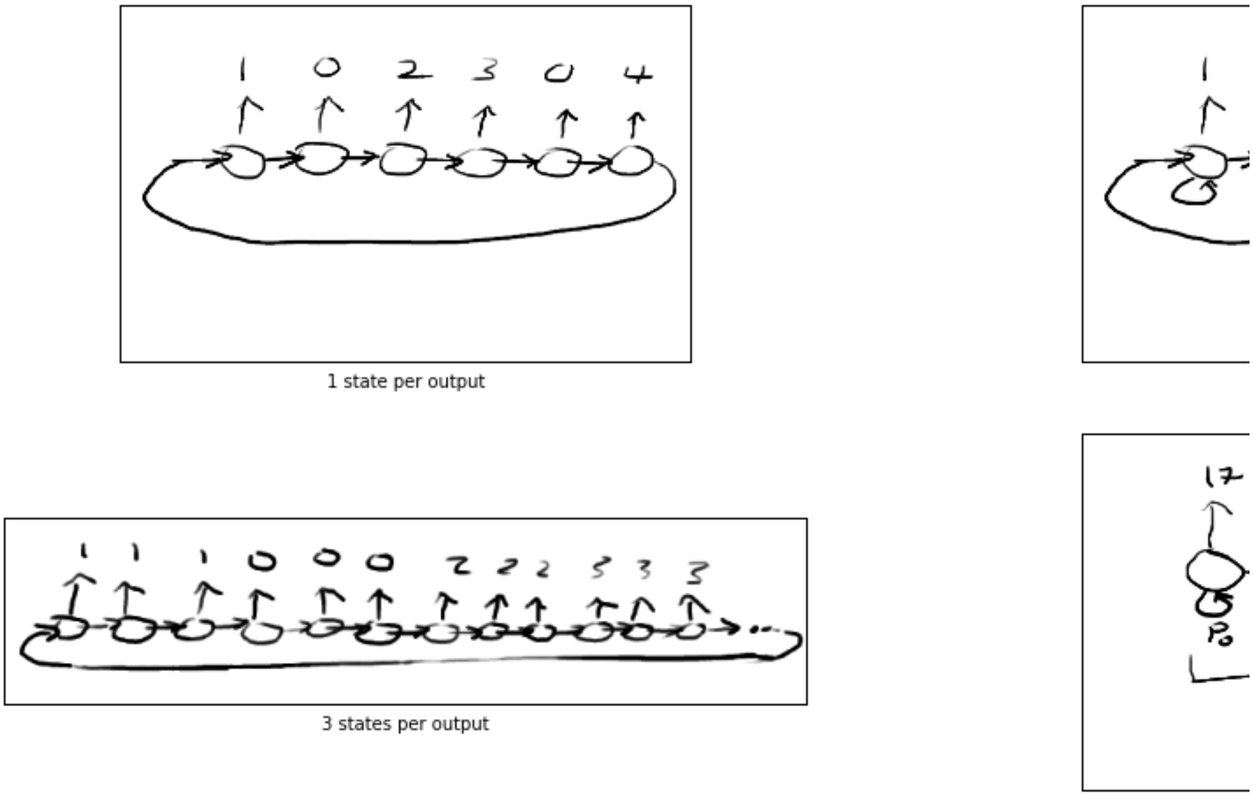
In the figure above, for the first case, we have 1 state per output. This yields a very compressed image. We need a *durational model* (thinking of the *[Math Processing Error]* axis as time).

The second case uses exactly three states for each output, giving rise to perfectly uniformly spaced outputs.

For our first attempt, we are going to use self transitions; these allow variable durations in each state. However, the probability distribution for how long we stay in a state has an exponential distribution.

In [34]:
```
figsize(18,8)
subplot(221); xticks([]); yticks([]); imshow(imread("Figures/states-1.png")); xlabel
subplot(222); xticks([]); yticks([]); imshow(imread("Figures/states-2.png")); xlabel
subplot(223); xticks([]); yticks([]); imshow(imread("Figures/states-3.png")); xlabel
subplot(224); xticks([]); yticks([]); imshow(imread("Figures/states-4.png")); xlabel
```

Out[34]: <matplotlib.text.Text at 0xb634c10>



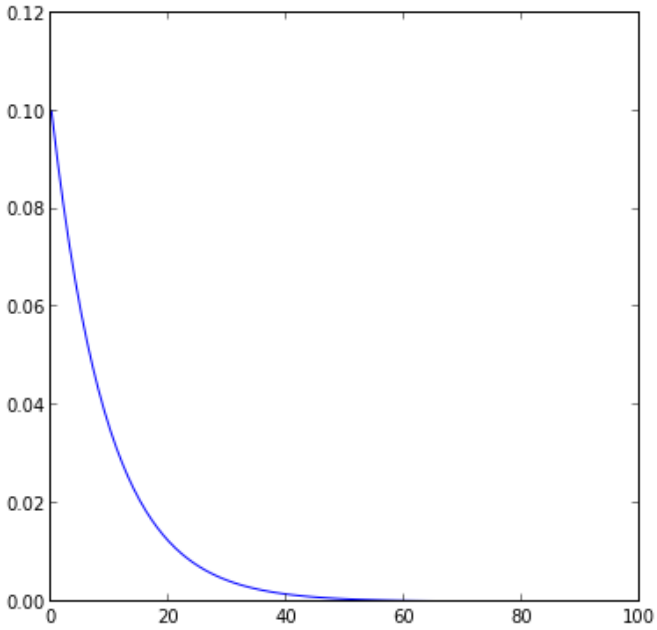
However, by using combinations of states that output the same symbol, we can approximate arbitrary distributions.

Particularly simple to approximate is a normal distribution for a durational model, because of the central limit theorem about sums of random variables.

The duration in the individual state is exponentially distributed.

In [35]:
```
figsize(6,6)
p0 = 0.9
decay = vs(array([p0**n for n in range(100)]))
plot(decay)
```

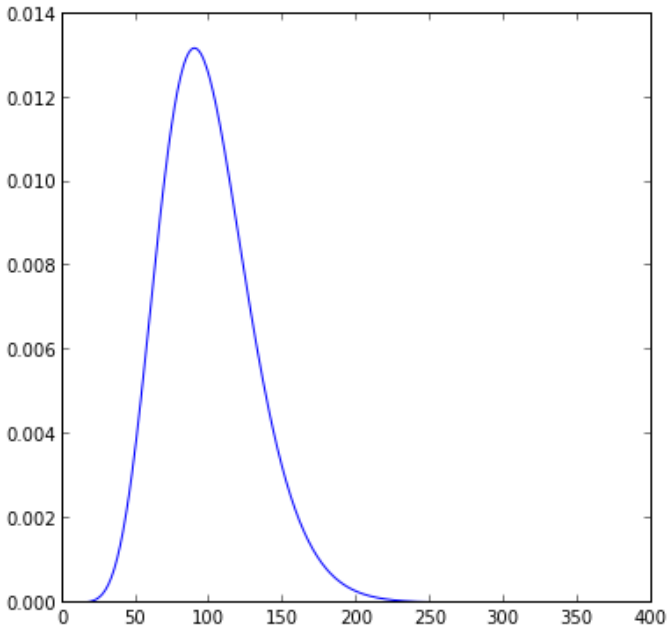
Out[35]: [<matplotlib.lines.Line2D at 0xc1299d0>]



But if we add up the durations of multiple states, we get a peaked distribution that approximates a normal distribution.

