Fast Transient Source-Finding Algorithms

Joshua Burt
Cornell University
Department of Astronomy

February 6, 2014

1 Candidate Algorithms

I am considering the following algorithms for fast-transient detection.

1.1 Simple Threshold

Finds signals above a single threshold value. These signals are then grouped into contiguous regions. This is the simplest approach I have attempted. This algorithm falsely flags a lot of noise, since we are looking for low signal-to-noise signals. This method is limited by its simplicity and is unlikely to be fruitful on its own.

1.2 Flood Fill

Finds signals using two threshold values, a node threshold and a fill threshold, referred to as m_1 and m_2. Threshold m_1 finds high signal-to-noise nodes. Threshold m_2 ($< m_1$) is then used to 'enlarge' regions around each node by checking adjacent signals. Connected regions are built up around each node by including all adjacent signals exceeding m_2. Future implementations would ideally contain the ability to allow 'gaps', where a given number of adjacent signals would be permitted to fall short of the m_2 criterion without halting the growth of the region in that direction, provided a signal bordering the 'gap' *does* meet the criterion. This method has worked well at finding pulsars and should become better with added sophistication.

1.3 Friends-of-friends

Two threshold values are again utilized - a first-pass threshold and a combined S/N threshold. The first threshold, m_1, does the equivalent of the simple threshold algorithm. These flagged signals are then connected into contiguous regions, whose combined signal-to-noise ratio must exceed the second threshold, m_2. In constructing contiguous regions, there is the option to allow a single gap between signals exceeding the m_1 threshold while still being considered connected. Future implementations would ideally contain the ability to allow gaps exceeding a single channel.

1.4 (De-)Dispersion

A dispersion measure is specified, and then each frequency channel is 'rolled' forwards or backwards in time according to the dispersion law

$$\Delta t = k_{DM} * DM * \left(\frac{1}{\nu_{lo}^2} - \frac{1}{\nu_{hi}^2}\right), \tag{1}$$

$$k_{DM} = \frac{k_e e^2}{2\pi m_e c} \simeq 4.149\mathrm{GHz}^2\mathrm{pc}^{-1}\mathrm{cm}^3\mathrm{ms} \tag{2}$$

Data is summed across all frequency channels, and if the correct DM was chosen we expect an optimized peak in the time series. Looping through dispersion measures to maximize the signal-to-noise ratio (SNR) of this peak should optimize the DM. This method works well, but we are not primarily interested in finding pulsars which obey this dispersion law, as they are already well-documented in the literature. This algorithm could, however, be potentially used to find astrophysical objects obeying an inverse dispersion law, whereby they appear to have been de-dispersed rather than dispersed.

1.5 Template Convolution

A template, or mask, is convolved with the input data file. Local maxima in the 2-D convolution exceeding some signal-to-noise threshold are considered potentially significant regions. Unfortunately, the convolution operation is much slower than the other algorithms considered here. However, it is also fundamentally different from the others, as the user specifies the desired shape of the region being sought when constructing the mask. Templates could be chosen from a 'template bank' and then applied to a chunk of data. The success of this algorithm will be contingent on wisely choosing a set of templates that will find regions characteristic of astrophysical processes, as well as optimizing the convolution process to run as efficiently as possible.

1.6 Edge Detection

A derivative operator kernel is constructed and convolved with the data. This is done in both the frequency and time dimensions so that the magnitude of the derivative at each point may be calculated. Connected curves with a large derivative magnitude are searched for in the data, presumably specifying the boundary of a region. Alternatively, the frequency and time derivatives may be used to compute the direction of the derivative at each point. This direction should always point normal to the edges of a high SNR region, so curves with a slowly varying direction enclosing a region are sought. This algorithm performed very poorly in tests, likely due to the nature of the data being used (very low SNR, and a very temporally thin signal).

2 Algorithm Implementation

I have chosen to proceed with the flood-fill and friends-of-friends algorithms. In this section I will outline the specific procedures I use to implement the algorithms in Python.

2.1 Flood-Fill

The data is first smoothed or decimated as specified by the user. The mean and RMS are computed and then the data are normalized to have zero mean. The indices of all pixels exceeding the fill threshold are found (the nodes will therefore be contained within this group). A new zero-valued array with the same shape as the input data is created, and a 1 is assigned to each pixel that met the above criterion. To clarify,

at this step I have effectively created a binary array specifying the positions of signals exceeding the fill threshold. If gaps are allowed in either dimension, I use a mathematical morphology dilation operation to create a new binary array with ones surrounding each true signal according to the user specified gap allowances. Each distinct, contiguous blob of signals is then assigned (in place, in a new array of the same shape) a label, from 1 to N_{blobs}. At this point, if gaps were allowed, I multiply my array of labels by my true binary array to remove any extraneous, fake signals I introduced during the contiguous blob-finding process.