In [36]:
```
figsize(6,6)
result = vs(decay)
for i in range(10):
    result = vs(convolve(result,decay))
plot(result[:400])
```

Out[36]: [<matplotlib.lines.Line2D at 0xc51cb10>]

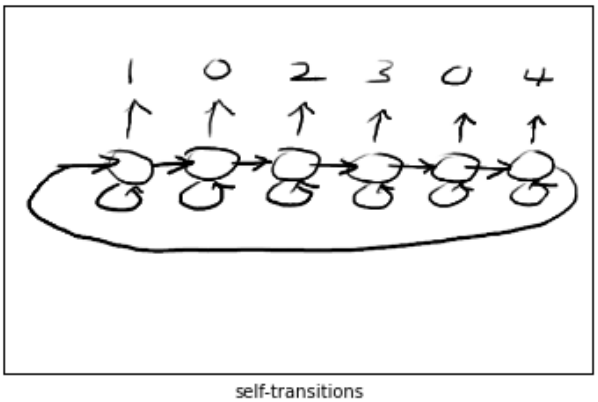


Transition Matrix for Manual Model

So for now, let's just build a model similar to the model below.

In [37]:
```
figsize(6,6)
xticks([]); yticks([]); imshow(imread("Figures/states-2.png")); xlabel("self-transit
```

Out[37]: <matplotlib.text.Text at 0xc137550>



self-transitions

This is not going to match the "durations" of our character models very well, but we can still have the learning algorithms optimize it.

In [38]:
```
ns = len(s0) # number of states
no = len(centers) # number of centers
```

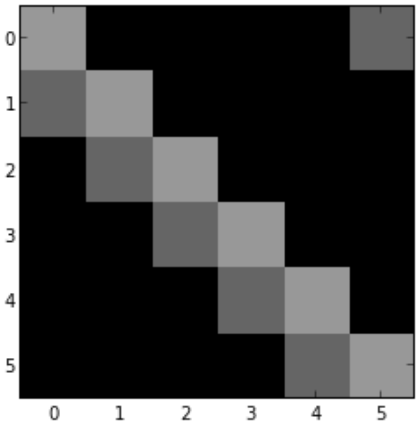
Here, we implement the state transition matrix. Note that we set a "probability floor", since we are going to use the matrix for estimation later, and zeros (=something that never happens) don't work so well.

This transition matrix can be thought of as a linear function that maps old states into new states.

Note that a lot of literature on Markov models uses the opposite convention for subscripts. Our transition matrices are multiplied from the left with a column vector of old states and yield a new state.

In [49]:
```
A = zeros((ns,ns))
for i in range(ns):
    A[i,i] = 0.6
    A[(i+1)%ns,i] = 0.4
A = ls(maximum(1e-3,A))
figsize(4,4)
imshow(A,interpolation='nearest',vmin=0,vmax=1)
```

Out[49]:
```
<matplotlib.image.AxesImage at 0xe26eb10>
```



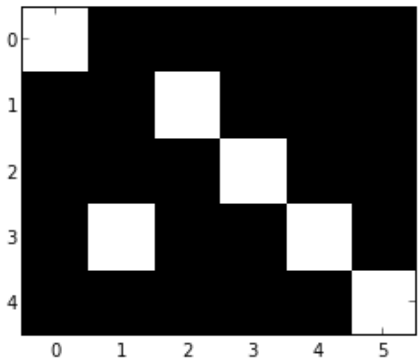
We still need to write down how the individual states correspond to output symbols. In fact, in our case, that's quite simple, since each state corresponds to a single output symbol (but not the other way around).

The B matrix is what makes Hidden Markov Models different from Markov chains. In Markov chains, we can observe what state the chain is in. In a Hidden Markov Model, we have imperfect information about the state of the system. B is a kind of "confusion matrix" or "error matrix" for observing the state of the system. It's really the simplest such system; there may be much more general kinds of observations and errors we can make.

In HMMs for speech recognition, the "observations" are actually usually the "slices" (short time spectrum) at a given time, and instead of approximating that as an output symbol via clustering (see above), the relationship between the observation vector *[Math Processing Error]* and the state is modeled directly.

In [50]:
```
B = zeros((no,ns))
for i in range(ns): B[s0[i],i] += 1.0
B = ls(maximum(1e-2,B))
figsize(4,4)
imshow(B,interpolation='nearest')
print s0
```

```
[0 3 1 2 3 4]
```



Sampling from the Distribution

First, let's define a simple function to sample from discrete distributions in general.

In [51]:
```
def rsample(dist):
    v = add.accumulate(dist)
    assert abs(v[-1]-1)<1e-3
    val = rand()
    return searchsorted(v,val)
```

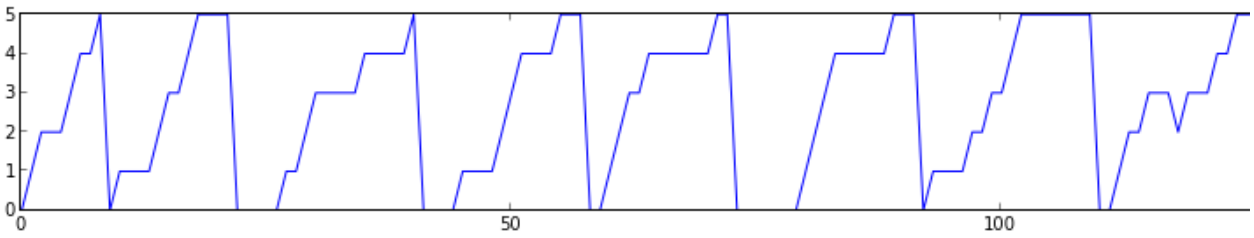
Sampling from Hidden Markov Models happens in two stages.

First, we sample the state sequence.

Then, given the state sequence, we sample the observations.

In [52]:
```
def chain_sample(A,n,state=0):
    states = zeros(n,'i')
    for i in range(n):
        p = A[:,state]
        state = rsample(A[:,state])
        states[i] = state
    return states
figsize(20,2)
plot(chain_sample(A,200))
```

Out[52]:
```
[<matplotlib.lines.Line2D at 0xe5bf650>]
```



Here is code for sampling both the state sequence and the sequence of observations.

In [53]:
```
def hmm_sample2(A,B,n,state=0):
    states = chain_sample(A,n,state=state)
    outputs = array([rsample(B[:,s]) for s in states])
    return array(outputs,'i'),states
def hmm_sample(A,B,n,state=0):
    return hmm_sample2(A,B,n,state=state)[0]
```

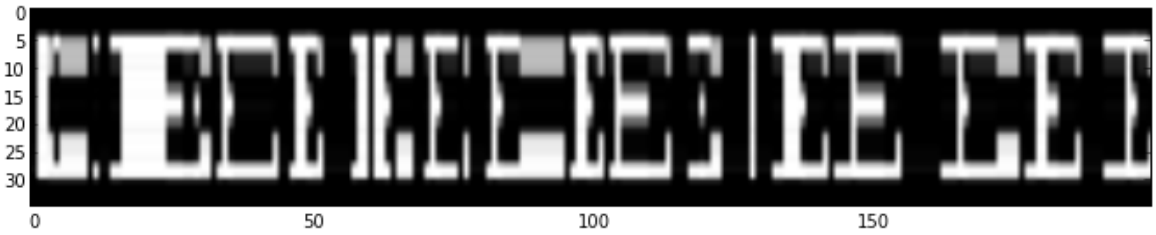
In [54]:
```
hmm_sample(A,B,20,state=s[0])
```

Out[54]:
```
array([0, 3, 1, 1, 1, 2, 3, 3, 3, 4, 4, 0, 3, 2, 1, 2, 2, 3, 4, 4], dtype=int32)
```

We can now generate new, random images from this.

In [55]:
```
imshow(centers[hmm_sample(A,B,200,state=s[0])].T)
```

Out[55]:
```
<matplotlib.image.AxesImage at 0xe795a90>
```

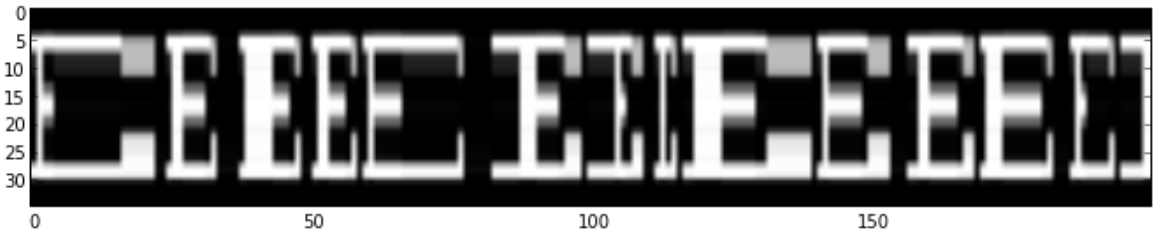


It's a little hard to tell whether this sometimes looks wrong because of the observations or because of the state sequence, so here is another sampling function that outputs the "best" observation for each state.

In [56]:
```
def best_sample(A,B,n,state=0):
    states = chain_sample(A,n,state=state)
    outputs = array([argmax(B[:,s]) for s in states])
    return array(outputs,'i')
```

In [57]:
```
imshow(centers[best_sample(A,B,200,state=s[0])].T)
```

Out[57]: <matplotlib.image.AxesImage at 0xe7cc6d0>



Since that looks kind of noisy, you might be wondering whether the model is doing anything at all. Here is the output from a completely random HMM, but using the same output symbols.

In [58]:
```
imshow(centers[hmm_sample(ls(rand(5,5)),ls(rand(no,5)),200,state=s[0])].T)
```

Out[58]: <matplotlib.image.AxesImage at 0xead5b50>

Forward Algorithm

To get started, let's compute what's known as the *forward algorithm*.

This algorithm computes *[Math Processing Error]*, that is, it updates our current estimate of what state the system is in based on the current and all previous observations.

To do this, it starts of with an initial state distribution (e.g., uniform), and then computes the posterior of the distribution given the observation:

[Math Processing Error]

Here:

- *[Math Processing Error]* is our prior
- *[Math Processing Error]* is the first observation
- *[Math Processing Error]* is given by the observation matrix *[Math Processing Error]*

As usual, we don't bother computing *[Math Processing Error]* explicitly and instead just normalize.

Now, to do the same thing for *[Math Processing Error]*, we reduce this to the above problem, by observing that all the information about *[Math Processing Error]* is already contained in *[Math Processing Error]*.

However, *[Math Processing Error]* is the state distribution prior to the state transition, so in order to get a "*[Math Processing Error]* prior", we need to multiply *[Math Processing Error]*.

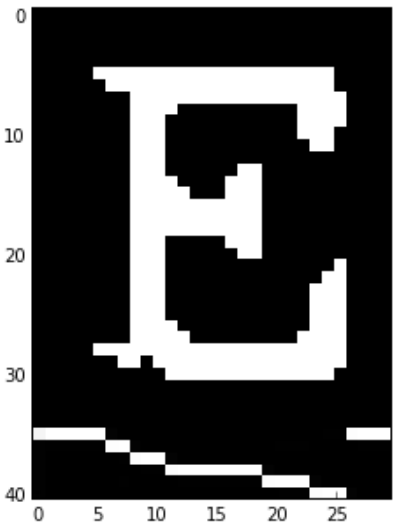
This then gives us the complete forward algorithm:

- obtain a new prior by computing *[Math Processing Error]* using the *[Math Processing Error]* matrix
- update the new state using Bayes rule and the observation *[Math Processing Error]*
- repeat

In [59]:
```
def hmm_forward(A,B,observations,p=None):
    if p is None: p = dot(A,ones(len(A))/len(A))
    fps = []
    for i,o in enumerate(observations):
        # update P(state|ovservation) using Bayes formula
        p = B[o,:]*p
        p /= sum(p)
        fps.append(p)
        # now compute the probabilities in the next state
        p = dot(A,p)
    return array(fps)
```

In [60]:
```
figsize(6,6)
fps = hmm_forward(A,B,outputs)
figsize(5,5)
imshow(r_[signal[:30].T,fps[:30].T/amax(fps)],interpolation='nearest')
```

Out[60]: <matplotlib.image.AxesImage at 0xeea3210>



Note that there are several things that this is *not*:

- the sequence of most probable states at time *[Math Processing Error]* does *not* necessarily belong to any sequence of states that can acctually even occur ("path")
- the most probable state at time *[Math Processing Error]* is *not* necessarily the "best" state given the observations

C.f.

- "The old man the boat."
- "The horse raced past the barn fell."

Viterbi Decoding

Let's now look at the problem of actually finding the best path.

The algorithm for this is quite similar to what we have already seen for string edit distance and dynamic time warping.