Since I only want blobs containing a node, I create another binary array using the node threshold. I then find all unique labels left after multiplying my binary array of nodes by my array of labels. This gives me a list of blobs meeting my criteria that I then loop over in my labeled array to find the positions of.

If decimation was performed, I 'un-decimate' before returning, so the output I return will always map to the original sized array.

The function returns a list of lists, each list comprised of tuples of each point within that distinct blob.

2.2 Friends-of-Friends

The process begins identical to flood-fill, with the first binary array this time generated using the first-pass threshold supplied to the algorithm.

After assigning labels, I compute the combined signal sum and number of constituent pixels for each blob. From this information and the RMS, I can get a new combined RMS value for each blob. If this value exceeds my second combined S/N threshold, I keep the blobs and output the information as in the flood-fill algorithm.

3 ROC Curves - Dispersed Pulses

I will be constructing ROC curves to determine the response of flood-fill and friends-of-friends over a variety of parametrizations. I will perform the simplest cases first, and my injected signals will be modeled after dispersed pulses. To construct ROC curves, I need to find the false-positive rate (FPR) and true-positive rate (TPR) for a given set of input parameters. A variety of approaches are possible at this stage. I will discuss the approaches I consider, along with their results.

3.1 Constructing fake data

Two parameters characterize the fake pulsars: the DM and the FWHM of the Gaussian. For each frequency channel, I generate a Gaussian array using those parameters. I do this for each channel, shifting them according to the dispersion law in equations 1 and 2. I create the equivalent of one second of one sub-band of Mock spectrometer data (512 channels x 15270 time samples). I also can randomly generate a frequency span for the pulsar if I don't wish for it to extend across the entire band, in which case the omitted channels are rows of zeros. This 2D array is then added to a normal distribution of noise with zero mean and 1σ RMS to create one second of fake data.

3.2 Event-oriented normalization

This method was only applied to friends-of-friends, and also used an erroneous approach to compute the FPR.

To determine the TPR, I normalized the number of simulated event detections by the total number of injected events. To do this, I looped over each injected event and determined what fraction of that event was contained within my algorithm output. I allow a threshold here specifying some fraction of each event that must be found in order to be considered a detection. Jim suggested I take the most inclusive approach

and set that threshold to a single pixel. That is, if a single pixel of an injected event was identified by my algorithm, then that event was considered detected and contributed to the TPR numerator.

To determine the FPR, I normalized the number of false event detections by the total number of blobs exceeding the first-pass threshold of the friends-of-friends algorithm. I specified another threshold that set the maximum number of real pixels allowed in a false detection. To be consistent with the TPR, I didn't allow any; that is, a single real pixel classified an event as a real detection, and omitted it from the FPR numerator.

3.2.1 Event-oriented implementation

Using this normalization scheme, I constructed a few simple ROC curves for the friends-of-friends algorithm. I injected 20 simulated pulses, varying the amplitude, FWHM, DM, and bandpass span. The amplitude was randomly chosen between 0.5 and 1.0σ. The 20 pulses sometimes overlapped, resulting in fewer than 20 contiguous injected events. Then, while holding the first-pass threshold constant at 1σ, I parametrized the curve by varying the combined S/N threshold. A sample curve is shown in figure 1 with two points on the curve labeled by their parametrization value, the combined S/N threshold value.

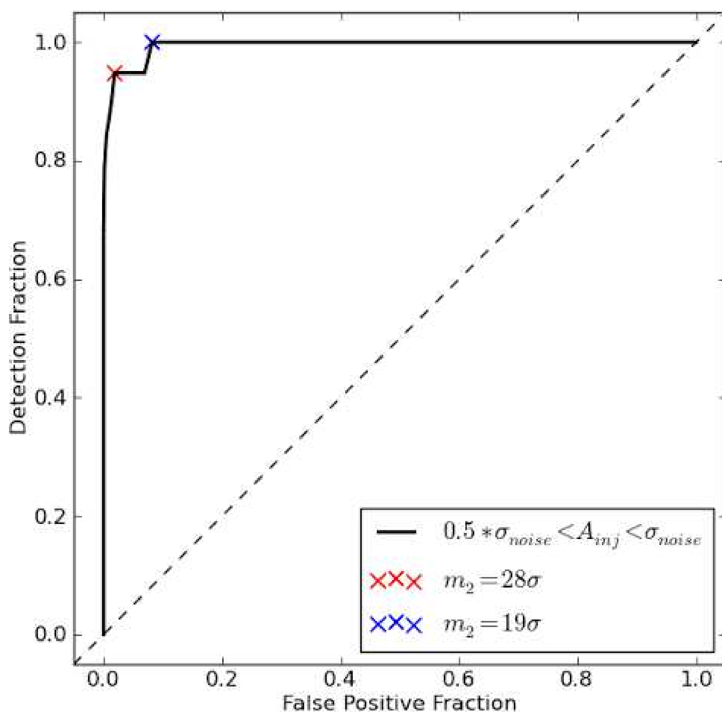

Figure 1: ROC curve using event-oriented normalization.