In [61]:
```
print ns,no
print len(outputs),amax(outputs)+1
```

```
6 5
900 5
```

What we want to find is the sequence of states *[Math Processing Error]* such that the likelihood of that state sequence given the observation is maximized:

[Math Processing Error]

The key insight is that once we know the best path to each of the states at time *[Math Processing Error]*, the subsequent search doesn't depend on it.

So, we maintain the accumulated probabilities in an array *[Math Processing Error]* (called `probs` in the code).

In addition, we maintain information about which state at times *[Math Processing Error]* actually gave rise to the best way of coming to each of the states at time *[Math Processing Error]* (called `pred` in the code). By tracing backwards, we can reconstruct the entire best path.

In [62]:
```
probs = zeros((len(outputs),ns))
pred = zeros((len(outputs),ns))
```

In [63]:
```
probs[0] = vs(ones(ns))*B[outputs[0]]
probs[0]
```

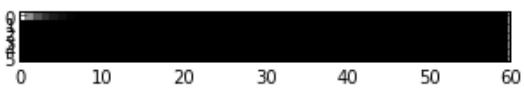
Out[63]:
```
array([ 0.16025641,  0.00160256,  0.00160256,  0.00160256,  0.00160256,
        0.00160256])
```

In [64]:
```
for t in range(1,len(outputs)):
    for j in range(ns):
        for k in range(ns):
            c = probs[t-1,k]*A[j,k]*B[outputs[t],j]
            if c>probs[t,k]:
                probs[t,k] = c
                pred[t,k] = j
```

Let's look at the probabilities.

In [65]:
```
xlim([0,60]); figsize(12,12)
imshow(probs[:60].T,interpolation='nearest')
```

Out[65]:
```
<matplotlib.image.AxesImage at 0xee8c790>
```



That's not very good; the probabilities become very small quickly because we are multiplying together. At the end, there is no information left.

In [66]:
```
probs[-1]
```

Out[66]:
```
array([ 0.,  0.,  0.,  0.,  0.,  0.])
```

We need to rewrite the algorithm to use logarithms. With that, we obtain:

In [67]:
```
costs = 999999*ones((len(outputs),ns))
pred = zeros((len(outputs),ns))
costs[0] = -log(vs(ones(ns))*B[outputs[0]])
print costs[0]
```

```
[ 1.83098018  6.43615037  6.43615037  6.43615037  6.43615037  6.43615037]
```

In [68]:
```
for t in range(1,len(costs)):
    for j in range(ns):
        for k in range(ns):
            c = costs[t-1,j]-log(A[k,j]*B[outputs[t],j])
            if c<costs[t,k]:
                costs[t,k] = c
                pred[t,k] = j
```

Now we can actually look at the costs in a sensible way.

In [74]:
```
imshow(costs[:100].T,interpolation='nearest')
```

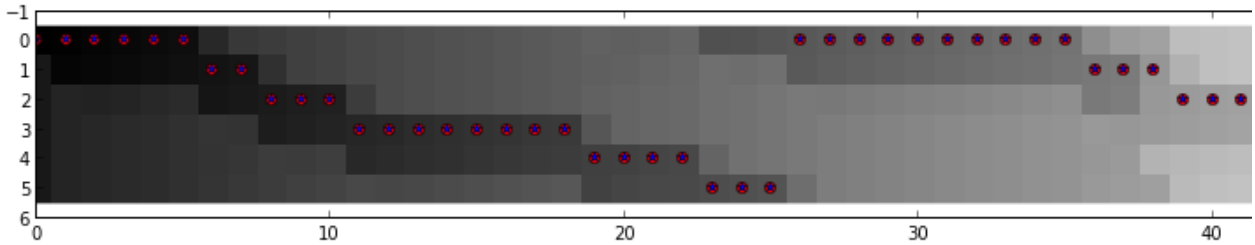
Out[74]:
```
<matplotlib.image.AxesImage at 0x10034d90>
```

We can also trace the predecessors back.

In [79]:
```
t = len(costs)-1
state = argmin(costs[t])
states = []
while t>0:
    state = pred[t,state]
    states.append(state)
    t -= 1
states.append(0)
states = array(states,'i')[::-1]
```

In [80]:
```
figsize(18,8)
xlim(0,60)
imshow(costs[:60].T,interpolation='nearest')
plot(states[:60],'ro')
plot(argmin(costs[:60],1),'b*')
```

Out[80]: [<matplotlib.lines.Line2D at 0x10332950>]



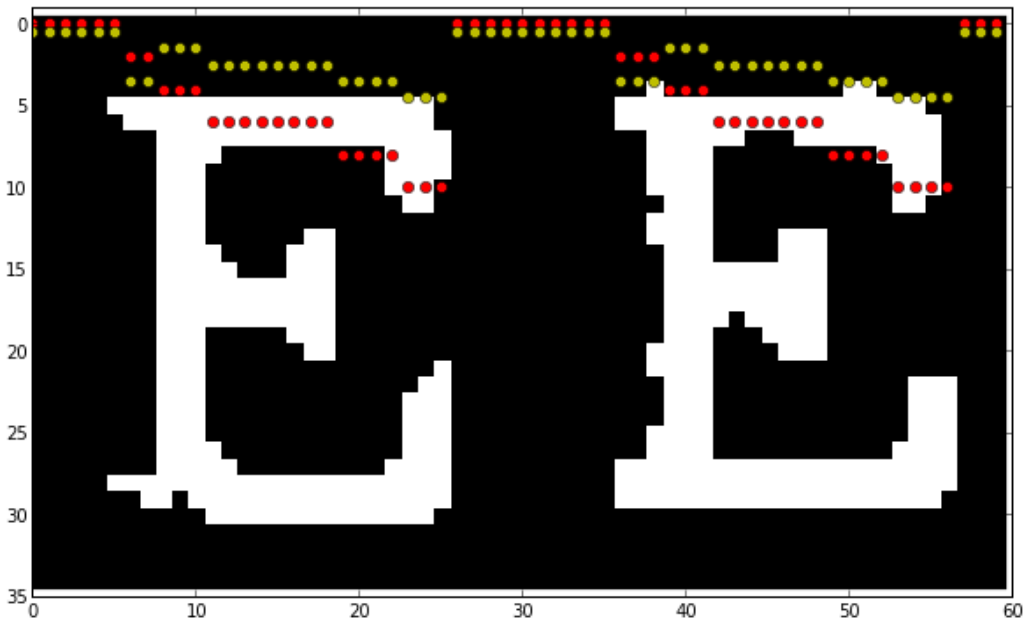
Note that the state estimates above almost agree, but not quite: the minimum cost state is not always on the best path.

Plotting the state transitions and preferred output labels on top of the data is also useful.

In [81]:
```
bout = argmax(B[:,states[:60]],0)
```

In [82]:
```
figsize(10,8); xlim((0,60)); ylim((35,-1))
imshow(signal[:60].T,interpolation='nearest')
plot(2*states[:60],'ro')
plot(bout+0.5,'yo')
```

Out[82]: [<matplotlib.lines.Line2D at 0x10791150>]



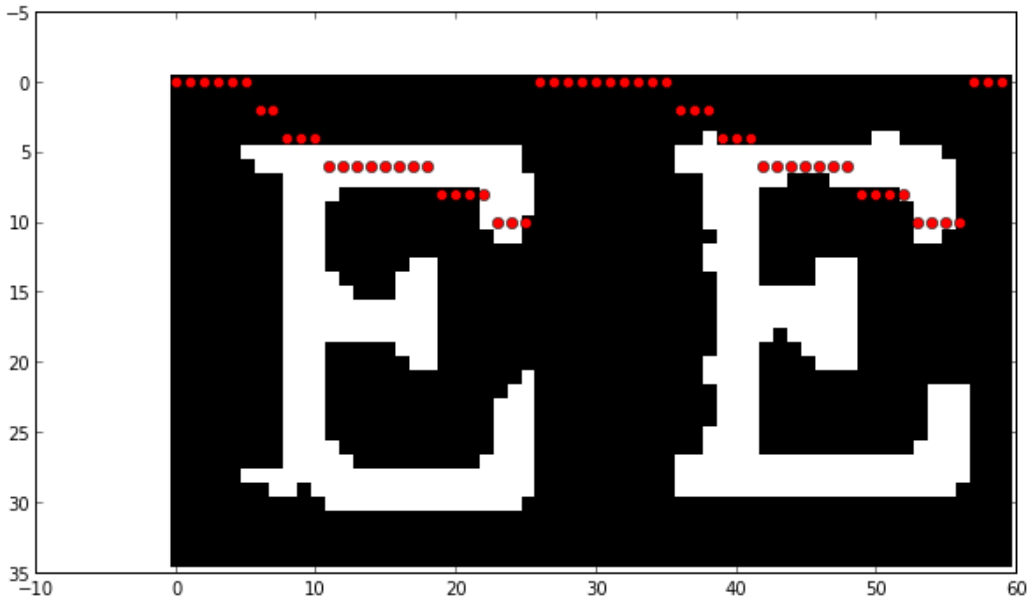
Here is everything wrapped up into a single function.

In [83]:
```
def hmm_viterbi(A,B,outputs):
    ns,ns1 = A.shape
    no,ns2 = B.shape
    assert ns==ns1
    assert ns==ns2
    costs = 999999*ones((len(outputs),ns))
    pred = zeros((len(outputs),ns))
    costs[0] = -log(vs(ones(ns))*B[outputs[0]])
    # propagate the costs forward with dynamic programming
    for t in range(1,len(costs)):
        for j in range(ns):
            ck = costs[t-1,j]-log(A[:,j]*B[outputs[t],:])
            pred[t] = where(ck<costs[t],j,pred[t])
            costs[t] = minimum(ck,costs[t])
    # trace the states backwards
    t = len(costs)-1
    state = argmin(costs[t])
    states = []
    while t>0:
        state = pred[t,state]
        states.append(state)
        t -= 1
    states = array(states,'i')[::-1]
    return states
```

Let's make sure it still works.

In [84]:
```
imshow(signal[:60].T,interpolation='nearest')
states = hmm_viterbi(A,B,outputs)
figsize(6,6)
plot(2*states[:60],'ro')
```

Out[84]: [<matplotlib.lines.Line2D at 0x10777490>]



Viterbi Training

The Viterbi algorithm gives us a simple way of "learning" the matrices A and B. This is called *Viterbi training*. While commonly used in some applications, it differs from the Baum-Welch procedure originally used for training Hidden Markov Models.

(You can think of Viterbi training as being analogous to *[Math Processing Error]*-means and Baum Welch being analogous to Gaussian mixture model; the latter uses "soft assignment".)

The idea behind Viterbi training is that, once we have aligned our observations with the data, we have an estimate of the hidden, unobservable variable, namely the sequence of states. Once we have that, we can directly update the A and B matrices simply by counting.

In [85]:
```
states = hmm_viterbi(A,B,outputs)
A1 = zeros(A.shape)
for t in range(1,len(states)):
    A1[states[t],states[t-1]] += 1
A1 = ls(A1)
B1 = zeros(B.shape)
for t in range(0,len(states)):
    B1[outputs[t],states[t]] += 1
B1 = ls(B1)
```

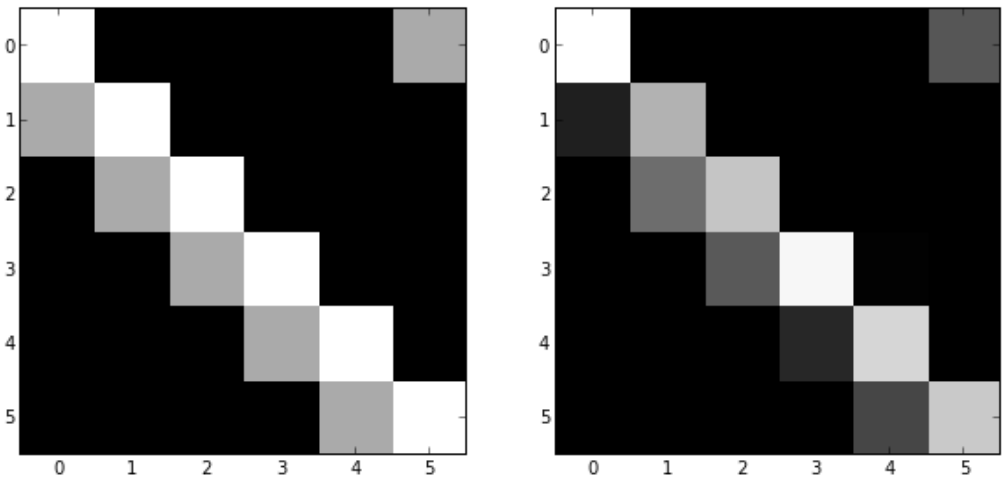
In [86]:
```
A1[:,1]
```

Out[86]:
```
array([ 0.        ,  0.62025316,  0.37974684,  0.        ,  0.        ,
        0.        ])
```

Here is a comparison of the transition matrix before and after. The structure hasn't changed much, but the probabilities have been adjusted.

In [87]:
```
figsize(10,8)
subplot(121); imshow(A,interpolation='nearest')
subplot(122); imshow(A1,interpolation='nearest')
```

Out[87]:
```
<matplotlib.image.AxesImage at 0x10e24c10>
```



As a consequence, the durational model has improved.

In [88]:
```
figsize(12,6)
subplot(211); imshow(centers[hmm_sample(A,B,200,state=s[0])].T)
subplot(212); imshow(centers[hmm_sample(A1,B1,200,state=s[0])].T)
```

Out[88]: <matplotlib.image.AxesImage at 0x11322bd0>

Let's wrap this up as well.