3.3 Pixel-oriented normalization

To determine both TPR and FPR, a much simpler and intuitive approach was taken. For the TPR, I simply found the fraction of the real signal pixels that was returned by my algorithm. The real signal pixels were defined to be all pixels exceeding half the maximum of the Gaussian-profile pulse, i.e. all points within the FWHM for each frequency channel. For FPR, I passed only the background noise array to my algorithm, and found the fraction of all pixels that was returned.

3.3.1 Pixel-oriented implementation

Using the pixel-oriented normalization scheme, I constructed a set of curves for each algorithm. For each curve, I performed many trials with a single simulated injected pulsar, with a Gaussian amplitude of 1σ. Each pulsar spanned the entire bandwidth, had a FWHM randomly generated between 1 and 5 ms, and a random DM between 50 and 300. To minimize the number of sampling points necessary for each curve, I tried to choose a final parametrization value which would yield 0 FPR, because from there on further increasing the parametrization variable would only reduce the TPR and I could just include the origin after the fact (without the computational overhead).

For my first attempt with friends-of-friends, my parametrization variable was the total combined S/N of the blob. Each curve had a first-pass threshold of 1σ. I conducted 100 trials and averaged them to get a single smooth curve. The two different curves in the set corresponded to allowing 0 and 1 gaps, respectively. The one gap was identical in each dimension (frequency and time). I wanted to include more gaps, but with a first-pass threshold of 1σ anything more than a single gap makes a smooth parametrization along the FPR axis impossible. This is because the algorithm is able to find one huge contiguous blob (because of the allowed gap size) along with much smaller blobs that happened to be isolated. When parametrizing the curve, all parametrization values below the combined S/N of this blob give me one point with a high FPR, and then once the parametrization variable exceeds the S/N of the large blob, the FPR basically falls to 0. These two points aren't enough to construct a smooth curve. I ran tests to determine the maximum expected combined S/N, so I knew up to what value I had to perform my parametrization to construct the full curve. I chose 10 sampling points along the curve, attempting to space them such that the curve was as smooth as possible. Because I was only using a first-pass threshold of 1σ, the curve could not possibly extend all the way to the upper-right-most part of the graph (as one would expect to find 50% of the points at the center of the pulse to be above 1σ, and fewer as you moved out to the FWHM). In fact, my maximum (TPR, FPR) point for these trials was around (0.16, 0.41). To complete the 200 total trials (2 curves with 100 trials each), with 10 sampling points per trial, took 2.9 hours. The results are displayed in figure 2.

For my first attempt with flood-fill, my parametrization variable was the node threshold. Each curve had a fill threshold of 1σ. The process was the same as in the preceding paragraph. For the same reason, I could not include gaps greater than 1. I suppose this says something about the typical length between signals exceeding 1σ in a normal distribution, i.e. that it is somewhere between 1 and 2 pixels. This will therefore limit the ability to allow gaps when looking for very low S/N features, as I imagine even 1 gap may be too much when dropping the fill threshold or the first-pass threshold below 1σ. This computation took 3.2 hours. The result is in figure 3.

Next, I tried to instead let the fill threshold be my variable parameter while still using the node threshold as the parametrization variable. I allowed no gaps for any of the curves. I chose fill threshold values of 0.25, 0.5, 0.75, and 1σ. This computation took 6.1 hours. The result is in figure 4.

As an aside, you'll notice that the curves don't appear to be very smooth at all, despite my attempt to choose good values of the parametrization variable. I suspect this is largely due to the nature of the algorithm. For example, if a single node with a value of 4.5σ is responsible for a blob consisting of 20% of the detection fraction, then any step from below 4.5σ to above will necessarily create a jump of that magnitude. This seems to often be the case, where one very large blob contributes significantly to the test statistic.

I generated a similar set of curves for friends-of-friends. I used 0.25, 0.5, 0.75, and 1σ as my first-pass thresholds, and the combined S/N threshold as my parametrization variable. I did not permit gaps. This computation took 5.2 hours. The results are shown in figure 5.