In [89]:
```
def hmm_viterbi_training(A,B,outputs):
    states = hmm_viterbi(A,B,outputs)
    A1 = ones(A.shape)
    for t in range(1,len(states)):
        A1[states[t],states[t-1]] += 1
        A1 = ls(A1)
    B1 = ones(B.shape)
    for t in range(0,len(states)):
        B1[outputs[t],states[t]] += 1
    B1 = ls(B1)
    return A1,B1
```

Forward-Backward Algorithm

Above, we already saw how we can compute the *forward probabilities*

[Math Processing Error]

What we might want to ask instead, however, is the probability that the system is in state *[Math Processing Error]* given all observations:

[Math Processing Error]

This is what the *forward backward algorithm* does for us.

We observe that

[Math Processing Error]

The second factor is the forward probabilities that we have already computed.

The first factor is an accumulated likelihood over a path. It is similar to the forward computation in the Viterbi algorithm, but instead of finding the best path, we add up the contributions from all paths that arrive in a particular state (we couldn't do that in the Viterbi algorithm because then we wouldn't have had a way of tracing back the best path). To propagate the probabilites backwards, we use the transpose of the state matrix.

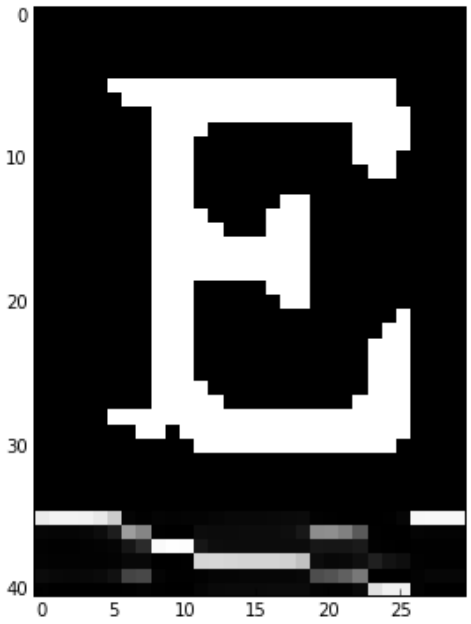
In the algorithm below, we renormalize the first factor after each step to avoid underflow.

In [90]:
```
def hmm_forward_backward(A,B,observations):
    p = dot(A,ones(len(A))/len(A))
    fps = []
    bps = []
    # the forward step
    for i,o in enumerate(observations):
        # update P(state|ovservation) using Bayes formula
        p = B[o,:]*p
        p /= sum(p)
        fps.append(p)
        # now compute the probabilities in the next state
        p = dot(A,p)
    fps = array(fps)
    # the backward step
    p = ones(len(A))
    bps = []
    for i,o in enumerate(observations[::-1]):
        # update P(state|ovservation) using Bayes formula
        p = B[o,:]*p
        p /= sum(p)
        bps.append(p)
        # now compute the probabilities in the previous
        p = dot(A.T,p)
    bps = array(bps)[::-1]
    smoothed = array(fps)*array(bps)
    smoothed /= sum(smoothed,1)[:,newaxis]
    return smoothed,fps,bps
```

Given our nice state sequence above, this would look completely boring, so let's run this with some noisy transition matrix.

In [91]:
```
figsize(6,6)
A2,B2 = ls(A1+0.5*rand(*A1.shape)),ls(B1+0.5*rand(*B1.shape))
smoothed,fps,bps = hmm_forward_backward(A2,B2,outputs)
imshow(r_[signal[:30].T,smoothed[:30].T/amax(smoothed)],interpolation='nearest')
```

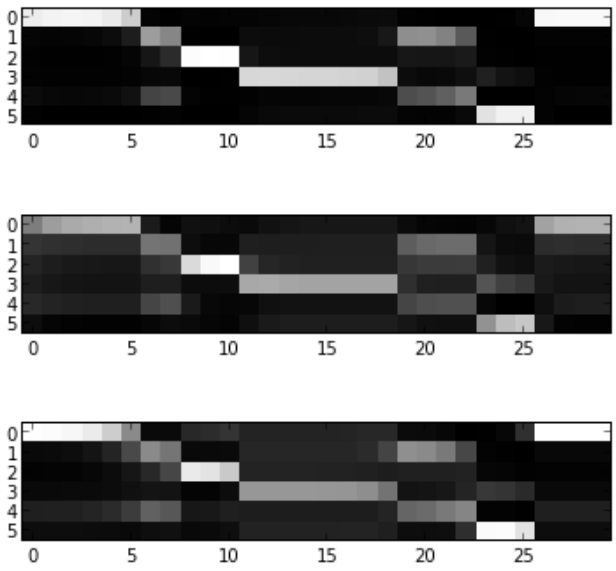
Out[91]: <matplotlib.image.AxesImage at 0x1160ca10>



Here you can see the contributions of the forward and backward directions.

In [92]:
```
subplot(311); imshow(smoothed[:30].T,interpolation='nearest')
subplot(312); imshow(fps[:30].T,interpolation='nearest')
subplot(313); imshow(bps[:30].T,interpolation='nearest')
```

Out[92]: <matplotlib.image.AxesImage at 0x11985950>



Compared to the *forward algorithm*, this algorithm does take advantage of all the information in the label sequence.

However, it is still not guaranteed that the states that maximize the posterior probability at each time *[Math Processing Error]* actually are all on the best path.

Baum-Welch Reestimation

The *forward backward algorithm* now gives us everything we need for the second HMM training algorithm, *Baum Welch reestimation*.

The idea here is similar to Viterbi training, but with two important differences:

- instead of the sequence of states on the best path, we use the most probable states at each time *[Math Processing Error]*
- instead of "hard updates" (0/1 indicator whether we are in a state), we update using the state probabilities computed by the forward backward algorithm.

In [93]:
```
ns = 5
A0 = A.copy()
B0 = B.copy()
```

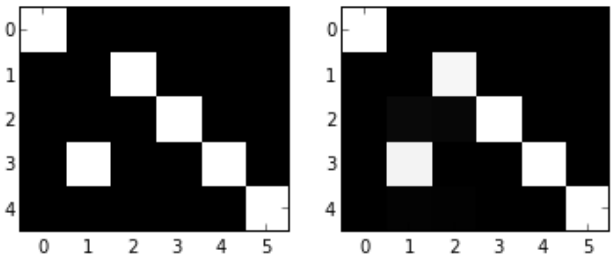
In [94]:
```
_,fps,bps = hmm_forward_backward(A0,B0,outputs)
```

The re-estimate of the matrix *[Math Processing Error]* is quite simple: we compute the probability of being in state *[Math Processing Error]* at times *[Math Processing Error]* and then update the "soft counts" in the new B matrix.