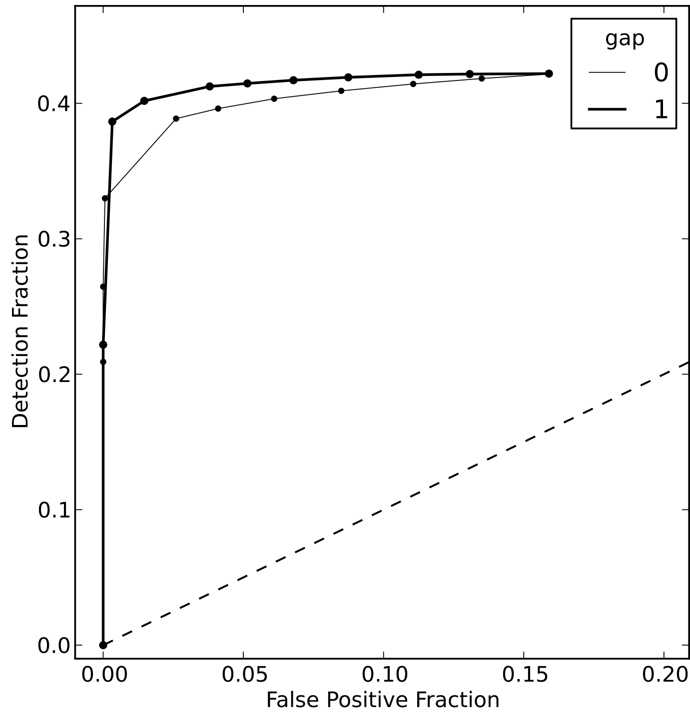

Figure 2: ROC curves using pixel-oriented normalization for the friends-of-friends algorithm. The curves are parametrized by the combined S/N threshold, use a first-pass threshold of 1σ, and have a 1σ injected signal.

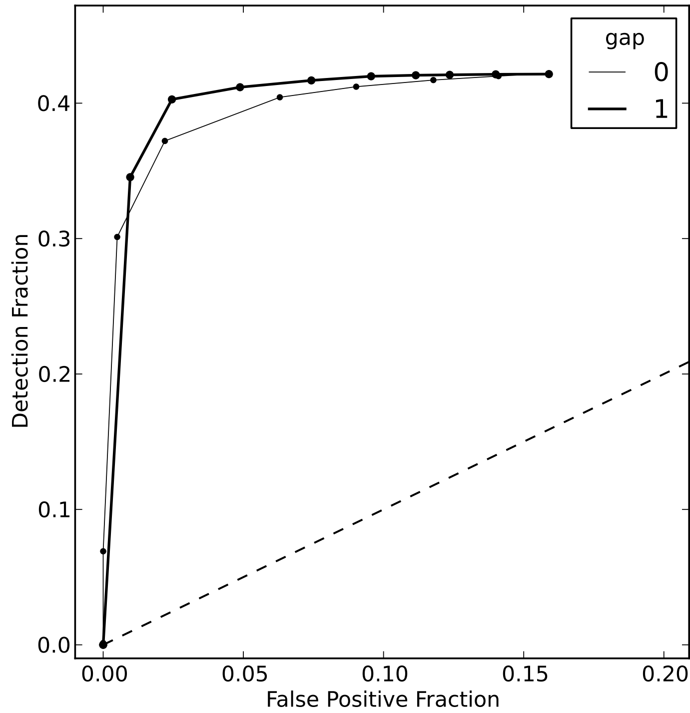

Figure 3: ROC curves using pixel-oriented normalization for the flood-fill algorithm. The curves are parametrized by the node threshold, use a first-pass threshold of 1σ, and have a 1σ injected signal.

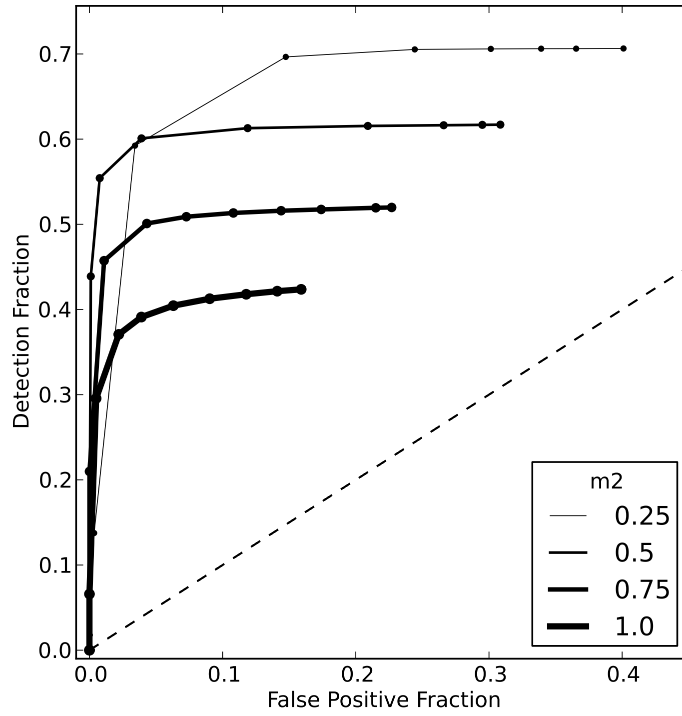

Figure 4: ROC curves using pixel-oriented normalization for the flood-fill algorithm. The curves are parametrized by the node threshold, permit no allowed gaps in either dimension, and have a 1σ injected signal.

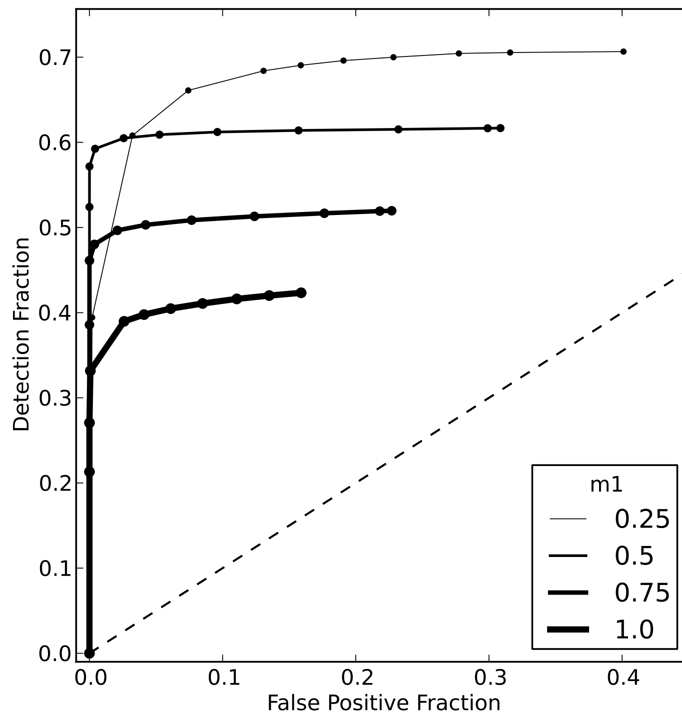

Figure 5: ROC curves using pixel-oriented normalization for the friends-of-friends algorithm. The curves are parametrized by the combined S/N threshold, permit no allowed gaps in either dimension, and have a 1σ injected signal.

To more easily compare the sets of curves, I overlaid both pairs of similar images in the section above. The results are shown in figures 6 and 7. From these plots, it is pretty clear that the friends-of-friends algorithm seems to continually outperform flood-fill, at least at low FPR values.

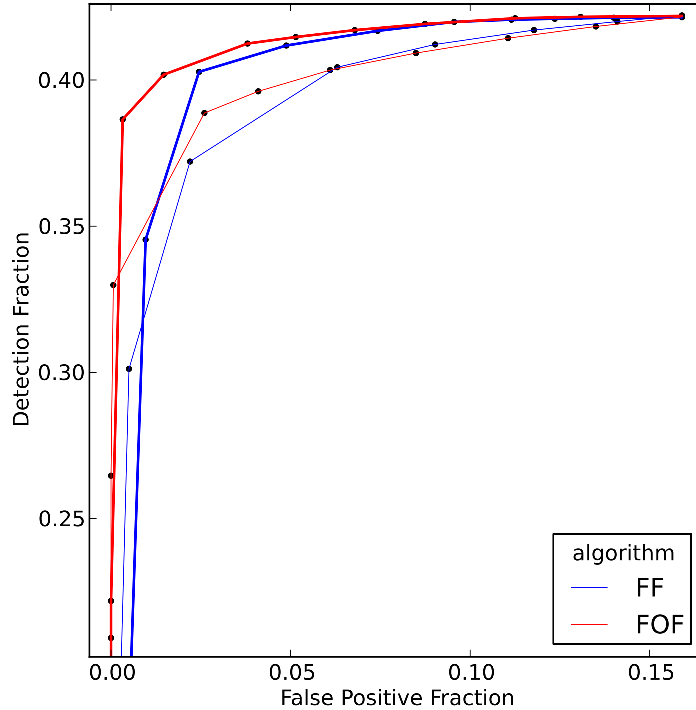

Figure 6: An overlay of the curves in figures 2 and 3.

Next, I tried to see what the curves would look like if I parametrized each algorithm by the other threshold; that is, I parametrized flood-fill by the fill threshold and used different trial values for the node threshold, and I parametrized friends-of-friends by the first-pass threshold and used different trial values for the combined S/N threshold. For each algorithm, I used 2, 3, 4, and 5σ as my m_2 trial values for consistency. I did not permit gaps in either dimension. I began the parametrizations at a threshold of -1σ for each algorithm to further extend the curves into the upper-right-hand corner. The computations took 13.0 and 11.0 hours for flood-fill and friends-of-friends, respectively. The curves can be seen in figures 8 and 9. I'm not really sure what happened with $m_1 = 5\sigma$ in figure 8, but I suspect it is a result of 5σ signals being quite rare, and thus there may not have been enough nodes to continue the trend set by the other three curves.

With these results, I will now use a much weaker injected signal, comparable to the signal strength of real astrophysical phenomena I hope to detect. The Gaussian amplitude for the injected pulsars will now be $\frac{5}{\sqrt{512}} \approx .221\sigma$. I used this value because if de-dispersion were performed at the correct DM, the time-series contribution from the injected signal at the peak of the pulse would be 5σ. For these trials I also used a pulse width of $3ms$ and a DM of $200cm^{-3}pc$.

I again constructed curves using the procedures outlined for figures 5 and 8. For flood-fill, I used a variable node threshold (between 4 and 4.4σ), and parametrized each curve using the fill threshold. For friends-of-friends, I used a variable first-pass threshold (between .221 and $.75\sigma$), and parametrized each curve using the combined S/N threshold. The results were not exactly what I expected (see figures 10 and 11). In each figure, there is at least one set of input parameters for which the curve falls below the probability of random chance. In figure 10, the curve gets somewhat better for increasing values of the node threshold until at some critical value (4.3σ) its shape alters drastically. Similarly in figure 11, the curve gets better for *decreasing* values of the first-pass threshold until at some critical value ($.22(1)\sigma$) its

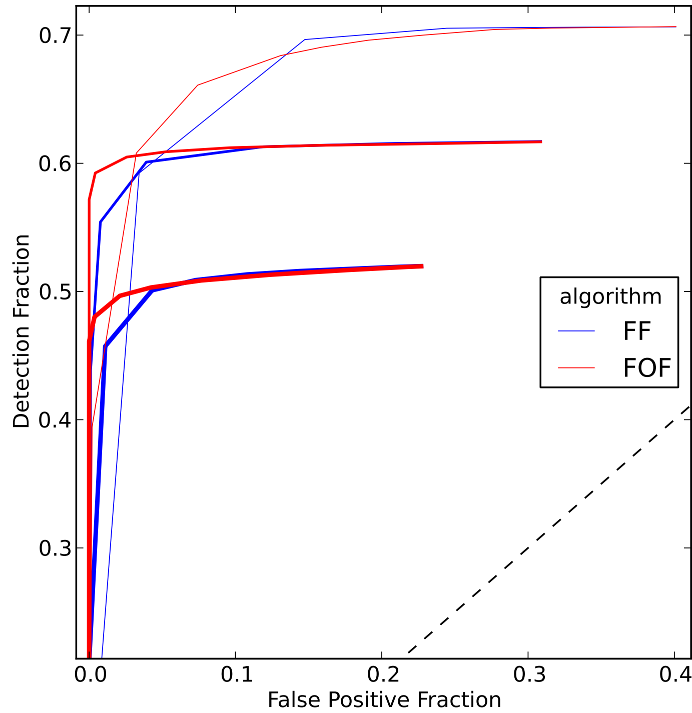

Figure 7: An overlay of the curves in figures 4 and 5. I have omitted the 1σ curves since they can be seen as the bold curves in figure 6.

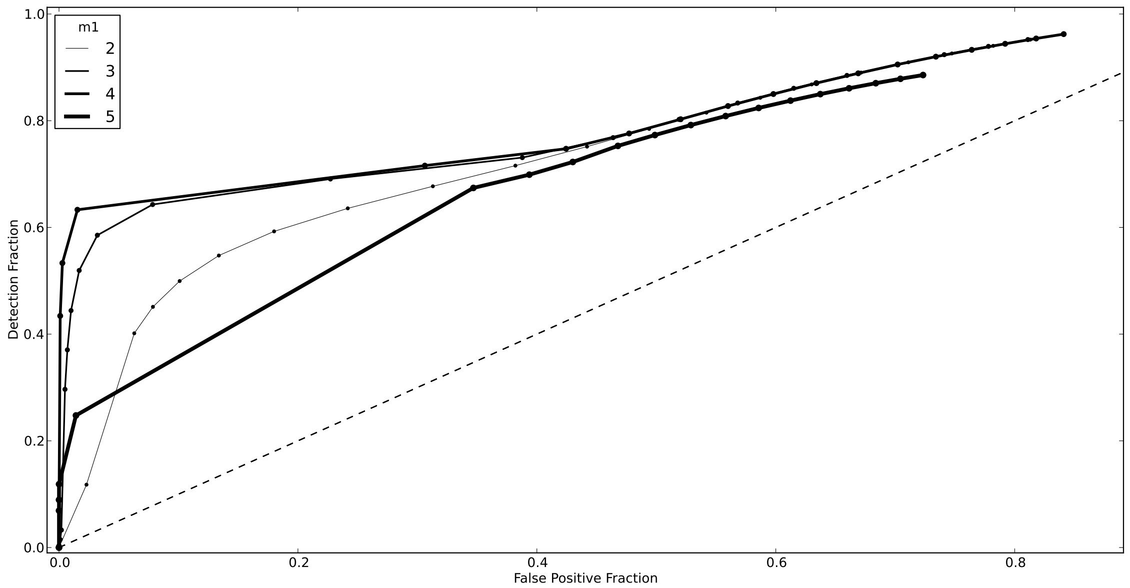

Figure 8: ROC curves using pixel-oriented normalization for the flood-fill algorithm. The curves are parametrized by the fill threshold, permit no allowed gaps in either dimension, and have a 1σ injected signal.

behavior differs. These behavior patterns are similar to those found in figures 5 and 8. I can conclude that the best set of input parameters out of the combinations I've tried, applied to a signal of this strength, are a 0.25σ fill threshold and a 200σ combined S/N threshold for the friends-of-friends algorithm. This led me to a maximum ratio of the true signals detected to false positive detections of 100. Unfortunately, this set

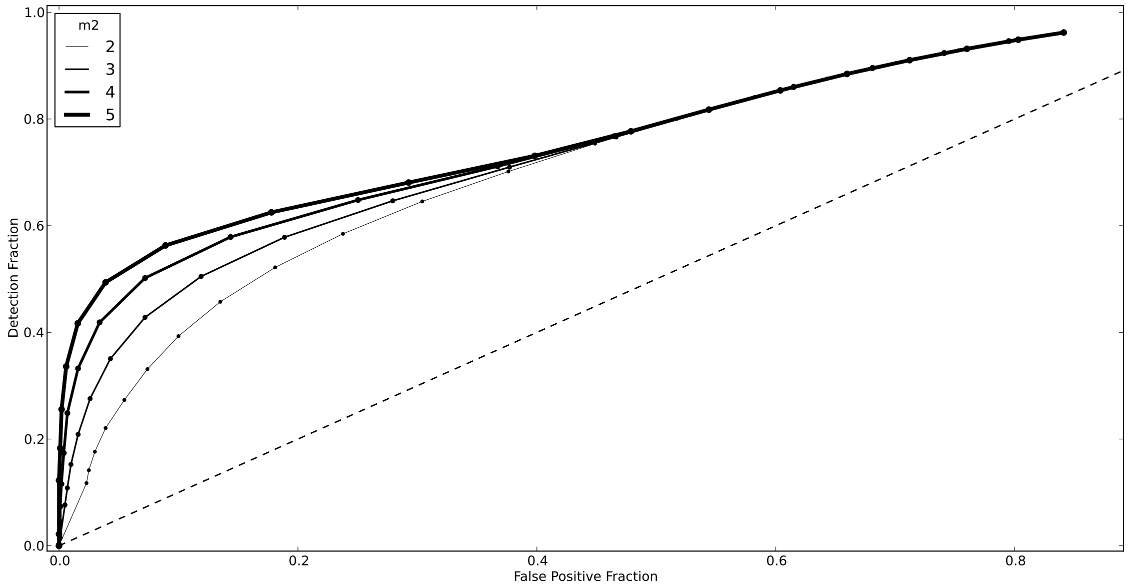

Figure 9: ROC curves using pixel-oriented normalization for the friends-of-friends algorithm. The curves are parametrized by the first-pass threshold, permit no allowed gaps in either dimension, and have a 1σ injected signal.

of parameters also fails to detect approximately 80% of the sources in these trials.

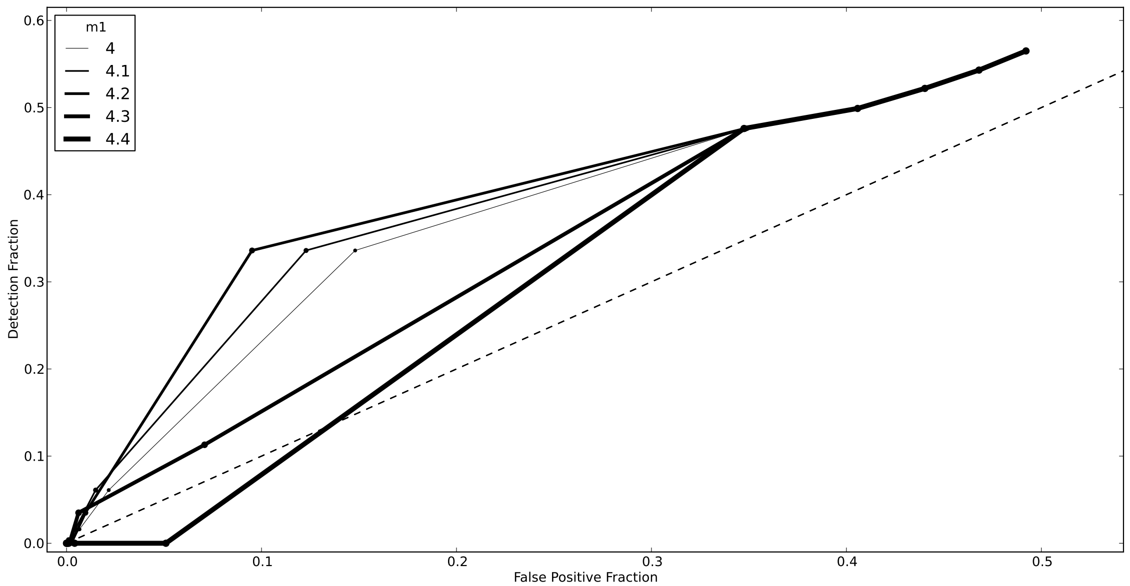

Figure 10: ROC curves using pixel-oriented normalization for the flood-fill algorithm. The curves are parametrized by the fill threshold, permit no allowed gaps in either dimension, and have a $.221\sigma$ injected signal.

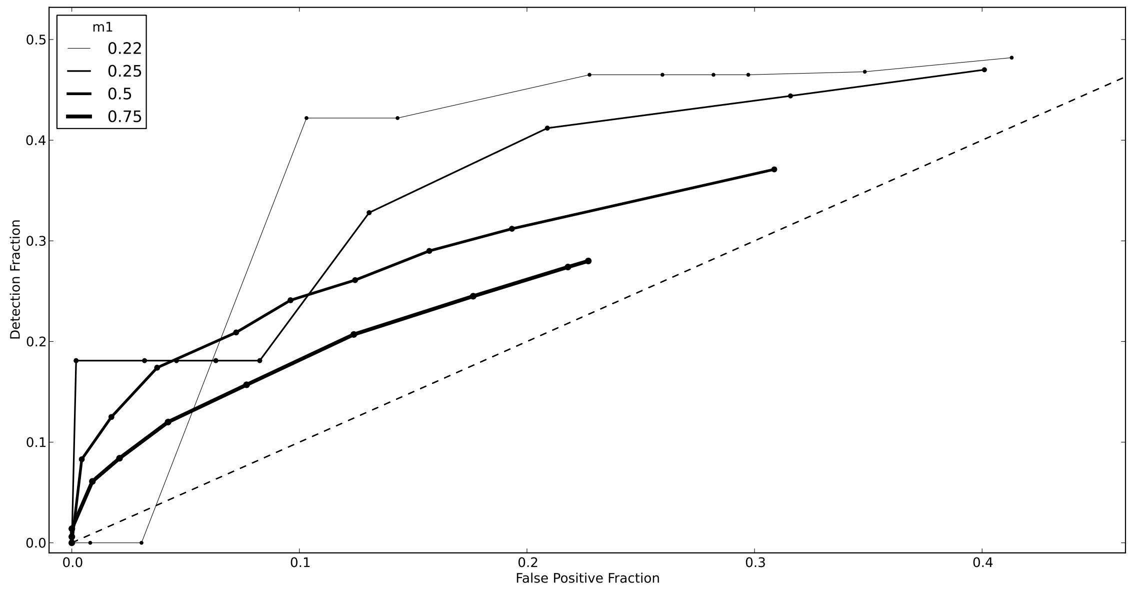

Figure 11: ROC curves using pixel-oriented normalization for the friends-of-friends algorithm. The curves are parametrized by the combined S/N threshold, permit no allowed gaps in either dimension, and have a $.221\sigma$ injected signal.