In [95]:
```
def re_B(A,B,outputs,fps,bps):
    B1 = zeros(B.shape,'f')
    for t in range(1,len(outputs)):
        state = vs(fps[t]*bps[t])
        B1[outputs[t]] += state
    return ls(maximum(1e-3,B1))
```

In [96]:
```
B1 = re_B(A0,B0,outputs,fps,bps)
subplot(121); imshow(B0,interpolation='nearest')
subplot(122); imshow(B1,interpolation='nearest')
```

Out[96]: <matplotlib.image.AxesImage at 0x11eec690>



The re-estimation for the matrix *[Math Processing Error]* is slightly trickier, since we are updating counts for a transition.

To do this, the correct procedure can be described as follows (I will not prove that here):

- we compute the forward probabilities to get into states at time *[Math Processing Error]*
- we compute the backwards probabilities to get into states at time *[Math Processing Error]* and combine that with the observation at time *[Math Processing Error]*

We might now just want to update by taking the outer product of the state vector at times *[Math Processing Error]* and *[Math Processing Error]* and add that as soft counts to our new transition matrix. If all the values were 0/1 (as in Viterbi training), that would be correct. However, for soft updates, we also need to take into account the existing weight assigned to the transition by the existing transition matrix *[Math Processing Error]* (we counted that in neither the forward nor the backward computations).

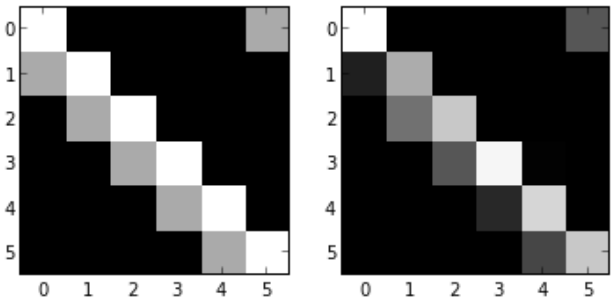
Therefore, the full reestimation for the matrix *[Math Processing Error]* looks like this:

In [97]:
```
def re_A(A,B,outputs,fps,bps):
    A1 = zeros(A.shape)
    for t in range(1,len(outputs)):
        state0 = vs(fps[t-1])
        state1 = vs(bps[t]*B[outputs[t],:])
        A1 += vs(outer(state1,state0)*A)
    return ls(maximum(1e-3,A1))
```

Running one reestimation step has an effect similar to what the Viterbi update had: some of the weights get updated, but the overall structure of the transition matrix isn't changing much.

In [98]:
```
A1 = re_A(A0,B0,outputs,fps,bps)
subplot(121); imshow(A0,interpolation='nearest')
subplot(122); imshow(A1,interpolation='nearest')
```

Out[98]:
```
<matplotlib.image.AxesImage at 0x1220d3d0>
```



Let's do some more updates and see what we get.

In [99]:
```
def reestimate(A,B,outputs):
    smoothed,fps,bps = hmm_forward_backward(A,B,outputs)
    return re_A(A,B,outputs,fps,bps),re_B(A,B,outputs,fps,bps)
```

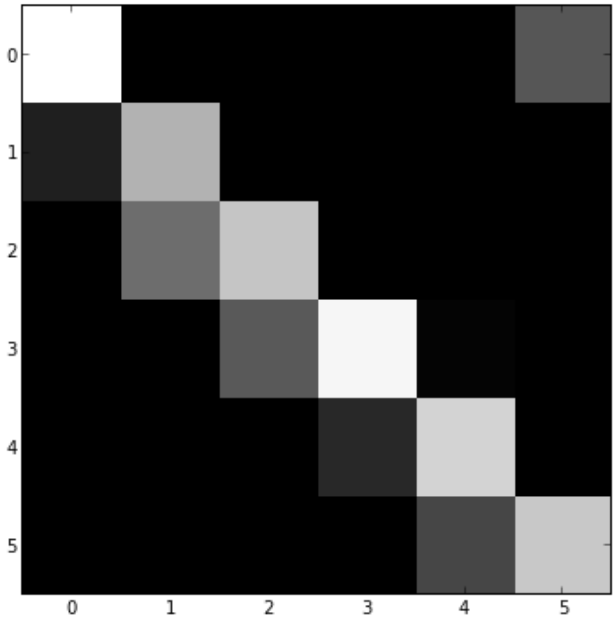
In [100]:
```
for i in range(50):
    A1,B1 = reestimate(A1,B1,outputs)
```

In [101]:
```
figsize(20,6)
subplot(211); imshow(centers[hmm_sample(A0,B0,400)].T)
subplot(212); imshow(centers[hmm_sample(A1,B1,400)].T)
```

Out[101]: <matplotlib.image.AxesImage at 0x12719110>

In [103]:
```
imshow(A1,interpolation='nearest')
```

Out[103]: <matplotlib.image.AxesImage at 0x129bcd10>



As before, the reestimated model gives much nicer outputs than the model we started with.

The odd variation in character size is a consequence of the fact that the width of each part of the letter varies randomly with an exponential distribution. Given the small number of states our model has, this is pretty close to what can be achieved.

Constrained Models and Parameter Tying

A simple way of dealing with this would seem to be to adopt a more complicated model with more states and transitions, as suggested above in the section on durational models.

The problem with that is that we have limited training data, and increasing, say, to 100 states requires then that we estimate 10000 transition probabilities; we don't have enough training data to do that.

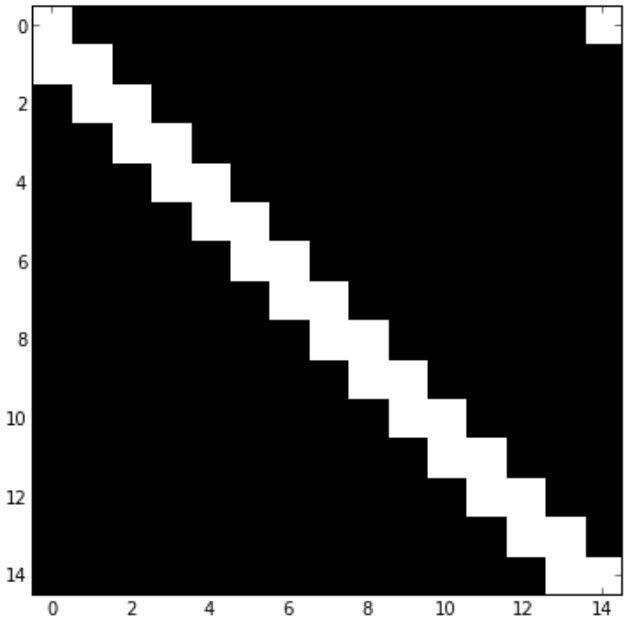
The way out of this dilemma is to constrain the structure of the transition matrices (i.e., force some of their elements to be small or zero) and/or to "tie" parameters together.

Here, I will illustrate a simple constraint on the state transition matrix, a restricted for of Bakis model.

To do this, we generate a mask of those values in the final transition matrix that are allowed to be non-zero. This matrix allows self loops and circular progression through a sequence of staes.

In [104]:
```
ns = 15
from scipy import linalg
v = zeros(ns)
v[:2] = 1.0
bakis = linalg.circulant(v)
imshow(bakis,interpolation='nearest')
```

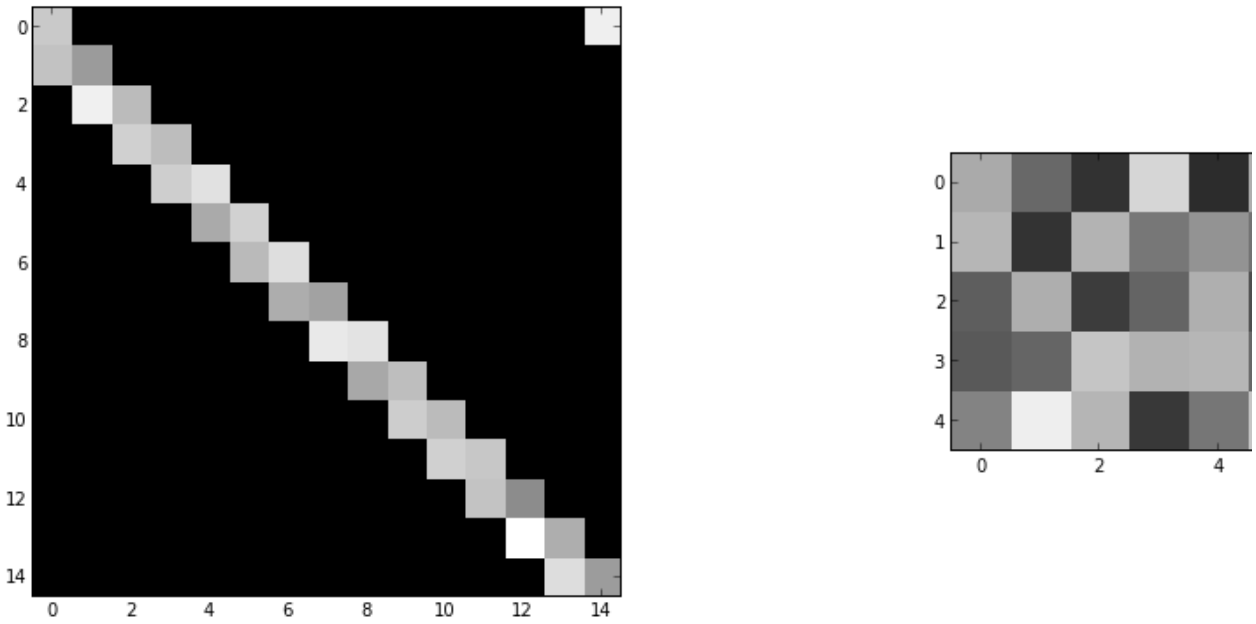
Out[104]: <matplotlib.image.AxesImage at 0x12ff9550>



Now we start with a completely random set of weights, albeit constrained by the form of the parameters above.

In [105]:
```
A0 = ls((1.0+rand(ns,ns))*bakis)
B0 = ls(1.0+rand(no,ns))
subplot(121); imshow(A0,interpolation='nearest')
subplot(122); imshow(B0,interpolation='nearest')
```

Out[105]: <matplotlib.image.AxesImage at 0x135d11d0>



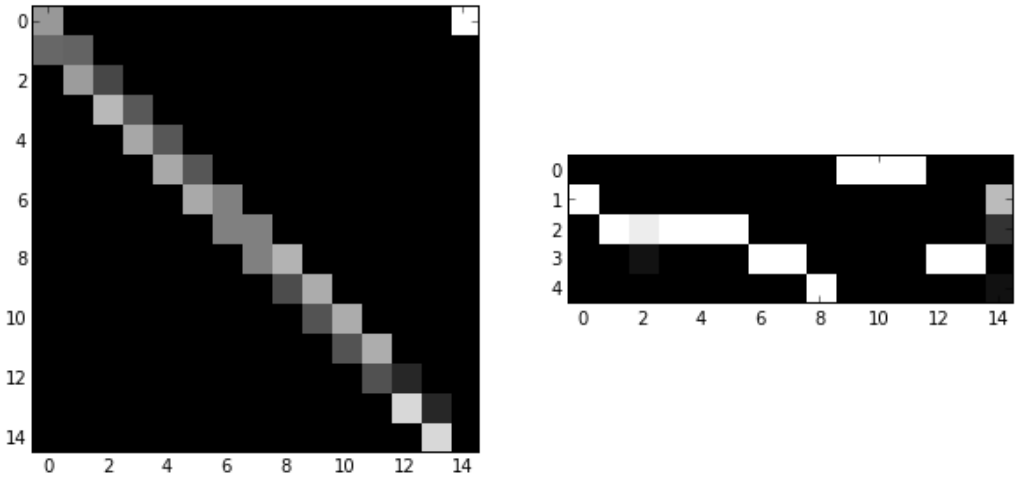
To estimate the parameters of this constrained model, we estimate the parameters of a general model and then gradually restrict the model into its desire form. That's not the best way of doing this computation (for that, you should modify the forward backward and reestimation procedures), but it will do here.

In [106]:
```
A1,B1 = A0,B0
for i in range(2,200):
    A1,B1 = reestimate(A1,B1,outputs)
    A1 = ls(A1*maximum(1.0/i,bakis))
```

These are the final transition matrices.

In [107]:
```
figsize(10,10)
subplot(121); imshow(A1,interpolation='nearest')
subplot(122); imshow(B1,interpolation='nearest')
```

Out[107]: <matplotlib.image.AxesImage at 0x13760d50>



And here are randomly generated samples from the final output.

In [110]:
```
figsize(6,6)
imshow(centers[hmm_sample(A1,B1,400)].T)
title("after forward backward training")
```

Out[110]: <matplotlib.text.Text at 0x140c43d0>

For comparison, this is what we started with.

In [111]:
```
figsize(6,6)
imshow(centers[hmm_sample(A0,B0,400)].T)
title("initial conditions")
```

Out[111]: <matplotlib.text.Text at 0x142e0250>

This looks a lot nicer than the other models, and spacing and size are much more controlled. Keep in mind that we started with random weights!

However, this model still cannot represent relationships between different parts of a character: if the first half of a character is "wide", then the second half should be as well, but that relationship is not represented here. To represent that, we would need more complex models (e.g., with multiple parallel paths).

In [ ]: