**Technical Report**

CCB-95-03/October 15, 1995

# Walsh-Fourier Analysis of Protein Sequence Property Profiles

Eleftherios Gkioulekas and Jerry Solomon

The Beckman Institute
California Institute of Technology
Pasadena, CA 91125

The work described in this report was supported in part through a grant from the National Science Foundation, Science and Technology Center for Molecular Biotechnology (BIR9214321), and by the Beckman Institute, California Institute of Technology. Part of this work was performed by E.G. as a Caltech Summer Undergraduate Research Fellowship (SURF) project.

# Walsh-Fourier Analysis of Protein Sequence Property Profiles

Eleftherios Gkioulekas and Jerry Solomon

June 29,1995

## Abstract

As a first step to predicting the 3D conformation of a protein given the amino acid sequence, we have addressed the problem of classifying protein sequences to structural families. To test whether a protein sequence belongs to a certain group, we use decision algorithms that decide between the group and a group of randomly generated sequences.Protein sequences are mapped to digital signals, called profiles, by mapping each of the 20 amino acids to a hydrophobicity value. The coefficients of the Walsh expansion of the sequence profile are used as discriminatory features for a Neyman-Pearson test. In this study we evaluate the Walsh transform as a possible feature space for sequence classification problems. Performance is illustrated on the Calcium-Binding sequence class and the Hemoglobin class. Finally we explore techniques for preprocessing property profiles before passing them on to walsh analysis. (gamma correction on property scale, smoothing of property profile before walsh transformation).

## I. Introduction

Proteins play a most vital role in biochemical processes.The 3 dimensional conformation of a protein can tell us a lot more about its function rather than a mere amino acid sequence, and this information can be very important to understanding larger biochemical pathways in which the protein in question contributes one of the many steps. Unfortunately, even though amino acid sequences are relatively easy to obtain, determining the 3D conformation requires a more sophisticated method, known as X-ray crystallography. As a result, many amino acid sequences have been cataloged, but we only know the conformations for a limitted number of them. The *protein folding problem* refers to determining how and why proteins in nature adopt a particular conformation. Given that the number of possible conformations approaches $10^{270}$ for an average sized protein, this is a non-trivial problem.

When a protein is first created in a cell, it doesn't adopt its final shape until it is completely constructed and released. Then a process called *protein folding* begins, in which the protein molecule goes through a sequence of different conformations, seeking to minimize its energy. The final result depends solely on the amino acid sequence and the enviroment. It usually takes 0.1 to 1sec for a protein to fold, however, a detailed dynamical simulation based on quantum mechanics can only follow the folding for a few picoseconds. For this reason, much work has been done on finding starting conformations from which one can hope that a dynamical simulation will lead to the final structure of the protein. If we examine the sequences for which the 3D conformations are known, it turns out that many different proteins adopt similar folds. This clustering of conformations around structural families is probably the result of the evolution of the current array of proteins from a small number of primordial folds.Once the structural family is known, we can incorporate all the implied restrictions on a Monte Carlo method that will find the most likely chains. For instance, a very common restriction is the tendency of hydrophobic aminoacids to hide at the core of the protein and the tendency of hydrophilic aminoacids to be on the surface. Such macroscopic information can eliminate many 'unphysical' conformations, but it cannot be incorporated directly into the Hamiltonian used in a dynamical simulation. However, it **can** be incorporated in a Monte-Carlo method that finds the most likely chains, which is why that extra step is necessary.

In this paper, we discuss and evaluate a method for creating algorithms that will classify protein sequences into groups of structural families. The importance of such algorithms is immense when addressing the protein folding problem, but they also are of use in many other circumstances. Hypothesis testing algorithms test whether a statement is true or not, and this is done by rejecting the *null hypothesis* $H_0$ (the predicate that our statement is incorrect). When it comes to hypothesis testing there are two types of errors that can occur. If the test incorrectly rejects the hypothesis when it is indeed correct, we say that we have a *type I error.* When our test says that the hypothesis is correct, when it is false, then we say that we have a *type II error.* One of the things that should be considered when designing a hypothesis test is how undesirable these two types of errors are. One of the advantages of our approach is that it will allow us to have a quantitive control over the propabilities for type I and type II errors.

## II. General classification theory

We begin by sketching out how one goes about training an algorithm that will tell two protein sequence classes apart. The very first thing that needs to be done is map our protein sequence to a *numeric* one. The simplest approach is to map every amino-acid to an appropriate number, following a fixed set of correspondances. The resulting numeric sequence is called a *property profile.*

The use of property profiles has a relatively long history in protein sequence analysis. Depending on one's objective any number of amino acid properties can be used in constructing such profiles, *e.g.*, hydrophobicity, charge, side-chain surface area,*etc.*. The most frequently used quantity has been hydrophobicity since many studies have been done attempting to characterize and identify amphipathic helices which are generally associated with transmembrane domains of integral membrane proteins. One important extension of the use of hydrophobic profiles is the concept of hydrophobic moment, introduced by Eisenberg. The hydrophobic moment is directly related to the Fourier power spectral density of the hydrophobic profile for a given sequence. Eisenberg and others have used this technique to identify and characterize long transmembrane helices in integral membrane proteins. The basic reason for using hydrophobicity in such studies is that amphipathetic helices (and membrane-spanning $\beta$-strands) exhibit periodicities in this quantity which can be captured by a linear transform, like Fourier.

There is however a more fundamental reason to utilize hydrophobicity as the physical parameter for constructing sequence property profiles. As Dill has pointed out, the major factor which drives the folding mechanism is the hydrophobic composition of the sequence. Of course, hydrophobic effects are not the only ones contributing to the folding process; other, such as hydrogen- bonding, van der Waals forces, electrostatic interactions,*etc.* clearly contribute to the detailed nature of the folded state of a particular sequence. The point is that if one wishes to capture as much as possible of the physics of folding in a single paramteter, then amino acid hydrophobicity is the candidate of choice.

A number of different hydrophobity scales have been developed over the past five to ten years; for our work, we considered the scales reviewed by Esposti, *et.al.* Recently Ponnuswamy and Gromiha have introduced a new hydrophobicity scale which incorporates topological context and which appears to have better discriminability with respect to structure variations than the usual scales. This *context-sensitive hydrophobicity* is shown in the following table, and it is the one which we have used in our investigation.

Once in a while, one encounters sequences where there is uncertainty about certain aminoacids. SWISS-PROT handles this by introducing three more entries: Asx (Aspartic acid or Asparagine), Glx (Glutamine or Glutamic acid) and Xaa (any amino acid). When we map sequence space to property space, we map Asx to the average properties of Asp and Asn, Glx to the average properties of Gln and Glu, and Xaa to the average property of all 20 aminoacids.

Now let $X_j(n)$ stand for a set of property profiles that belong to the sequences

| Aminoacid | Hydrophobicity | Aminoacid | Hydrophobicity |
|---|---|---|---|
| Ala | 13.85 | Met | 13.86 |
| Asp | 11.61 | Asn | 13.02 |
| Cys | 15.37 | Pro | 12.35 |
| Glu | 11.38 | Gln | 12.61 |
| Phe | 13.39 | Arg | 13.10 |
| Gly | 13.34 | Ser | 13.39 |
| His | 13.82 | Thr | 12.70 |
| Ile | 15.28 | Val | 14.56 |
| Lys | 11.58 | Trp | 15.48 |
| Leu | 14.13 | Tyr | 13.88 |

Table 1: Amino-acid hydrophobicity scale we use

of a certain class. The *unique signal* model suggests that if the sequence class is homologous, we can model the property profiles as:

$$X_j(n) = \theta_j + S(n) + \epsilon_j(n), \forall j$$

$S(n)$ has mean zero and is uncorrelated with $\epsilon_j(n)$ which also has mean zero. $\theta_j$ is just an appropriate shifting constant. So, this says that all the sequence profiles in the class, are distortions of a unique original sequence. $S(n)$ corresponds to that sequence and $\epsilon_j(n)$ corresponds to the distortion. Common classification mini-problems are comparing classes like that against each other, and also against a *random class*, that is, a class in which $S(n) = 0$.

To do this, it is necessary to transform $X_j(n)$ to what we will call a *feature space.* To understand why this is necessary, consider the analogous problem of classifying sounds produced by musical instruments. No two pianos produce exactly the same kind of sound. Still, we can tell sounds generated by pianos apart from sounds generated by other instruments that sound sufficiently different. You can look at the waveforms of all these sounds for the rest of your life, if you like, but to analyse them closely, people will look at the *Fourier transform* of the waveform. That is, they will map the signal in question to a *feature space* and work with that instead. When it comes to protein sequences, we have to do the same thing. The difference is that it is not intuitively obvious what feature space is most appropriate to use. There has been work done in this area, where the "first thing that comes to mind" was used: a Fourier Transform. In this research, we have opted to use an alternative transform called *Walsh Transform.* This is a discrete transform, in some ways similar to the Fourier Transform, in others different. The main difference is that F.T. is a continuous transformations that is more appropriate to continuous signals (sound for example). Protein sequences are not continuous signals however; they are discrete, so we think that it is more appropriate to use a transform that by definition is discrete, rather than a discretized continuous transform. An alternative feature space can be ARMA model parameters and that will be investigated in future research. We will discuss Walsh transforms in detail later. For now, assume that we have chosen a feature space, and that we have mapped our property profiles to that space:

$$\phi : X_j(n) \longrightarrow d_j(\lambda)$$

Depending on the feature space, $\lambda$ can be discrete or continuous, but we would like it to be discrete, and that is what it is in the case of Walsh transforms.

The "fingerprint" of a sequence class,as seen in some feature space,is the probability distribution functions that describe how the $d_j(\lambda)$ are distributed. If those functions don't overlap over different classes, then théy can be used to distinguish between classes. The problem is that we don't have those functions. We only have a sample of protein sequences for each class. So, we have to *estimate* them. This problem is non-trivial, and we will take some time to discuss it later in this paper. In any case, let

$$C(\lambda) = \{d_j(\lambda) : j \in \{1, \ldots, L\}\}$$

be the samples of our feature variables for different $\lambda$ and suppose that we have an algorithm that will map them to probability distribution functions:

$$\delta : C(\lambda) \longrightarrow p_\lambda(x)$$

What would these probability distributions look like? Well, this is mostly dependent on the feature space, and has a lot to do with whether the transformation involved is linear or not. Walsh transforms are linear. Another result that we will show later is that the fingerprint of a random sequence class in Walsh space is a gaussian distribution with mean zero and a varience that is dependent on the property we use (hydrophobicity) and the sequence length, but *indepedent* of $\lambda$. These *baseline* pdfs maybe different for other feature spaces, but that's what they are in Walsh space. The linear property, has the following effect. If, our class follows the unique signal model then:

$$d_j(\lambda) = \phi(\theta_j + S(n) + \epsilon_j(n)) = \phi(\theta_j) + \phi(S(n)) + \phi(\epsilon_j(n)) = U(\lambda) + Z_j(\lambda)$$

where we assumed that $\phi$ maps constants to zero (and the Walsh transform is nice enough to do that) and

$$U(\lambda) = \phi(S(n)) \qquad Z_j(\lambda) = \phi(\epsilon_j(n))$$

$Z_j(\lambda)$ being the transform of the noise term, will give us the baseline distribution, and the presense of a signal will merely *shift* it by an amount dependent on $\lambda$, i.e. $U(\lambda)$. For some $\lambda$ the shift will be great, for others negligible. The ones that give us the greatest shift are the ones that are most *sensitive* to the signal of the class. In non-linear cases, other things happen, but the idea is still the same. For some $\lambda$, the pdf will look very different from the baseline pdf, while for other it will look less different. At this point, notice what happens when we mix two classes that follow different unique signal models. For every $\lambda$, the first class will give rise to a gaussian peak, and the second class will give rise to a second gaussian peak. So, rather than obtaining a gaussian distribution, we will get a linear combination of two gaussians. One can test in this way whether a sequecne class is homogenous. This is yet another convenience that follows from the linearity assumption.

Let $X(n) \in H_0 \cup H_1$ be a sequence which is known to belong in either class $H_0$ or $H_1$, and $d(\lambda)$ the corresponding transform. $H_1$ usually corresponds to some sequence class, while $H_0$ is the so called *null hypothesis*, i.e. the random class. If we have two samples $X_{j,0}(n)$ and $X_{j,1}(n)$ ,representing $H_0$ and $H_1$, with corresponding pdfs $p_{\lambda,0}(x)$ and $p_{\lambda,1}(x)$, then we describe our decision algorithm as:

$$\forall \lambda \in \Lambda : p_{\lambda,1}(d(\lambda)) > \eta(\lambda) p_{\lambda,0}(d(\lambda)) \Longleftrightarrow X(n) \in H_1$$

$$\exists \lambda \in \Lambda : p_{\lambda,1}(d(\lambda)) \leq \eta(\lambda) p_{\lambda,0}(d(\lambda)) \Longleftrightarrow X(n) \in H_0$$

This decision test in known as the *Neyman-Pearson* test. There are two free parameters involved here that we have not defined, which control the behaviour of this decision algorithm:

- $\eta(\lambda)$ are the so called Neyman-Pearson *threshold levels* and they may only depend on $\lambda$ and the training samples $X_{j,0}(n)$ and $X_{j,1}(n)$
- $\Lambda$ is a collection of the most sensitive $\lambda$ values that we use to discriminate between the two sequence classes. Which values to use, and how many, are also free parameters.

In the introduction, we briefly mentioned *type I errors* and *type II errors.* A type I error occurs when we incorrectly reject the $H_1$ hypothesis. A type II error occurs when we incorrectly accept the $H_1$ hypothesis. The probability by which type II errors occur is known as the *false-alarm probability* and the probability by which type I errors **do not** occur is known as the *sensitivity* of the classifier. The training of our classification algorithm involved the computation of $p_{\lambda,0}(x)$ and $p_{\lambda,1}(x)$ from the sequence samples. That is the part where the algorithm learns what the two classes look like. Determining the remaining free parameters, will simply tune the algorithm to a desired false-alarm level, and maximized sensitivity.

To begin, consider the case where $\Lambda = \{\lambda\}$ , i.e we use only one transform component for classification. Also, let $\eta = \eta(\lambda)$ be the corresponding threshold. Then define:

$$D_\lambda(\eta) = \{x \in \Re : p_{\lambda,1}(x) > \eta\, p_{\lambda,0}(x)\}$$

This is the domain of values for which the Neyman-Pearson criterion chooses $H_1$. In other words $D_\lambda(\eta)$ is such that we can rewrite the Neyman-Pearson criterion in the following form:

$$X(n) \in H_1 \Longleftrightarrow d(\lambda) \in D_\lambda(\eta)$$

Notice that this is only a decision algorithm, and that there is a probability, $\epsilon_\lambda(\eta)$, that $X(n) \in H_0$ even if $d(\lambda) \in D_\lambda(\eta)$. That probability is given by integrating the null-hypothesis $(H_0)$ pdf over the domain $D_\lambda(\eta)$, and it is the *false alarm* associated with sequency $\lambda$ and threshold $\eta$:

$$\epsilon_\lambda(\eta) = \int_{D_\lambda(\eta)} p_{\lambda,0}(x)dx$$

We obtain the *sensitivity* in a similar manner by integrating $p_{\lambda,1}(x)$:

$$\sigma_\lambda(\eta) = \int_{D_\lambda(\eta)} p_{\lambda,1}(x)dx$$

These two parameters, the *false alarm ratio* $\epsilon_\lambda(\eta)$ and the *sensitivity* $\sigma_\lambda(\eta)$ are the key to picking an appropriate set $\Lambda$ and setting the threshold parameters $\eta(\lambda)$.

To determine $\eta(\lambda)$ and the $\lambda$ of choice for the simple case $\Lambda = \{\lambda\}$ given a desired false alarm level $f$ we use the following three steps:

- Solve the equation $\epsilon_\lambda(\eta(\lambda)) = f$ for $\eta(\lambda)$:

  $$\eta(\lambda) = \epsilon_\lambda^{-1}(f)$$

  where $\epsilon_\lambda^{-1}$ indicates the inverse function of $\epsilon_\lambda$.

- Find the most sensitive $\lambda$ by computing the sensitivity $s(\lambda)$ that corresponds to the $\eta(\lambda)$ computed in the previous step:

$$s(\lambda) = \sigma_\lambda(\eta(\lambda)) = \sigma_\lambda(\epsilon_\lambda^{-1}(f))$$

- Find which $\lambda$ maximises $s(\lambda)$. That determines the most sensitive choice of $\lambda$ and that along with $\eta(\lambda)$, uniquely defines our classification algorithm.

Next we discuss the case where $k$ transform components are being used. We need to worry about choosing the appropriate set of components

$$\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$$

as well as the appropriate thresholds:

$$\vec{\eta} = (\eta_1, \eta_2, \ldots, \eta_k)$$

The corresponding sensitivities and false-alarm rates as functions of $\vec{\eta}$ are given by:

$$\epsilon_\Lambda(\vec{\eta}) = \prod_{i=1}^{k} \epsilon_{\lambda_i}(\eta_i) = \prod_{i=1}^{k} \int_{D_{\lambda_i}(\eta_i)} p_{\lambda_i,0}(x)dx$$

$$\sigma_\Lambda(\vec{\eta}) = \prod_{i=1}^{k} \sigma_{\lambda_i}(\eta_i) = \prod_{i=1}^{k} \int_{D_{\lambda_i}(\eta_i)} p_{\lambda_i,1}(x)dx$$

Trying all possible permutations for $\Lambda$ and then calculating thresholds in an $k-$dimensional space can be a formidable problem. At this point in our investigation, we have taken the following shortcut:

- To achieve a false alarm rate of $f$, solve the 1 component problem for false alarm $\sqrt[k]{f}$. In other words, for all values of $\lambda$, we set the thresholds to:

$$\eta(\lambda) = \epsilon_\lambda^{-1}(\sqrt[k]{f})$$

For such thresholds, you can verify that we get the correct false alarm.

$$\epsilon_\Lambda(\vec{\eta}) = \prod_{i=1}^{k} \epsilon_{\lambda_i}(\eta(\lambda_i)) = \prod_{i=1}^{k} \epsilon_{\lambda_i}(\epsilon_{\lambda_i}^{-1}(\sqrt[k]{f})) = \prod_{i=1}^{k}(\sqrt[k]{f}) = f$$

- Pick a permutation for $\Lambda$ by computing

  $$s(\lambda) = \sigma_\lambda(\eta(\lambda)) = \sigma_\lambda(\epsilon_\lambda^{-1}\underline{\underline{(f)}})$$

  as in the one-component case. Then choose the $k$ best $\lambda$ that give $s(\lambda)$ the greatest values. These $\lambda$'s determine one of the best permutations for $\Lambda$. This along with the thresholds we computed in the previous steps, define one of the best $k$-component classifiers.

- Evaluate the classifier by calculating the theoretical sensitivity that is expected by

  $$\sigma_\Lambda(\vec{\eta}) = \prod_{i=1}^{k} \sigma_{\lambda_i}(\eta(\lambda_i)) = \prod_{i=1}^{k} \sigma_{\lambda_i}(\epsilon_\lambda^{-1}(\sqrt[k]{f}))$$

To understand what motivated the above method, think in terms of the following analogy. One has a set of sequences to pass through a classifier which will only permit a false alarm equal to 1%. In the one-component case, we just set the thresholds so that all components can individually serve as classifiers with 1% false alarm. Then we look at which component will yield the best sensitivity and we pick that one. In a two-component case, say we set the thresholds so that one component allows a f.a.p. of $f_1$ and another $f_2$. A two-component Neyman-Pearson classifier that uses these two components, means that a sequence must pass through two one-component tests: one that allows f.a.p. of $f_1$ and one that allows f.a.p. of $f_2$. Both tests put together will allow a f.a.p of $f_1 f_2$. So, if the thresholds are set so that each component will permit 10% false alarm, then both will allow only 1%. The argument for the general k-component case is similar. The weakness of this approach is that there may still be room for optimising the sensitivity by making some components more permissive (in terms of f.a.p.) than others. So, one should bear in mind, that we have not fully optimised our classifier construction and that we do not know how much room for more improvement we have.

To wrap this up, the above algorithms will provide us with hopefully the best possible classification algorithm *for every $k$* , along with an evaluation of it's sensitivity. The final touché is to pick the $k$ for which we obtain the best classifier. Maybe at this point, we may consider optimising this classifier as we discussed in the last paragraph, but that we will do some other time.

## III. Theory of Walsh Transforms

Training a classifier to distinguish two sequence classes, when we are given samples from both classes boils down to doing the following things:

- Map the sequences in the samples to property profiles which is trivial.
- Transform the property profiles to a feature space.
- Estimate the probability distribution functions by which the components of the property profile transform distribute themselves.
- Find a Neyman-Pearson test based on those pdfs that accomplishes the best possible sensitivity for a fixed false alarm.

In the previous section, the emphasis was on the fourth step, and the discussion we had there was indepedent of the feature space used. In this section we discuss the feature space of our choice: The Walsh transform.

In many aspects Walsh functions are similar to the systems of sines and cosines. However, unlike their trigonometric counterparts, the Walsh functions are square waveforms that take on only two values: $+1$ and $-1$. There are many ways to define the Walsh functions, and the one we shall adopt here is in terms of the *Hadamard matrices.* The Hadamard matrices are defined recursively as follows: begin with $H(0) = +1$, and then let

$$H(k+1) = \begin{pmatrix} H(k) & H(k) \\ H(k) & -H(k) \end{pmatrix}$$

so for example

$$H(1) = \begin{pmatrix} +1 & +1 \\ +1 & -1 \end{pmatrix}$$

and

$$H(2) = \begin{pmatrix} +1 & +1 & +1 & +1 \\ +1 & -1 & +1 & -1 \\ +1 & +1 & -1 & -1 \\ +1 & -1 & -1 & +1 \end{pmatrix}$$

and so on. The Hadamard matrix contains the Walsh functions as rows (or columns noting the symmetry). The order by which the functions are arranged is called *natural order* or *Hadamard order.* A better way to rearrange them is in *sequency order.* Sequency is defined as the number of sign changes that the walsh function makes. For instance, the first function in $H(2)$ has sequency 0, the next one has sequency 3, then 1 and finally 2. The walsh functions generated by $H(p)$ have length $N = 2^p$ and their sequencies can range to $s = 0, 1, 2, \ldots, N-1$. We will adopt the notation $W_N(s,i)$ for the walsh function of sequency $s$ generated from $H(p)$ where $i$ is the index of the sequence.So, for example

$$W_4(3,0)=+1 \quad W_4(3,1)=-1 \quad W_4(3,2)=+1 \quad W_4(3,3)=-1$$

From the symmetry of the Hadamard matrices, it follows that

$$W_N(s,i)=W_N(i,s)$$

Walsh functions have a some pretty interesting properties,the most important of which is the fact that for different sequences they are orthogonal to each other. Before proving this fundamental result we mention a couple of useful identities.

A result that can be proved by induction from the Hadamard definition is that:

$$s \neq 0 \Longrightarrow \sum_{i=0}^{N-1} W_N(s,i)=0$$

In other words, the "integral" (or sum) of the Walsh functions is zero for all but the zero sequency function.

Another result involves the concept of *dyadic sums.* Let $m$ and $n$ be two non-negative integers that have the following binary expansions:

$$m=\sum_{j=0}^{q} m_j 2^j \qquad n=\sum_{j=0}^{q} n_j 2^j$$

The *dyadic sum* $m \oplus n$ (also known as bit-by-bit mod 2 addition) is defined as

$$m \oplus n=\sum_{j=0}^{q} |m_j-n_j| 2^j$$

With this definition, it can be shown that

$$W_N(s_1,i)W_N(s_2,i)=W_N(s_1 \oplus s_2,i)$$

and since $s$ and $i$ are interchangeable by the symmetry property we also have

$$W_N(s,i_1)W_N(s,i_2)=W_N(s,i_1 \oplus i_2)$$

The orthogonality result says that

$$\sum_{i=0}^{i=N-1} W_N(s_1,i)W_N(s_2,i)=N\delta_{s_1,s_2}$$

where $\delta_{i,j}$ is 1 when $i=j$ and 0 when $i \neq j$. For a sort proof, consider the two cases:

- If $s_1 \neq s_2$ then

$$s_1 \oplus s_2 \neq 0 \Longrightarrow \sum_{i=0}^{i=N-1} W_N(s_1,i)W_N(s_2,i) = \sum_{i=0}^{i=N-1} W_N(s_1 \oplus s_2, i) = 0$$

- If $s_1 = s_2$ then

$$s_1 \oplus s_2 = 0 \Longrightarrow \sum_{i=0}^{i=N-1} W_N(s_1,i)W_N(s_2,i) = \sum_{i=0}^{i=N-1} W_N(0,i) = N$$

Because the Walsh functions are orthogonal, we can take any digital signal of length N and expand it like:

$$X(i) = \sum_{s=0}^{s=N-1} w(s)W_N(s,i)$$

$w(s)$ is called the *Walsh transform of* $X(i)$ and is given by:

$$w(s) = \frac{1}{N}\sum_{i=0}^{i=N-1} X(i)W_N(s,i)$$

The last equation describes the transformation of the property profile to Walsh space, that we briefly mentioned in the previous section. The $\lambda$ variable in this space is the sequency $s$ that appears in the walsh transform $w(s)$. It is also appearent from this formula that the Walsh transform is a *linear transformation* just like we claimed in the previous section. Now we know what a Walsh transform is, and how it's done.

At this point, a few points need to be made. First of all, notice that in order to be able to take the Walsh transform of a property profile, the length of the profile needs to be a power of 2, that is $N = 2^p$, because the Hadamard matrices $H(p)$ are $2^p \times 2^p$. Since protein sequences can have any length, it is necessary to pad them somehow to the proper length. The standard way of doing this is just padding the signal with zeroes. However we find it more appropriate to pad the sequence with a randomly generated signal. In other words, we append to the real sequence, a random sequence of aminoacids, and we take the profile of the padded sequence.

Moreover, a nice property that follows from the statement

$$s \neq 0 \Longrightarrow \sum_{i=0}^{N-1} W_N(s,i) = 0$$

is that if you shift the property profile by a constant, it leaves it's Walsh transform unchanged, except for the $s = 0$ component. Indeed, if the transform of $X(i)$ is $w(s)$, then the transform of the shifted signal $X(i) - a$ is:

$$w_1(s) = \frac{1}{N} \sum_{i=0}^{i=N-1} (X(i) - a) W_N(s,i) = w(s) - \frac{a}{N} \sum_{i=0}^{i=N-1} W_N(s,i) = w(s) \ , \ \ \forall s \neq 0$$

Once the sequence is mapped into a property profile $X(n)$ we subtract the average. This makes $w(0) = 0$ but all other Walsh coefficients are unchanged.

Finally, we need to discuss how the walsh transform components distribute themselves over a sample of the random class. One way to think of a Walsh transform component is as the result of a random walk; every term in the summation we can think of as a step. The step size can have one of 20 values, out of our hydrophobicity scale. The fact that Walsh functions take only values $+1$ and $-1$ is also convenient because we can interpret this all as taking a step forward or a step backwards. The complication arises from the fact that

$$\sum_{i=0}^{N-1} W_N(s,i) = 0 \ \ , \ \forall s \neq 0$$

If we interpret every term in the sum

$$w(s) = \frac{1}{N} \sum_{i=1}^{N} X(i) W(s,i)$$

as a step, then we are taking as many steps forward, as backwards. This is not what happens in a random walk, so we are not quite there yet! Suppose now that we take a permutation of the terms such that the $+1$ and $-1$ terms alternate:

$$w(s) = \frac{1}{N} \sum_{i=1}^{N/2} (X_+(i) - X_-(i))$$

Now, if we consider these $N/2$ terms as steps in walk process,it will indeed be a random walk! To see, why this is so, let

$$H = \{h_1, h_2, \ldots, h_n\}$$

be our property scale. The $N/2$ terms in the summation are drawing values out of the set:

$$S = \{\zeta_{ij} = h_i - h_j : h_i, h_j \in H\}$$

These are the steps of walk, and some of the steps are forward steps, some of them are backwards. On average, we have:

$$\hat{\zeta} = \sum_{i=1}^{n}\sum_{j=1}^{n}\zeta_{ij} = \sum_{i=1}^{n}\sum_{j=1}^{n}(h_i - h_j) = \sum_{i=1}^{n} h_i - \sum_{j=1}^{n} h_j = 0$$

so the chances for forward steps balance the chances for backwards steps. The root-mean-square step, be it forward or backwards is:

$$\sigma_{rms}^2 = \frac{1}{n^2-1}\sum_{i=1}^{n}\sum_{j=1}^{n}(h_i - h_j)^2$$

Our walsh transform is equivalent to doing a walk drawing steps out of the set $S$. We see now, S elements have average zero, which means the walk, is equivalent with a random walk in which we take steps of size $\sigma_{rms}$ either forwards or backwards. The outcome of such a process is gaussian distributed with zero mean, so we can approximate the distribution of the walsh coefficients over a random sequence class by:

$$p_\lambda(x) = \frac{e^{-x^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

In general, for a random walk with step $\sigma_{rms}$, for $N$ steps the varience $\sigma$ is $\sigma = \sigma_{rms}\sqrt{N}$. Taking into account that in the Walsh transform, we divide the sum with $N$, we have that

$$\sigma^2 = \frac{\sigma_{rms}^2}{2N} = \frac{1}{2N(n^2-1)}\sum_{i=1}^{n}\sum_{j=1}^{n}(h_i - h_j)^2$$

Notice that all this is only an approximation: Our expression for $p_\lambda(x)$ spans $(-\infty, +\infty)$ and is continuous while in reality, the walsh components for a fixed length can't take more distinct values than there are permutations for sequences on that given length (and it follows from this that walsh components are bounded, so their distribution function can't really span the entire line of real numbers).

In any case, it is a very very good approximation, so we found necessary to verify this result by *experimentally* estimating the standart deviation $\sigma$ for various length over our hydrophobicity property map. The estimate was done by repeating the following steps

- We generate a random sequence of length $N$, take the transform of it's property profile and pick a random component out of the transform. Repeat and generate a sequence $w_j$ of a sufficient number $n$ of components.

- Calculate the mean and varience of $w_j$:

$$\mu = \frac{1}{n}\sum_{j=1}^{n} w_j \qquad \sigma^2 = \frac{1}{n-1}\sum_{j=1}^{n}(w_j - \mu)^2$$

- Repeat the previous step many times to collect values of $\mu$ and $\sigma$. Then, estimate, their average and standard deviation (error).

The mean $\mu$ should have greater standard deviation than average value (which indicates that it's on it's way to converge to zero) and the result we get for $\sigma$ is the standard deviation that appears in our approximation of $p_\lambda(x)$. We have performed the above procedure numerically. We computed $\mu$ and $\sigma$ 70 times and we looked at 8000 components $w_j$ for every iteration. The experiment's results are shown in this table:

| length | theoretical | exper. sigma |
|---|---|---|
| 32 | 0.204985 | $0.207958 \pm 0.001729$ |
| 64 | 0.144946 | $0.147225 \pm 0.001046$ |
| 128 | 0.102492 | $0.104137 \pm 0.000855$ |
| 256 | 0.072473 | $0.073555 \pm 0.000501$ |
| 512 | 0.051246 | $0.051984 \pm 0.000410$ |
| 1024 | 0.036237 | $0.036736 \pm 0.000294$ |

Table 2: Standard deviations for random class walsh pdfs

This table is very useful, when we want to evaluate the performance of the algorithms that estimate the distribution function of the walsh components over a sequence class, based on a sample of sequences. What we do, is that we try the algorithm on a random class, and see what it says. In a random class, we know that the pdfs should all show up as gaussians with mean zero, and standard deviations equal to the values in the table above, so we know what the answer is, and we can see how close to the answer, our estimator gets. Which brings us to the next topic: How to calculate these probability distribution functions?

## IV. Estimating the Probability Distribution Functions

As we said earlier, the part in the classification algorithm training, where the algorithm learns what the two sequence classes look like is when we estimate the distribution function $p_\lambda$ of the $\lambda$ transform component of the sequences' property profile, over some feature space. The problem is formulated as finding an estimate of the probability distribution function that would generate the stochastic variable sampled by the set:

$$C = \{d_j(\lambda) : j \in \{1, 2, \ldots, n\}\}$$

To find a good way to address this problem, we did a lot of trial and error experimentation of various ideas. In this section we distill the conclusions of these efforts and conclude by formulating the algorithm we used.

When the collection C is large (say 10000 entries) one can get a very good estimate of the probability distribution, in a very straightforward way:

- Determine the smallest and largest values $\mu$ and $M$ in the set $C$. We assume then that our distribution function takes values in the interval

  $$I = (\mu, M) \ , \qquad \mu = \min(C) \quad M = \max(C)$$

  and is zero outside that interval.

- Split the interval to $r$ subintervals (where $r$ stands for *resolution*)

  $$I_n = (\mu + n\ell, \mu + (n+1)\ell), \forall n \in \{0, 1, \ldots, r-1\}$$

  $$\ell = \frac{M - \mu}{r}$$

- Count how many of the variables in the sample $C$ lie within each $I_n$ interval and define a sequence of numbers $f_n$ based on that count as:

  $$f_n = |C \cap I_n|$$

  where the notation $|A|$ is the cardinality of the set $A$ (in simple terms, how many elements there are in the set).

- The numbers $f_n$ can be graphed as a histogram which sketches out the shape of the distribution function that generated the variables we sample. To obtain an estimate for the distribution function itself, we normalise $f_n$ by multiplying it with an appropriate constant, so that

  $$\sum_{n=0}^{r-1} f_n \ell = 1$$

  and use an interpolation method to extract values for $p_\lambda$.

The resolution $r$ is a free variable, and we must pick an appropriate value for it. A good rule of thumb is to set r equal to $\sqrt{|C|}$ if that is possible. In selecting $r$, the trade off is between introducing artifacts to the pdf's shape when one uses too high a resolution, to having serious errors when integrating the pdf, when the resolutionis not sufficient. If it were at our hand to have a sample $C$ as big as we want it, then this discussion would stop right here. In real life, nice sequence samples are about the order of a thousand or two, and there are other less nice samples that are about the order of a couple of hundred. In general, it is nice to have 100 points $f_n$ to interpolate with, and if we only have a few hundred samples, then every $f_n$ will only get to count 3 or 4 counts at the most, which is little good.

To deal with this problem, we modify the above algorithm by using the *expanded interval*

$$J_{n,w} = (\mu + (n - w)l, \mu + (n + 1 + w)l), \quad \forall n \in \{0, 1, \ldots, r-1\}, \quad \forall w \in \{1, 2, \ldots\}$$

to obtain our points $f_n$, now by:

$$f_n = |C \cap J_{n,w}|$$

A second variable, $w$ , is introduced in the game, that we choose to call *blurring.* A way to think about what we are doing here is this. Previously, we splited our big interval to pieces, and counted how many "peas" we have in every interval. This can be visualised with a little window interval $I_n$ hopping accross $I$, with hops that are as long as the window itself. Well, since 10 hopes are not any good when dealing with a 300 sequence sample, we would rather move our window smoothly accross the interval $(\mu, M)$, and hope that this will actually get to milk some more information out of the sample. Of course, smoothly just means that we split the interval into a higher resolution (still denoted as $r$), but scroll on it an interval of length larger than just $\ell = (M - \mu)/r$, $(2w+1)\ell$ in particular. Our experimentation with this technique, shows that not only does it do something about the problem of small samples but also smooths out somewhat the statistical fluctuations that you would expect to have on the histogram.

This method is similar in spirit to extrapolation methods and asymptotic expansions, where you do not have the information you would like to have, and you resort to squeezing it out. The problem with these methods is that if you get greedy they become counterproductive. The principle here is that since we do not have enough samples on the spot where we want them, we count in the samples of neighbohring spots as an estimate of how many samples we *would* have on the spot, if we had a bigger $C$ to begin with. The idea becomes absurd however when one starts to count

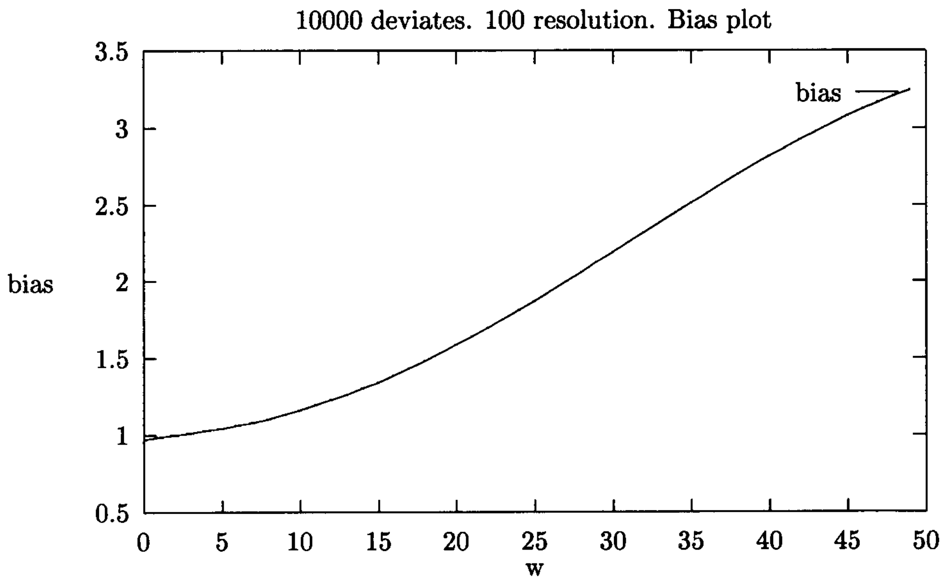


Figure 1: Bias vs the $w$ variable

in things that are far and away from the spot. To get a feel of how far we can go before we get greedy, we performed the following experiment:

- We generated a sample of 10000 numbers that follow a gaussian distribution of mean 0 and varience 1. So, we know the answer.
- We estimate the pdf of the data set, using different values for $w$ as discussed above, and we look at the ratio of the peaks of the real one , to the estimated one.

As one increases $w$, the estimated gaussian opens up and it's peak begins to fall. In technical terms, we say that it becomes *biased.* Following is a plot of the bias (real gaussian peak divided by estimated peak) versus $w$:

By inspecting plots like this we came to the conclusion that we can afford to set $w$ to be up to 5% of the resolution $r$.

To enhance the estimated pdf we pass $f_n$ through a *smoothing filter.* The filter we chose to use, is known as *Savitzky-Golay.* In a nutshell, for every data point $f_i$ we consider a couple of neighbohring points $f_{i-n_L}, \ldots, f_{i+n_R}$, fit an M order polynomial (typically quadratic or quartic) to those points and set the corresponding smoothed $g_i$ to the value of the polynomial at position $i$. It can be shown that there exit coefficients $\beta_n$, $n \in \{-n_L, \ldots, n_R\}$ that are dependent only on $n_R, n_L$ and the order $M$ for which we can do the above procedure magically by evaluating:

$$g_i = \sum_{n=-n_L}^{n_R} \beta_n f_{i+n}, \quad \forall i \in \{0, \ldots, r\}$$

Since $f_i$ is only defined within $\{0, \ldots, r\}$ we extend it by setting:

$$f_n = f_0, \quad \forall n < 0$$

$$f_n = f_r, \quad \forall n > r$$

We use quadratic order, and we set $n_L = n_R = s$ where $s$ is some value. So, the smoothing step, introduces yet another variable, $s$ that is to be toyed with. We discuss the filter in some more detail in Appendix B.

To put it all together, the algorithm that maps $\delta : C \longrightarrow p(x)$ consists of the following steps:

- Determine the min and max variables in the set:

  $$\mu = \min(C) \quad M = \max(C)$$

  and let the interval $I = (\mu, M)$.

- For chosen $(r, w)$ make a preliminary histogram by:

  $$J_{n,w} = (\mu + (n - w)\ell, \mu + (n + 1 + w)\ell)$$

  $$f_n = |C \cap J_{n,w}|, \quad \forall n \in \{0, 1, \ldots, r - 1\}$$

- Use some interpolation scheme to increase the resolution of the histogram somewhat up to $R$.

- Smooth $f_n$ and obtain a new histogram $g_n$.

- Normalise $g_n$

- Interpolate $p(x)$ from $g_n$.

Overall, our estimation algorithm can be written as $\delta(r, w, R, s)$ because it's those four variables that it is directly dependent on.

Of course, now we are faced with the tedious puzzle of tuning these four variables to make the algorithm work. Our take on this is to use the algorithms on samples where we know what the answer is. Such samples are sets of randomly generated sequences. We know that the walsh transforms of the profiles of such samples will give rise to zero-mean gaussian pdfs, and we have also measured experimentally the expected sigma $\sigma$ of these gaussians for various sequence lengths. If we apply the $\delta(r, w, R, s)$ algorithm on such a sample, we obtain probability distributions $p_\lambda(x)$. These are remembered by the computer by their values at a couple of points $x_n$ and to measure how much they deviate from a gaussian, we use the expression:

$$\chi^2(\lambda) = \frac{1}{N-1} \sum_{n=1}^{N} \left\{ \frac{(p_\lambda(x_n) - g_\sigma(x_n))^2}{g_\sigma(x_n)} \right\}$$

$$g_\sigma(x_n) = \frac{e^{-x_n^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

What happens in practice, is that for some $\lambda$ we get distributions that are closer to a gaussian than others. So the final number $\chi$ is the average of $\chi(\lambda)$ over all values of $\lambda$. We will discuss case studies of this later.

## V. Application to artificial model sequences

The last 3 sections covered in detail how one constructs classification algorithms that will classify two sequence classes from each other, and in particular, a sequence class against the class of random sequences. Throughout our analysis, we have modeled a sequence class by

$$X_j(n) = \theta_j + S(n) + \epsilon_j(n), \forall j$$

so, we decided to test our methods on model sequences that have been generated according to this signal plus noise model. To generate such a class, we create a random protein sequence and then we collect *mutations* of that sequence. We haven't gotten very involved in the way the signal sequence is mutated; we simply pick some random amino-acids on the sequence and substitute them with another random aminoacid. The amount of amino-acids that get changed is fixed and we will briefly refer to it as *distortion* of the signal. So, when we talk of a signal class with distortion 0.3, we refer to a sequence class where every member differs from the original signal sequence only on 30% of it's amino-acids. A class of distortion 1 is indistinguishable from a random class. In other words, the higher the distortion, the "closer" we are to the random class. This way, we can see how the classification ability of our Walsh technique depends on how close we are to the random class.

We have examined sequences with lengths 128 and 256. We pursued a false alarm probability of 1% and the algorithm we used to estimate the pdfs of the signal group was:

$$\delta(r, w, R, s) = (50, 4, 100, 4)$$

In our experiments, we have generated *two* signal classes out of the same signal sequence with the same distortion. One is to train the classification algorithms, the other is to test them. We found it unnecessary to generate a random class as well, since we know well in advance the probability distribution functions of walsh coefficients for random classes. Using our training set, we create the best 1-component, 2-component, 3-component, etc. classification algorithms and we test every sigle one of them. For every algorithm, testing means that we obtain the following variables:

- The *theoretical sensitivity* is the sensitivity that we expect from classification algorithm. It is calculated with the assumption that the Walsh transform coefficients are uncorrelated (which could be a poor assumption) by multiplying the sensitivities of the individual Walsh components that are used in the algorithm.

- The *verified sensitivity* is the sensitivity we obtain by applying the algorithm on the *training group!*. In an ideal world it should be the same as the theoretical one, but there is some disagreement, and most probably it indicates that the Walsh transform components are correlated somewhat.

- The *experimental sensitivity* is the sensitivity we obtain by applying the algorithm on the *testing group!*. Here, we expect some more disagreement, because the training set is only a small sample of the entire sequence class and there is some error in how it gives our algorithm a "feel" of what the sequence class itself looks like.

- The *experimental false alarm rate* is the probability that a random sequence will be incorrectly accepted by our classifier. To calculate it, we generate a huge amount of random sequences, feed them to the classifier and count how many get through. A decent amount to test a theoretical false alarm of 1% would be 2000 sequences, but for better statistics, 8000 sequences or even 10000 sequences would be a better choice (although more CPU intensive and 5 times slower). The experimental fap tells us among other things, if the thresholds have been set correctly. If it persists on being smaller than the theoretical fap, then this tells us that the thresholds have been set too high. If it persists on being bigger

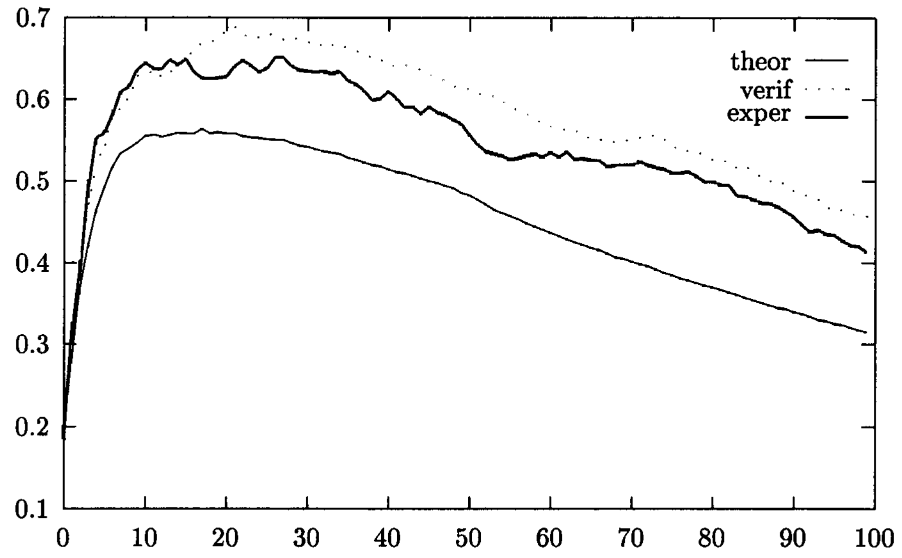


Figure 2: theoretical, verified and experimental sensitivities vs number of components for a signal class with distortion 0.5 and length 128

> than the theoretical fap, then this tells us that the thresholds have been set too low. If the Walsh components happen to be correlated for some reason, then the thresholds will not be set correctly, and that we will know from the experimental fap.

These variables are obtained for 1-component, 2-component, etc. algorithms, so the result is a plot of these variables against the number of components used. These plots are very important so to understand them, let's take a look at a specific case.

We study a case where 1000 sequences of distortion 0.5 and length 128 are being compared against the random class. In Figure 2 you can see the shape of the sensitivity curves against the number of components.

The first thing to see is that all three curves start out from almost the same point and disagreement begins to show up as one adds more components. This suggests that the 1-component algorithm is calculated with very good accuracy. We can also assume that it follows that in multiple component algorithms, the individual Neyman-Pearson thresholds are correctly set to yield a false alarm of $f^{1/k}$ ($f$ is the desired false alarm rate, and $k$ is the number of components in the algorithm). The most possible

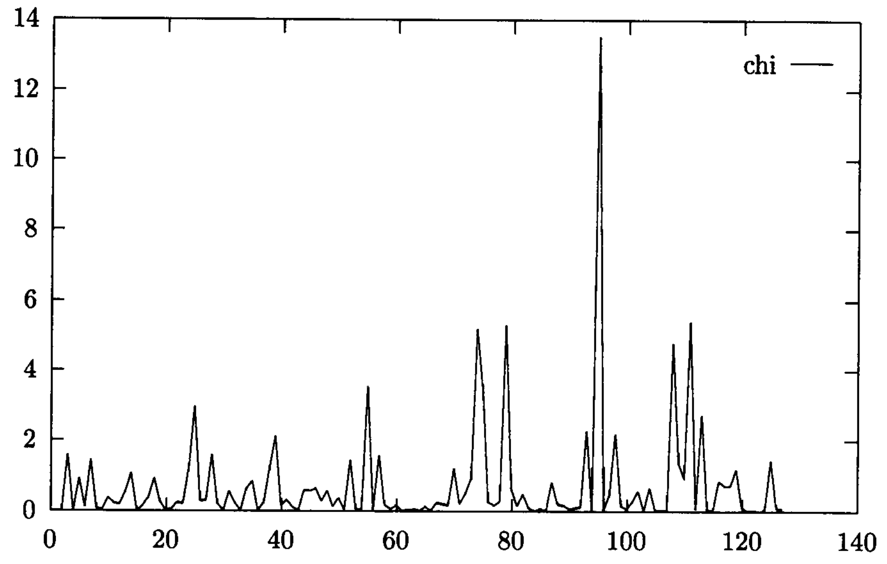
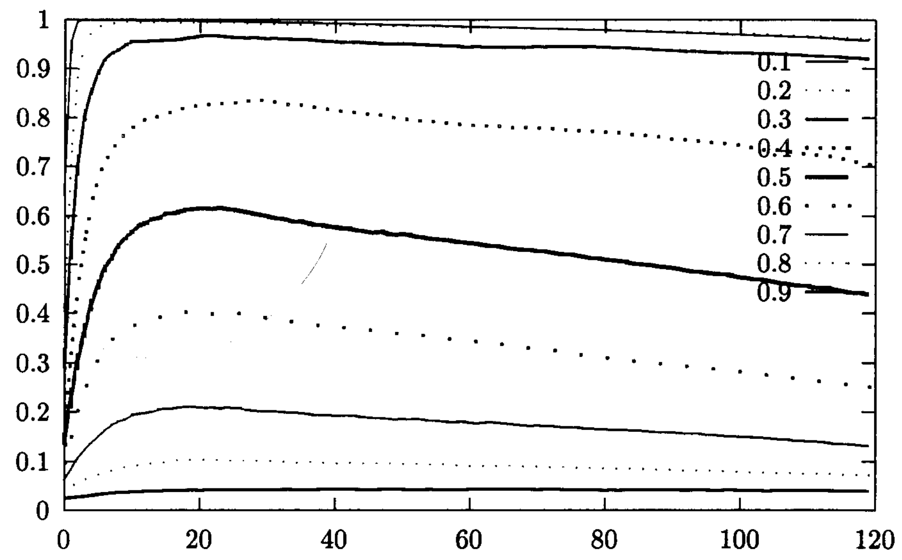


Figure 3: $\chi^2$ against sequency for our case. About 10 sensitive components

explanation for the deviation that builds up is that the components are correlated. Another source of error for very large $k$ could be the round off error in evaluating products of sensitivities as well as $f^{1/k}$ itself.

Also notice that as we add more components sensitivity improves, until about 10 components, and then it starts to decline. This means that there are about 10 very sensitive components and every one of them contributes new information to the classification algorithm, so performance improves. Once we added the good components though, adding insensitive components makes it worse. If we compute $\chi^2$ for every sequency, from the probability distribution functions, we can see by inspecting a $\chi^2$ vs sequency plot that there are indeed about 10 sensitive components. For the specific case we looked at, that plot is shown in Figure 3. Of cource, once all this is known, we make sure, as we explained before, to choose the number of components that optimizes the sensitivity of our classifier.

We have conducted multiple experiments like this one, for different sequence lengths and distortions. In all cases, we used 1000 sequences for the signal classes and we computed the false alarms on 2000 random sequences. Figure 5 shows how the performance of our classification algorithm is affected by changes in sequence length and distortion. Moreover, Figure 4 shows the theoretical sensitivity curves for



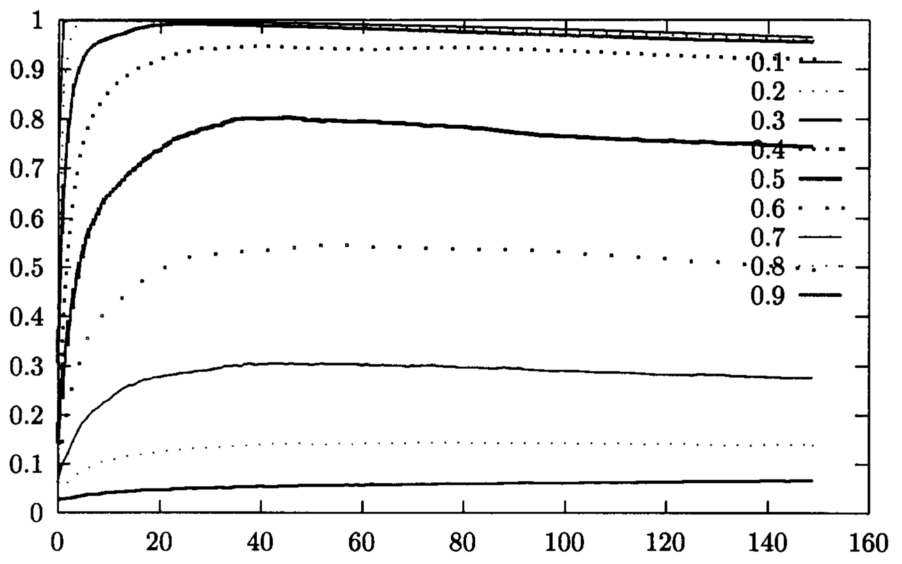


Figure 4: Theoretical sensitivities vs number of components for sequence length 128 (first plot) and 256 (second plot)

| distortion | t.sens | comps | v.sens | e.sens | fap |
|---|---|---|---|---|---|
| 0.1 | 0.998904 | 4 | 0.996000 | 0.993000 | 0.001500 |
| 0.2 | 0.995215 | 10 | 0.991000 | 0.985000 | 0.004500 |
| 0.3 | 0.967213 | 23 | 0.963000 | 0.927000 | 0.003000 |
| 0.4 | 0.834502 | 30 | 0.862000 | 0.856000 | 0.014000 |
| 0.5 | 0.616119 | 24 | 0.727000 | 0.725000 | 0.016000 |
| 0.6 | 0.402632 | 19 | 0.494000 | 0.498000 | 0.012000 |
| 0.7 | 0.210929 | 19 | 0.290000 | 0.297000 | 0.016500 |
| 0.8 | 0.104019 | 18 | 0.163000 | 0.144000 | 0.011000 |
| 0.9 | 0.043589 | 43 | 0.074000 | 0.056000 | 0.020500 |

| distortion | t.sens | comps | v.sens | e.sens | fap |
|---|---|---|---|---|---|
| 0.1 | 0.999068 | 5 | 0.992000 | 0.994000 | 0.003000 |
| 0.2 | 0.997830 | 6 | 0.996000 | 0.974000 | 0.005000 |
| 0.3 | 0.991159 | 27 | 0.970000 | 0.959000 | 0.002000 |
| 0.4 | 0.946807 | 41 | 0.932000 | 0.897000 | 0.010000 |
| 0.5 | 0.803454 | 46 | 0.846000 | 0.783000 | 0.016500 |
| 0.6 | 0.544741 | 53 | 0.604000 | 0.615000 | 0.022500 |
| 0.7 | 0.305358 | 39 | 0.423000 | 0.389000 | 0.027500 |
| 0.8 | 0.145011 | 76 | 0.258000 | 0.181000 | 0.024500 |
| 0.9 | 0.065964 | 150 | 0.109000 | 0.075000 | 0.034500 |

Figure 5: The first table corresponds to sequence length 128. The second table corresponds to sequence length 256. t.sens is the theoretical sensitivity, v.sens is the verified sensitivity, e.sens is the experimental sensitivity and fap is the false alarm probability. Also, comps is the number of components that yielded the best classifier.

all these cases. Note that the signal plus noise sequence classes that were used in this experiment were generated as a stochastic process. So, if the experiment is to be repeated with different sequence classes, we might expect the resulting sensitivities to fluctuate somewhat. To assess the significance of these fluctuations, we did a very careful investigation of sequences with length 128, where for every distortion, we repeated the experiment 10 times, with a different signal sequence each time. Our data are shown in Appendix B, and they suggest that the fluctuation in sensitivity is roughly about 4%. Keeping always in mind these fluctuations, what does Figure 5 tell us?

First, we see that as we increase the distortion, the sensitivity decreases. This of course happens because increasing distortion brings the signal class "closer" to the random class, which makes classification harder. Second, look at the disagreement between the theoretical and verified sensitivities. We see that the disagreement tends to be greater when the experimental fap deviates from the theoretical fap. And the experimental fap in turn seems to deviate more when the number of components used grows! This tells us that there are two possible reasons why the theoretical and verified sensitivity disagree: One, because of a possible correlation between Walsh transform components, something that can be checked and needs to be checked. Two, because of little errors in setting the thresholds for each component that magnify themselves for lots of components. To attack the first source of error, we would need to decorrelate the Walsh components, which really means that we would need to use a *different feature space*, one that is a "rotation" of Walsh space. With the second source of error, we could use the experimental false alarm rate itself (instead of the theoretical fap which we compute by integrating pdfs) to set the thresholds more accurately. The obstacle with doing this is that calculating the experimental fap is many times more CPU intensive than doing the theoretical fap, and things are slow enough as they are. Our line of attack would be then to set the thresholds roughly first and then refine them using experimental faps but that would still be a lot of work. Dealing with all this is still an open field for further research. Also, notice that the disagreement between the verified and experimental sensitivity doesn't seem to correlate with anything else, and is pretty much random. It is also not very bad, when compared to the disagreements with theoretical sensitivity in some cases. This means that if we can make the theoretical and verified sensitivity agree better, the experimental sensitivity will be dragged along into agreement as well! Finalle, these observations above do not contradict the additional data in Appendix B.

One last observation that will be interesting to explain is the effect of sequence length on the classifier performance. Notice that the 256-long sequence classes perform better than the corresponding 128-long ones. Why? To explain, we need to quantify the "distance" between the signal class and the random class. The way to visualize any protein sequence of length $N$ is as a point in an N-dimensional cubic lattice with 20 points for every edge. The random class draws points randomly from the entire lattice, while the signal class draws points only from a subset of the lattice: the mutations of a given signal. Let's give these things names: let $\mho(N)$ be the entire N-dimensional lattice, and $\wp(\sigma, m, N)$ all possible mutations of the signal $\sigma \in \mho(N)$ where $m$ out of $N$ aminoacids have been mutated. The idea of a classification algorithm is that by mapping into a feature space these points will pack themselves around a close cluster, if the feature space is nice enough to do that. Regardless of the feature space, we can say with certainty that the clustering will be easier if there are fewer mutations for the signal sequence, and we should expect better sensitivities when there are fewer mutations in $\wp(\sigma, m, N)$. We quantify how many mutations are out there by a metric that we will call *clustering* and define as:

$$\xi(m, N) = \log_{10} \frac{|\wp(\sigma, m, N)|}{|\mho(N)|}$$

Taking the logarithm is necessary, since these numbers tend to be big. The bigger $\xi(m, N)$ the closer the signal class is closer to the random class. A combinatorics argument suggests that:

$$|\mho(N)| = 20^N$$

$$|\wp(\sigma, m, N)| = 19^m \frac{N!}{m!(N-m)!}, \forall \sigma \in \mho(N)$$

so it follows that

$$\xi(m, N) = m \log_{10} 19 - N \log_{10} 20 + (\log_{10} N! - \log_{10} m! - \log_{10} (N-m)!)$$

Plotting this for $N = 128$ and $N = 256$ we see in Figure 6 that longer sequences cluster better, and therefore, should yield better sensitivities, which is exactly what we have observed.

| Distortion | $\xi(m, 128)$ | $\xi(m, 256)$ |
|---|---|---|
| 0.1 | -133.02 | -265.26 |
| 0.2 | -107.03 | -213.16 |
| 0.3 | -84.58 | -168.20 |
| 0.4 | -64.79 | -128.58 |
| 0.5 | -47.31 | -93.62 |
| 0.6 | -32.05 | -63.11 |
| 0.7 | -19.11 | -37.26 |
| 0.8 | -8.82 | -16.74 |
| 0.9 | -2.08 | -3.37 |

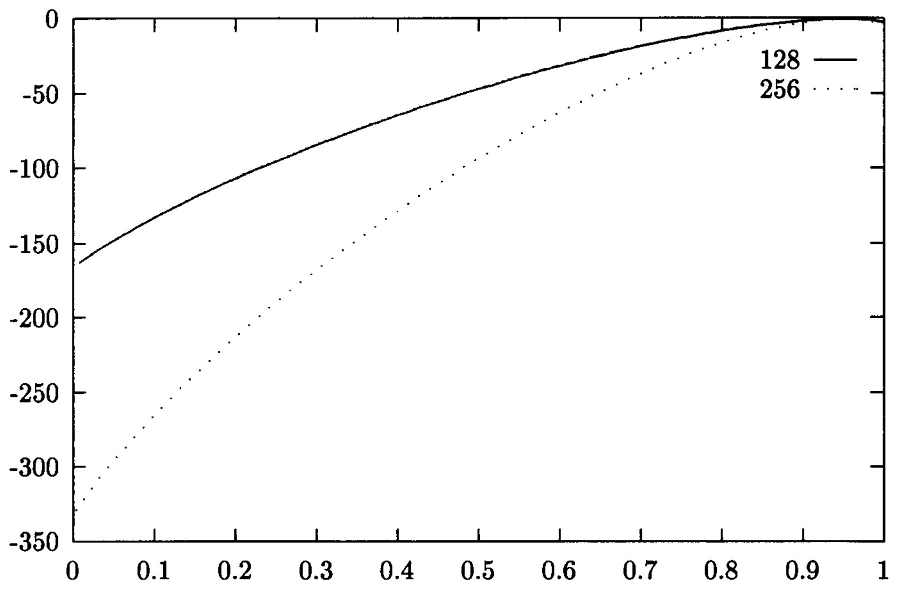


Figure 6: Clustering plotted against distortion for sequences of length 128 and 256. The longer the sequence, the better the clustering

## VI. Application to real sequence classes

The purpose of our work is to evaluate the Walsh transform as a classification feature space for protein sequences, and seek ways to optimise it. We have applied our techniques on artificial sequences because with those we can do experiments where we have complete control of the situation. We have also done this to get a feel of what's the "normal" behaviour of the classifier performance, to compare it with what we see in real sequences. Now we discuss our observations with real sequences.

When applying a classification algorithm to a real sequence class, there are two reasons why we might see poor performance. Either the algorithm is not good, or the sequence class is not good. The last bit means that the sequence class is so broad that it includes enough different things to make it look like very random. Unfortunately with our protein sequence databases, this is more true of sequence groups that would appear appealing because they have a large sample set. Smaller groups of sequences while they may offer less samples very often offer a better set of real sequences to work with. In this study we will discuss our observations on two such groups: the *calcium-binding proteins* and the *hemoglobin proteins.* Both groups, contain sequences that are very similar structurally, amongst themselves.

As with artificial sequence experiments, we plot the same good old *theoretical* and *verified sensitivity* against the number of components used. However *experimental sensitivity* acquires a new meaning. In artificial sequence models, we had a training and a testing set. While we could split our real sequence class also to training and testing sets, we definetely don't want to do this with calcium-binding sequences because we have too few samples. Moreover, we believe that our attention should be focused on another more important issue: As we explained earlier, to take the Walsh transform of any signal, the signal needs to have length $2^p$ for some integer $p$. To make it so with real sequences, we pad them to the next appropriate length with a random protein sequence. When we train our classification algorithms, we use a randomly generated padding, and plot the theoretical sensitivity curves. Then we feed the real sequence class *with the same padding as during training* to our classification algorithm. The count of sequences that get through is the *verified sensitivity.* But what about the padding? How will the sequence class be perturbed in the eyes of our classifier training techiniques when it's repadded differently? To get a handle on that we repad the sequence class and feed it to our trained classification algorithm. Thusly we obtain our new sense of *experimental sensitivity.*

## • Calcium-Binding sequence class

The calcium-binding group contains sequences that can have lengths that range from an order of 128 to an order of 1024. The most populated length subgroup are sequences of length between 128 and 256. In our database, we have included 132 such sequences, all padded to sequence length 256. This is a tightly small sample, and one needs to be careful with estimating the walsh components' distributions. To test if a pdf estimation algorithm is appropriate for a given sample size, we apply it on a randomly generated sample of that size, and compared the estimated pdfs with the *known* theoretical pdfs using the $\chi^2$ metric we defined earlier. Normally, we obtain a different $\chi^2$ for every Walsh component. To narrow down the information we look at the mean and standart deviation of $\chi^2$ over the set of Walsh components. The algorithm that gives as a small mean and standart deviation for $\chi^2$ is the algorithm of choice. In Figure 7 we list a few algorithms. Our algorithm of choice was: $\delta(r, w, R, s) = (4, 50, 100, 4)$

As we discussed earlier, we can determine the most sensitive walsh components in advance by doing a $\chi^2$ comparison of the calcium-binding pdfs with the gaussian distribution of the random sequence class. That comparison gave us $\chi^2(1) = 111.98$ for the sequency 1 component and $\chi^2(128) = 7$ for the next more sensitive component. This is not the behaviour we have observed with artificial sequences.(see Figure 3 ) Here we see that for some reason, most of the classification information is concetrated only on the sequency 1 component. Figure 8 shows who the most sensitive components are after 1 and 128, and Figure 9 shows the probability distribution functions for the sequency 1 component. Notice that a different padding of the sequence class will probably give somewhat different values for $\chi^2$ but what we said still stands.

| $\delta(r,w,R,s)$ | $\chi^2$ | $\delta(r,w,R,s)$ | $\chi^2$ |
|---|---|---|---|
| (4,40,100,4) | $0.118744 \pm 0.104488$ | (4,50,100,4) | $0.121506 \pm 0.088179$ |
| (3,40,100,4) | $0.125527 \pm 0.094695$ | (5,50,100,4) | $0.115122 \pm 0.093365$ |
| (2,40,100,4) | $0.153314 \pm 0.097496$ | (6,40,100,4) | $0.154060 \pm 0.153089$ |
| (5,40,100,4) | $0.128439 \pm 0.125849$ | (4,40,100,3) | $0.119185 \pm 0.103637$ |
| (4,30,100,4) | $0.150853 \pm 0.148302$ | (4,40,100,2) | $0.119955 \pm 0.102931$ |
| (4,20,100,4) | $0.361463 \pm 0.289077$ | (4,40,100,1) | $0.121027 \pm 0.102401$ |

Figure 7: $\chi^2$ against pdf estimation algorithm configurations

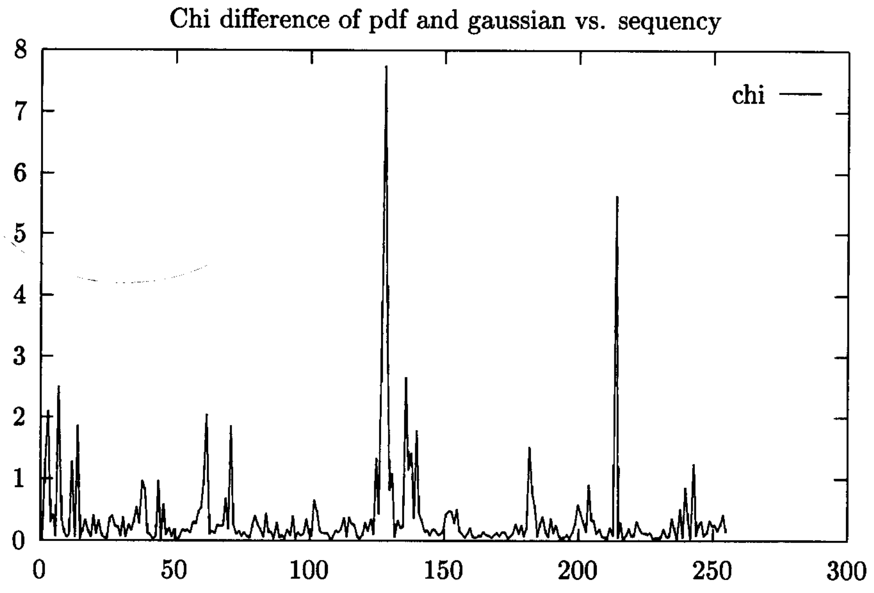


Figure 8: $\chi^2$ comparison of calcium-binding class and random class against sequency. We omit sequency 1, since it is more sensitive by some orders of magnitude ($\chi^2(1) = 111.98$)

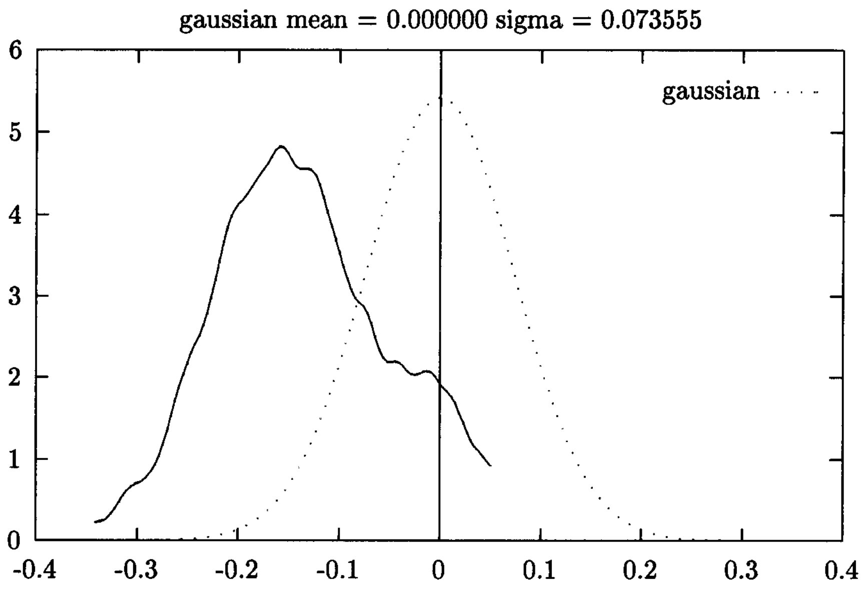
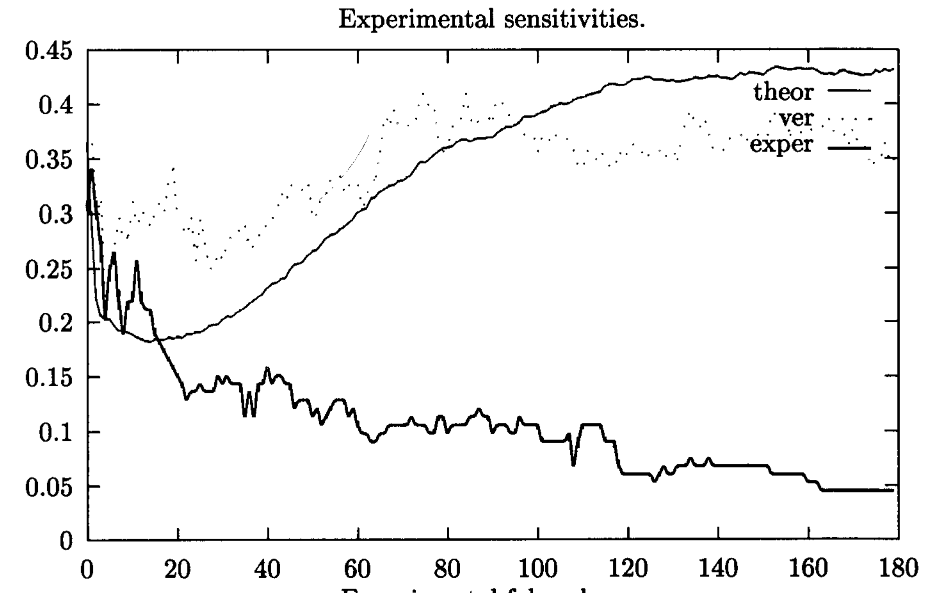


Figure 9: Probability distributions for the sequency walsh component. calcium-binding class versus random class.



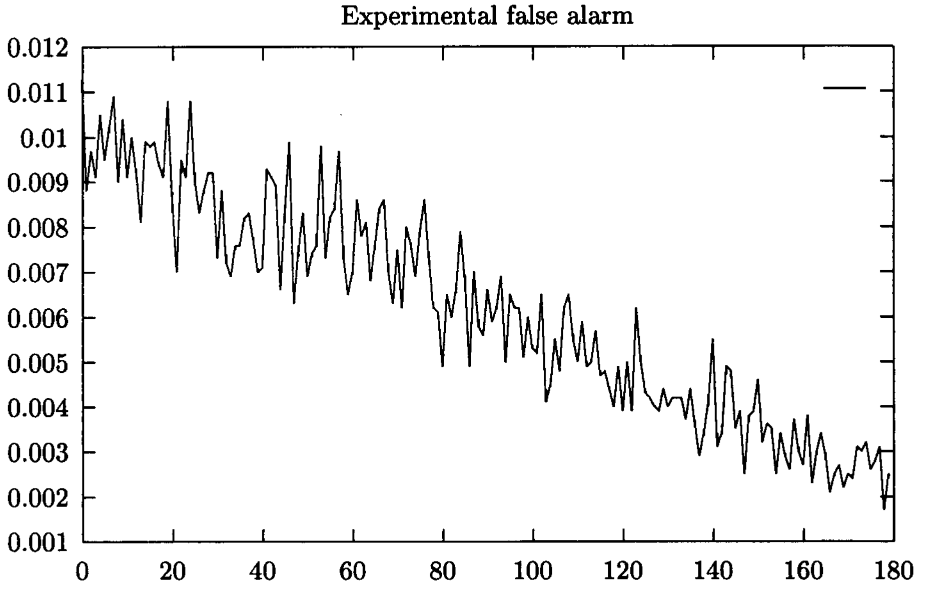


Figure 10: Theoretical, verified and experimental sensitivities against sequency on the upper plot. False alarm probability against sequency on the lower plot. Both, for classification of calcium-binding class against random class.

The sensitivity and false alarm curves are very interesting for this class. A one-component classifier (using sequency 1) gives:

- Theoretical Sensitivity = 36.50%
- Verified Sensitivity = 35.60%
- Experimental Sensitivity = 30.30%
- Experimental False Alarm = 1.13%

This is appearently about as good a performance as we can hope to have with hydrophobicity profiles, and walsh feature-space, on the calcium binding class. Adding more components initially decreases the theoretical sensitivity because these components are not good. At about 20 components there is a turning point, and it would appear that we will get a better performance with a multi-component test given enough components. Well, this is not true! The experimental sensitivity is monotonically decreasing throughout the plot. This discrepancy has to do with artifacts that are being introduced to our sequence class because of the random padding. The extra information that a multi-component classifier appears to be picking up does not come from the real part of the sequence as much as it does from the padding. So, while a multi-component classifier is very sensitive to the sequences that trained it, when those sequences are padded differently it becomes much less sensitive to them. We can probably explain this drastic behaviour if we look at the lengths of the actual calcium-binding proteins. It turns out that the sequences with length between 128 and 256 that are included in our sample, have actual lengths that are around 140. So padding accounts for pretty much half the sequence!! And it may very well be that the reason why sequency 1 is so sensitive to this class is because the first half of the sequence looks different from the second half and not because it is a calcium-binding sequence.

Since we are not interested in multi-component classifiers, there is not much to tell about the experimental false alarm probability. The plot indicates that the experimental fap is decreasing as we add in more components. We think that this happens because small errors in the threshold estimation magnify themselves when long products of probabilities become longer. Notice also, that in the case of interest, that is one-component classifiers, the experimental fap was 1.13% and that's good enough. Since we are not interested in multi-component classifiers for this sequence we will not worry about improving this situation.

• **Hemoglobin sequence class**

Our hemoglobin sequence class contains 485 samples with lengths between 128 and 256. Just like the calcium-binding sequences, most of the hemoglobin sequences have length between 140 and 160. However, the behaviour we observe is very different. Figure 11 shows which walsh components are most sensitive. Figures 12 show the sensitivity and false alarm curves. The false alarm curves do the same thing as in calcium-binding. On the other hand, notice that up to a point all three of our sensitivity curves increase. The experimental sensitivity is the first curve that stops increasing at about 30 to 40 components. This is the point when artifact information from the padding is being introduced to our classification algorithm. A good classification algorithm can be obtained for approximately 10 or so components with sensitivity over 50%.

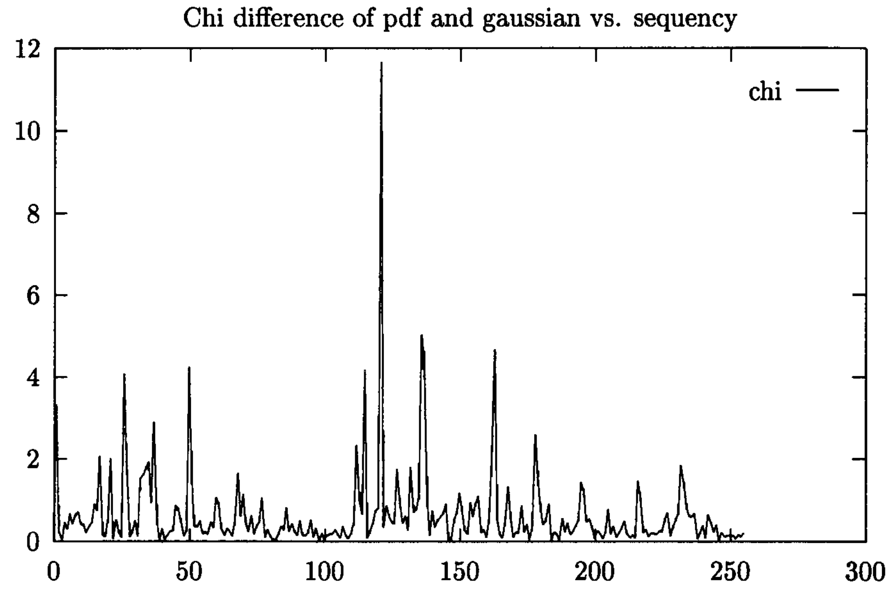
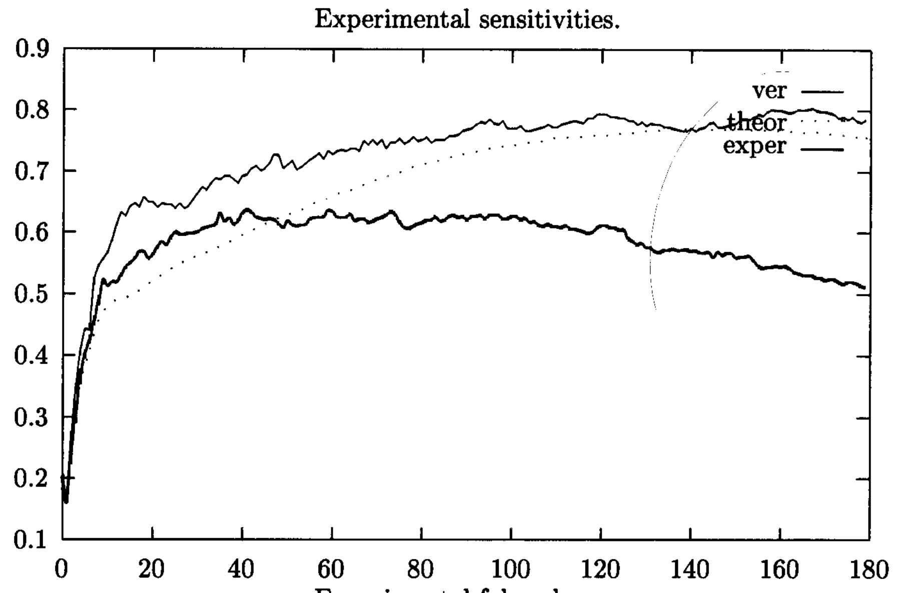


Figure 11: $\chi^2$ vs sequency comparison of hemoglobin's walsh pdfs versus random class walsh pdfs. The behaviour we see here is different from the behaviour we saw with calcium-binding sequences.



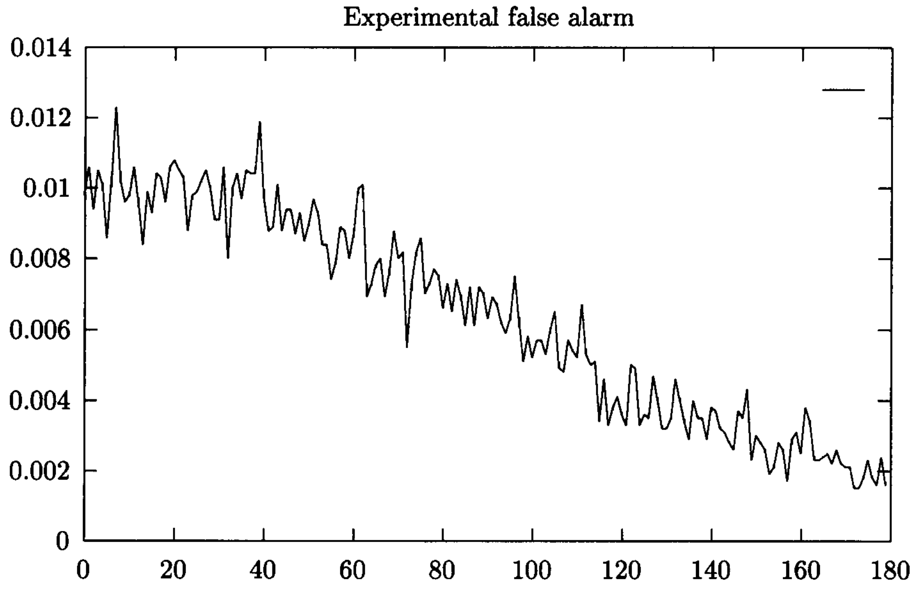


Figure 12: Sensitivity and false alarm vs. number of Walsh components. Hemoglobins perform much better than calcium-binding

## VII. Optimizations of Walsh-based classifiers

In this section we discuss our experience with various ideas that seem to be useful in optimizing Walsh-based classifiers. Two types of transformations that we explored were to change the property scale by a non-linear transformation (a *gamma correction*) and another is by applying our smoothing algorithm on the property profiles before taking the Walsh transform.

### • Gamma correction of the property scale

The idea behind the gamma-correction transformation is that it makes the most hydrophilic and hydrophobic aminoacids look even more so, while leaving the ones in the middle mostly unperturbed. Or it does the opposite thing, depending on the gamma itself. In terms of image processing, this is very similar to increasing/decreasing the contrast between various colors. Maybe for an appropriate configuration, the pattern that characterizes a sequence class can become more visible to Walsh analysis. The simple model to model this transormatio is to substitute the original hydrophobicity scale $h_i$ with

$$g_i = \begin{cases} h_i^\gamma & \text{if } h_i > 0 \\ -(-h_i)^\gamma & \text{if } h_i \leq 0 \end{cases}$$

Note that it is assumed that we have subtracted the average from our property scale, so that about half aminoacids have negative $h_i$ and the other half have positive. That way, this power transformation will have the appropriate stretching effect that we want. Our preliminary investigation suggests that transformations like this indeed affect the sensitivity of the walsh components, sometimes in a positive way and sometimes in a negative way. In this section, we will briefly outline our findings.

The first problem is quantifying the effect $\gamma$-correction has on the sensitivity of the walsh components. Calculating the sensitivities themselves over a fixed false alarm, is a very expensive computation. The second-best thing to do, is to use a $\chi^2(\lambda)$ comparison of the sequence class walsh pdfs versus the random class walsh pdf. For reminders, if sequency $\lambda$ component is distributed according to $p_\lambda(x)$, then the equations for calculating $\chi^2(\lambda)$ are:

$$\chi^2(\lambda) = \frac{1}{N-1} \sum_{n=1}^{N} \left\{ \frac{(p_\lambda(x_n) - g_\sigma(x_n))^2}{g_\sigma(x_n)} \right\}$$

$$g_\sigma(x_n) = \frac{e^{-x_n^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

where, it is assumed, that these distribution functions have been sampled over the points $\{x_1, x_2, \ldots, x_n\}$. Say that our property scale $\{h_1, h_2, \ldots, h_m\}$ is zero-mean then the sigma $\sigma$ of the random class walsh pdf is going to be:

$$\sigma^2(\gamma) = \frac{1}{2N(m^2-1)} \sum_{i=1}^{m} \sum_{j=1}^{m} (h_i^\gamma - h_j^\gamma)^2$$

For a fixed $\lambda$ then, $\chi^2$ will vary as we vary $\gamma$. The $\gamma$ at which $\chi^2$ is maximum, is the optimal gamma-correction that makes the specific sequency component most sensitive. That optimal gamma may not be the same for all sequency components, and as a matter of fact it turns out that it is not. In general, it seems that if we use the same $\gamma$ for all sequency components, there is little flexibility in optimising a classifier that uses many components, since each of those components has a different optimal $\gamma$. However, in the special case of the calcium binding sequences, we can accomplish a lot of optimisation, since the best classifier is a *one component* classifier! For a false alarm 1% we have raised the sensitivity of our optimum *calcium-binding vs random* classifier from 35% to around 50% !. In Figure 13 we show that the optimal gamma for the sequency 1 component is $\gamma = 2$. We also show the distribution of the sequency 1 component for the calcium binding class and the random class using that optimal gamma.

Setting the thresholds for 1% false alarm with these distributions, will give a sensitivity of almost 50%. This is quite an improvement over the performance we had for $\gamma = 1$ (sensitivity 35%). While this spectacular improvement could be an exception to the general rule, it suggests that the property that is being used in the property profiles is a significant part in creating the optimal classifier, and needs to be investigated more thoroughly in the future.

- **Smoothing of property profiles**

The idea of smoothing property profiles before using them came from the fact that smoothed property profiles look more pretty than the raw ones. Of course, the appreciation of a profile's beauty is at the eyes of the beholder, so in order for the reader to appreciate it with us we give an example of a property profile before and after smoothing in Figure 14 Recall that our smoothing algorithm (introduced in section IV) takes a numerical sequence $f_i, 0 \leq i \leq N$ and replaces it with

$$g_i = \sum_{n=-w}^{w} \beta_n f_{i+n}, \quad \forall i \in \{0, \ldots, r\}$$

where $f_i$ is extended by

$$f_n = f_0, \quad \forall n < 0$$

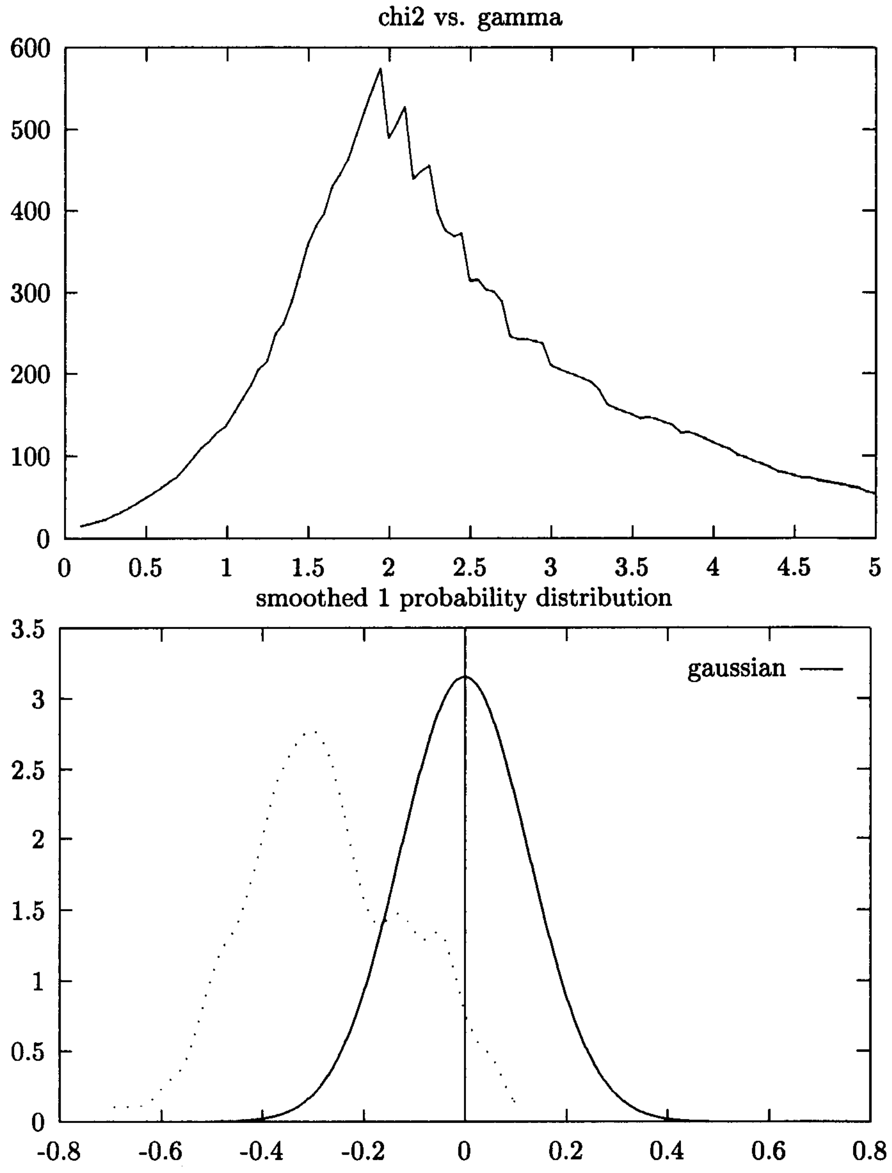


Figure 13:
**(up)** $\chi^2$ plotted against $\gamma$ for sequency 1, comparing the walsh component probability distributions of the calcium-binding class against the random sequence class.
**(down)** The probability distribution functions for the sequency 1 walsh components of the calcium-binding class and the random class. We use hydrophobicity property profiles with $\gamma$-correction $\gamma = 2$

$$f_n = f_N, \quad \forall n > N$$

The details about the $\beta_n$ coefficients are discussed in Appendix B. The filters we use are symmetric, i.e. they span from $-w$ to $w$. One can think of this as pre-processing the property profile by replacing every point in it, by a linear combination of the neighbouring points, and there is plenty of room for generalising this concept into taking a linear combination of *not* just neighbouring points.

Using smoothing means that we have to also apply smoothing to the profiles in our random class. Normally, we don't really generate a set of random sequences to *obtain* the pdf functions of the Walsh components over the random class. This is because those pdfs can be easily computed. The only place where we normally generate random sequences is to compute the experimental false alarm of a classifier. Now however that we apply a smoothing filter to the property profiles, the random sequences *also* have to be smoothed, and so our previews arguments don't work any more. To handle this then, we *do* generate a random sequence set of 1500 sequences and estimate the pdfs. Figure 15 shows how the sigma of the pdfs varies as a function of sequency for various smoothing windows. The mean of these gaussians of course is zero.

The big question is, how much does it help to smooth profiles? To preview classifier performance, rather than compute sensitivities, which takes time, we compared the sequence class pdfs $p(x)$ with the random class pdfs $r(x)$ with a $\chi^2$ test. Because in this case, both pdfs are computed from experimental data, our $\chi^2$ definition needs to be modified to:

$$\chi^2(\lambda) = \frac{1}{N-1} \sum_{n=1}^{N} \left\{ \frac{(p_\lambda(x_n) - r_\lambda(x_n))^2}{p_\lambda(x_n) + r_\lambda(x_n)} \right\}$$

Using this expression on the hemoglobin sequence class, in Figures 16 and 17 we see that $\chi^2$ improves for $w = 1, 2, 3$ and declines at $w = 4$. We also see that sensitivity migrates from the low-sequency components to the high-sequency components. Figure 18 showes how this reflects with sensitivity's improvement. The improvement is astounding! From a mere 50% we rise to sensitivities of the order 70%. *This is probably the most important result of our research so far.*

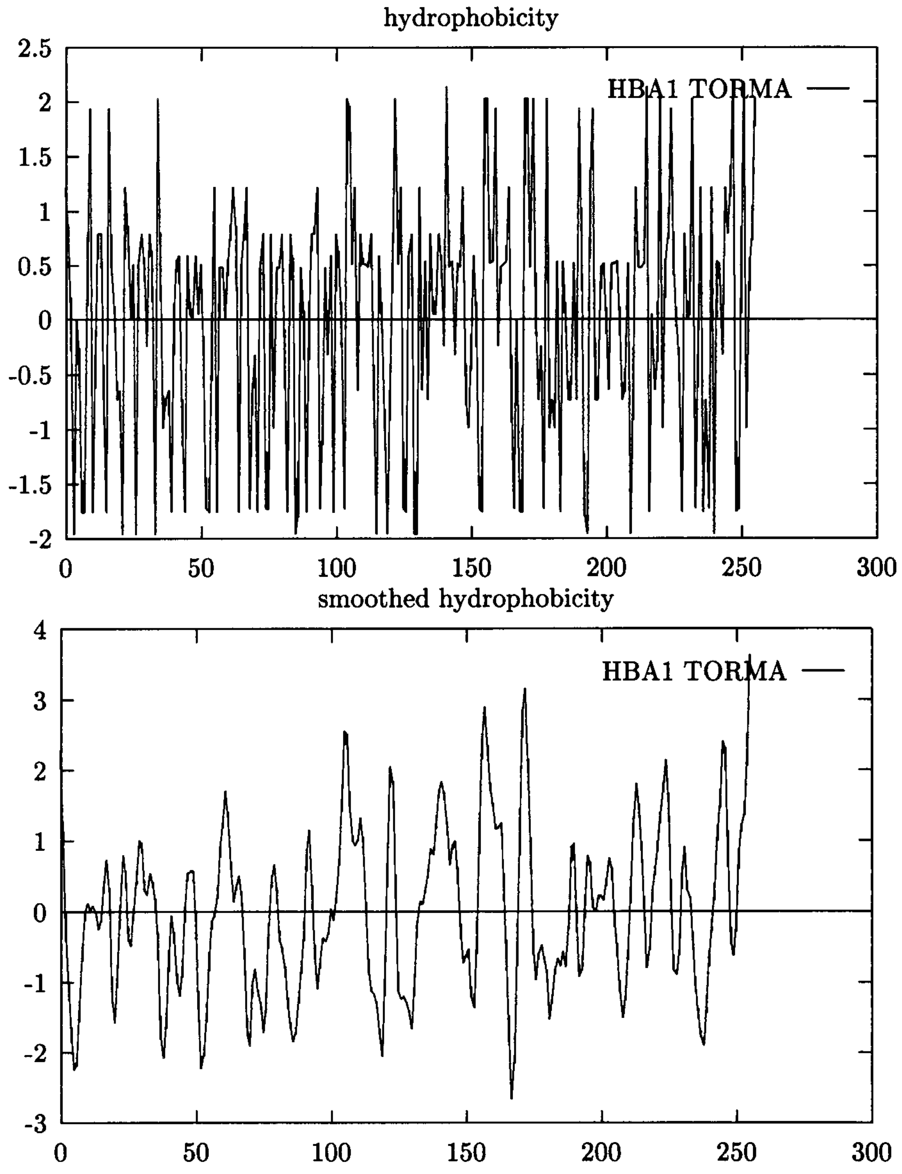


Figure 14: The property profile of a hemoglobin, before and after smoothing with window $w = 3$

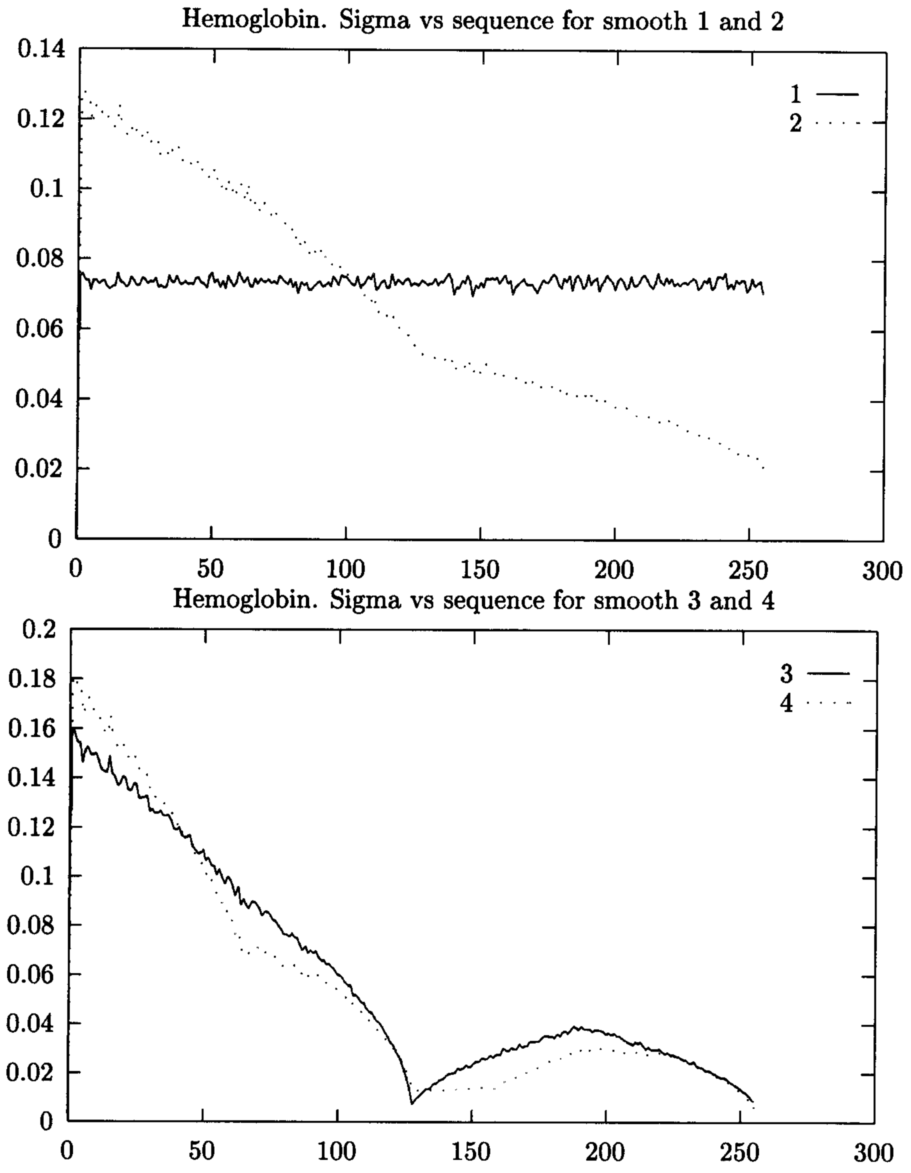


Figure 15: Sigma of pdfs to random sequence class vs. sequency for various smoothing windows. For $w = 1$, sigma is more or less constant, for higher $w$ it varies.

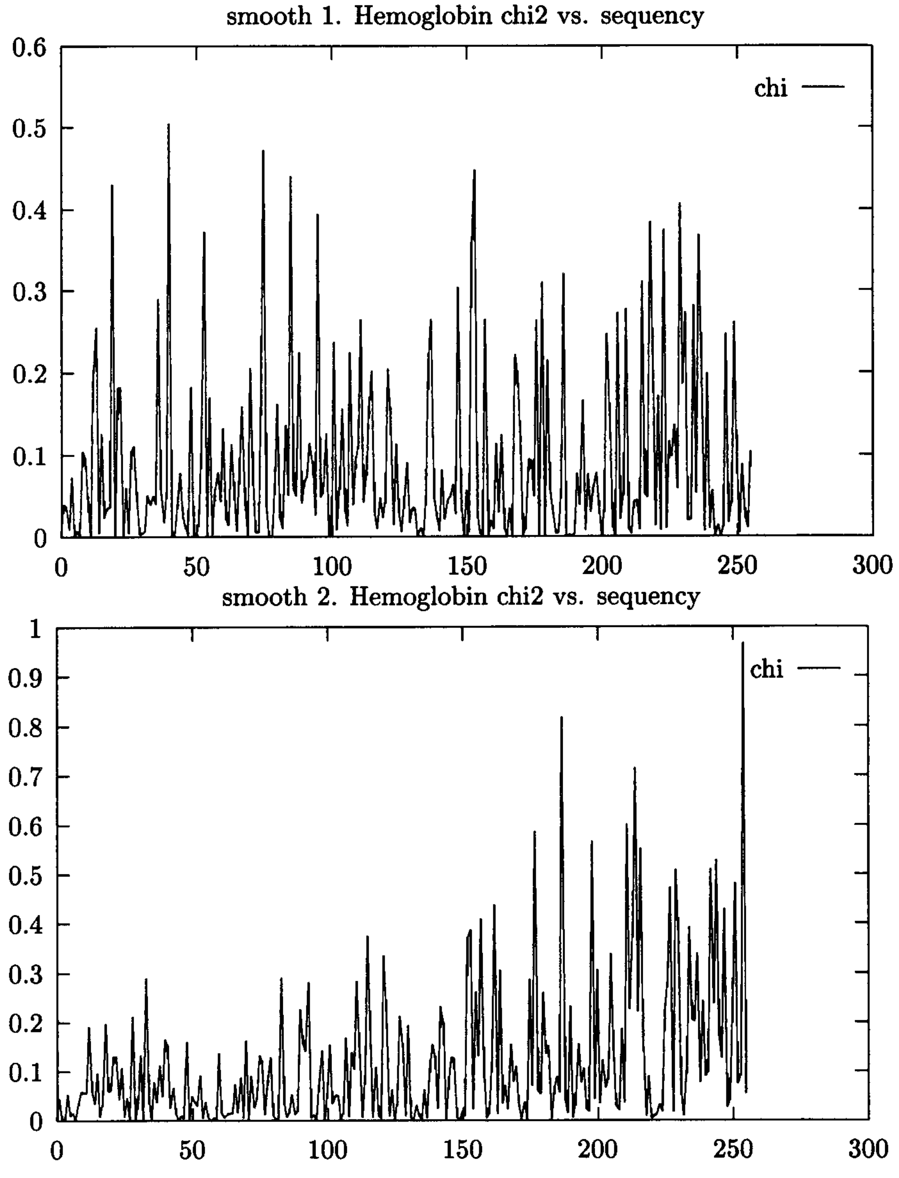
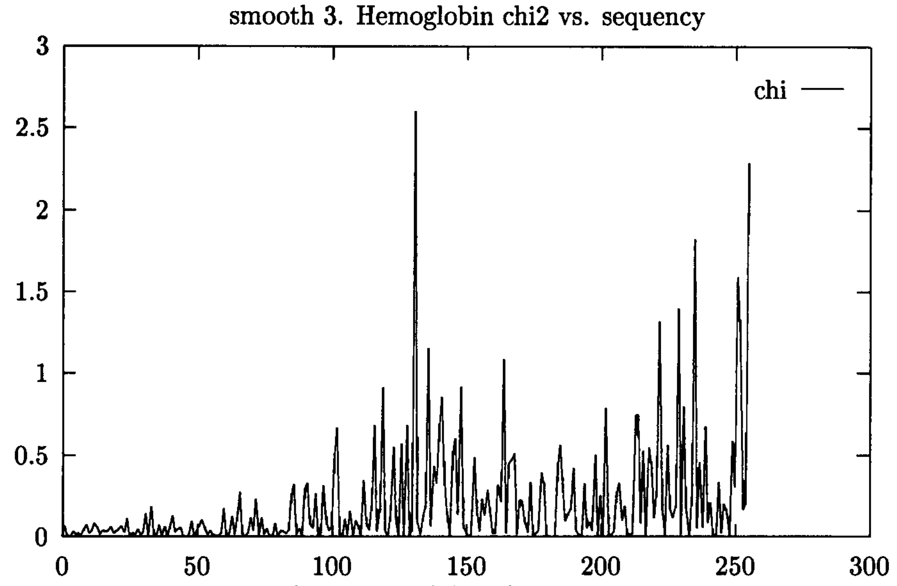


Figure 16: $\chi^2$ vs. sequency for $w = 1$ (up) and $w = 2$ (down). Notice the improvement.



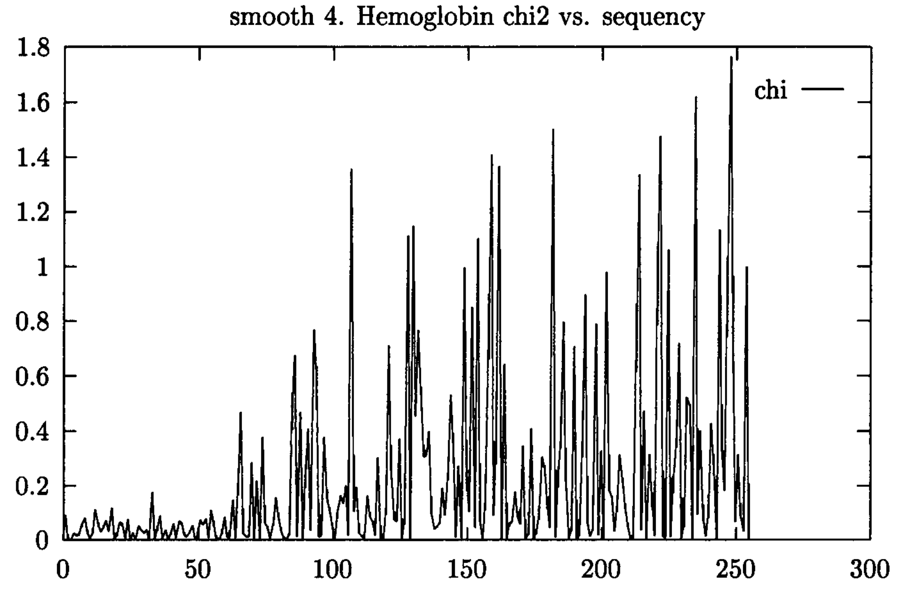
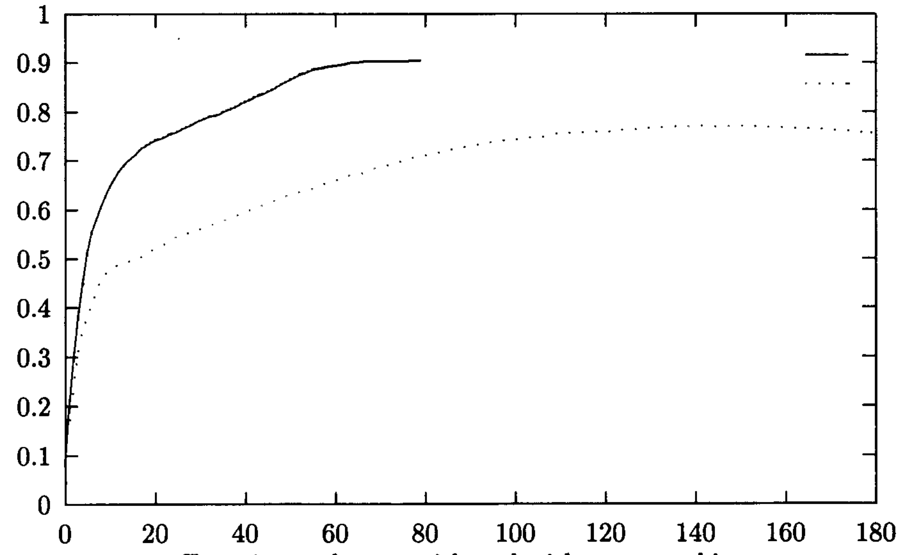


Figure 17: $\chi^2$ vs. sequency for $w = 3$ (up) and $w = 4$ (down). Notice the progressive improvement up to $w = 3$ and the decline when go to $w = 4$. The optimum smoothing window would be $w = 3$.

Theoretical Sensitivity with and without smoothing



Experimental sens. with and without smoothing

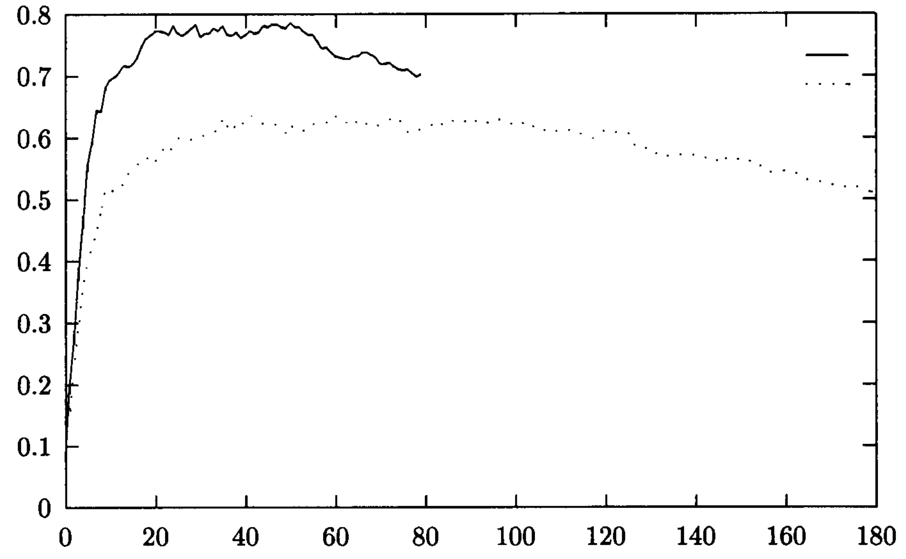


Figure 18: Comparing the sensitivities obtained for the hemoglobin vs. random sequence class, with and without applying smoothing to the property profiles before consideration. In the smoothing case, $w = 3$. The improvement is obvious and pretty astounding

## VIII. Conclusions and future directions

We have seen from our experiments that Walsh functions are capable of classifying a protein sequence class against the random class. It is also appearent that the performance of our classifiers is heavily dependent on the particular sequence classes that we work with. One problem with Walsh analysis is that we have to pad the sequence to a length that is a power of 2; we have seen the artifacts that this introduces in action. Another problem is that good sequence group samples are hard to come by.

The objective of studies like this is two-fold. One, we are trying to create the best sensitive classifier we can get. And two, we want to learn what is the most appropriate stochastic process that models protein sequences in the real world. We believe that the real model will be a non-linear model. Before plunging into that area however, we want to see how far we can go with linear stochastic models. Our Walsh analysis is such a model.

Our work with Walsh transforms is by no means complete. We need to look more closely at various sequence classes. We also need to try different properties as well as transformations of the properties currently in use and explore various ways to pre-process property profiles before passing them on to walsh analysis.Finally an area that has been left unexplored so far is the degree of correlation of Walsh components with each other.

## Appendix A: Sensitivity fluctuation experiment

The purpose of this experiment was to study how much the sensitivity performance of Walsh classifier fluctuates, for different signal classes that have the same distortion. To do this, we studied 128-long sequence classes, and for every distortion, we performed 10 classification experiments. As always, for every classification experiment, we generate a training and a testing class, and evaluate the theoretical, verified and experimental sensitivities, and experimental false alarm probability, as described in the paper.

| t.sens | comps | v.sens | e.sens | fap |
|---|---|---|---|---|
| Distortion 0.1 | | | | |
| 0.998703 | 7 | 0.996000 | 0.997000 | 0.002500 |
| 0.998098 | 10 | 0.991000 | 0.994000 | 0.001500 |
| 0.998426 | 6 | 0.996000 | 0.987000 | 0.003500 |
| 0.998004 | 8 | 0.994000 | 0.989000 | 0.000000 |
| 0.999752 | 1 | 0.998000 | 0.998000 | 0.002500 |
| 0.998501 | 7 | 0.992000 | 0.991000 | 0.004000 |
| 0.997764 | 4 | 0.994000 | 0.985000 | 0.000000 |
| 0.998539 | 7 | 0.994000 | 0.990000 | 0.003500 |
| 0.998745 | 7 | 0.995000 | 0.995000 | 0.001000 |
| 0.998690 | 6 | 0.995000 | 0.988000 | 0.000500 |
| Distortion 0.2 | | | | |
| 0.995059 | 15 | 0.984000 | 0.971000 | 0.003500 |
| 0.994637 | 13 | 0.990000 | 0.970000 | 0.002500 |
| 0.994079 | 13 | 0.986000 | 0.977000 | 0.001000 |
| 0.995438 | 18 | 0.984000 | 0.973000 | 0.002000 |
| 0.995152 | 11 | 0.993000 | 0.974000 | 0.001500 |
| 0.992850 | 25 | 0.977000 | 0.970000 | 0.002500 |
| 0.994636 | 16 | 0.984000 | 0.971000 | 0.005000 |
| 0.994905 | 14 | 0.985000 | 0.969000 | 0.003500 |
| 0.994033 | 11 | 0.989000 | 0.978000 | 0.004500 |
| 0.993319 | 17 | 0.986000 | 0.965000 | 0.004000 |

| t.sens | comps | v.sens | e.sens | fap |
|---|---|---|---|---|
| Distortion 0.3 | | | | |
| 0.972840 | 36 | 0.959000 | 0.916000 | 0.001000 |
| 0.981105 | 40 | 0.959000 | 0.900000 | 0.002500 |
| 0.973985 | 42 | 0.963000 | 0.898000 | 0.001500 |
| 0.964136 | 45 | 0.939000 | 0.922000 | 0.003000 |
| 0.972254 | 40 | 0.944000 | 0.904000 | 0.001500 |
| 0.970196 | 21 | 0.960000 | 0.932000 | 0.006000 |
| 0.974703 | 19 | 0.973000 | 0.942000 | 0.004500 |
| 0.968539 | 54 | 0.940000 | 0.896000 | 0.000500 |
| 0.971032 | 56 | 0.936000 | 0.890000 | 0.000000 |
| 0.975275 | 47 | 0.942000 | 0.916000 | 0.004500 |
| Distortion 0.4 | | | | |
| 0.860917 | 87 | 0.852000 | 0.788000 | 0.011000 |
| 0.846008 | 45 | 0.854000 | 0.821000 | 0.017000 |
| 0.827012 | 27 | 0.862000 | 0.835000 | 0.011500 |
| 0.848464 | 86 | 0.861000 | 0.749000 | 0.006500 |
| 0.825961 | 22 | 0.869000 | 0.871000 | 0.014000 |
| 0.852066 | 37 | 0.876000 | 0.846000 | 0.011500 |
| 0.830438 | 36 | 0.864000 | 0.846000 | 0.017500 |
| 0.860251 | 60 | 0.859000 | 0.806000 | 0.019500 |
| 0.830102 | 41 | 0.842000 | 0.858000 | 0.012000 |
| 0.849885 | 35 | 0.884000 | 0.863000 | 0.015000 |
| Distortion 0.5 | | | | |
| 0.620263 | 38 | 0.713000 | 0.675000 | 0.021000 |
| 0.599496 | 26 | 0.704000 | 0.671000 | 0.017500 |
| 0.628119 | 48 | 0.717000 | 0.677000 | 0.025000 |
| 0.607812 | 36 | 0.708000 | 0.663000 | 0.026500 |
| 0.619034 | 37 | 0.722000 | 0.694000 | 0.019000 |
| 0.594962 | 51 | 0.701000 | 0.674000 | 0.023000 |
| 0.603358 | 36 | 0.705000 | 0.678000 | 0.019000 |
| 0.623925 | 15 | 0.715000 | 0.701000 | 0.020500 |
| 0.612976 | 38 | 0.721000 | 0.692000 | 0.027500 |
| 0.622151 | 25 | 0.716000 | 0.703000 | 0.017500 |

| t.sens | comps | v.sens | e.sens | fap |
|---|---|---|---|---|
| Distortion 0.6 | | | | |
| 0.404803 | 34 | 0.536000 | 0.500000 | 0.018500 |
| 0.409328 | 34 | 0.523000 | 0.464000 | 0.019000 |
| 0.401206 | 44 | 0.519000 | 0.473000 | 0.024000 |
| 0.395204 | 33 | 0.518000 | 0.499000 | 0.021500 |
| 0.387867 | 49 | 0.515000 | 0.495000 | 0.026000 |
| 0.419014 | 25 | 0.529000 | 0.511000 | 0.019500 |
| 0.382161 | 28 | 0.502000 | 0.455000 | 0.019000 |
| 0.384520 | 36 | 0.503000 | 0.451000 | 0.016000 |
| 0.381923 | 28 | 0.501000 | 0.526000 | 0.017500 |
| 0.367986 | 36 | 0.489000 | 0.473000 | 0.023500 |
| Distrotion 0.7 | | | | |
| 0.211101 | 36 | 0.313000 | 0.282000 | 0.025500 |
| 0.206170 | 50 | 0.334000 | 0.305000 | 0.020500 |
| 0.224614 | 16 | 0.349000 | 0.271000 | 0.014000 |
| 0.195378 | 72 | 0.339000 | 0.264000 | 0.029000 |
| 0.207579 | 42 | 0.303000 | 0.269000 | 0.025500 |
| 0.204259 | 23 | 0.298000 | 0.271000 | 0.021500 |
| 0.220667 | 67 | 0.334000 | 0.269000 | 0.028000 |
| 0.214467 | 28 | 0.301000 | 0.276000 | 0.021500 |
| 0.218944 | 36 | 0.318000 | 0.333000 | 0.016500 |
| 0.201208 | 45 | 0.308000 | 0.279000 | 0.026000 |
| Distortion 0.8 | | | | |
| 0.097473 | 20 | 0.133000 | 0.130000 | 0.023000 |
| 0.098670 | 30 | 0.155000 | 0.148000 | 0.020000 |
| 0.098513 | 30 | 0.146000 | 0.133000 | 0.019500 |
| 0.098953 | 34 | 0.153000 | 0.146000 | 0.020500 |
| 0.098425 | 43 | 0.154000 | 0.150000 | 0.024000 |
| 0.094408 | 65 | 0.179000 | 0.110000 | 0.030500 |
| 0.112666 | 13 | 0.159000 | 0.150000 | 0.017500 |
| 0.089716 | 60 | 0.160000 | 0.128000 | 0.026500 |
| 0.109362 | 32 | 0.181000 | 0.141000 | 0.023000 |
| 0.094696 | 54 | 0.172000 | 0.127000 | 0.027000 |

| t.sens | comps | v.sens | e.sens | fap |
|---|---|---|---|---|
| Distortion 0.9 | | | | |
| 0.046278 | 78 | 0.068000 | 0.054000 | 0.028000 |
| 0.047633 | 78 | 0.070000 | 0.064000 | 0.022500 |
| 0.044158 | 77 | 0.083000 | 0.054000 | 0.026000 |
| 0.045488 | 77 | 0.085000 | 0.044000 | 0.029000 |
| 0.042362 | 61 | 0.090000 | 0.059000 | 0.025000 |
| 0.046805 | 85 | 0.084000 | 0.057000 | 0.027500 |
| 0.049524 | 92 | 0.085000 | 0.076000 | 0.029500 |
| 0.045492 | 57 | 0.077000 | 0.060000 | 0.025000 |
| 0.048575 | 72 | 0.107000 | 0.057000 | 0.028500 |
| 0.048728 | 83 | 0.104000 | 0.054000 | 0.031500 |

## Appendix B: Smoothing filters

The simple minded way to implement a sequence filter, is to substitute every entry of the sequence with the average of it's neighbors. This is equivalent to convoluting with a box function. The disadvantage of this method though is that it biases the signal hidden beneath the noise. For instance, consider passing a gaussian distribution through that filter. Then, the points near the peak would be biased to decrease, since all their neighbohrs take smaller values on the average. So, a smoothed gaussian would have larger variance than the real signal. The Savitzky-Golay filtering preserves these features.

For every data point $f_i$ we consider a window of neighbohring points $f_{i-n_L}, \ldots, f_{i+n_R}$, fit an M order polynomial (typically quadratic or quartic) to those points and set the smoothed $g_i$ to the value of that polynomial at position $i$. All these least squares fits would be quite CPU intense, if we were to perform one, for every data point. Luckily, it can be shown that there exist coefficients $\beta_n$, $n \in \{-n_L, \ldots, n_R\}$ for which

$$g_i = \sum_{n=-n_L}^{n_R} \beta_n f_{i+n}, \quad \forall i \in \{0, \ldots, r\}$$

will automatically accomplish the process of polynomial least-squares fitting inside the moving window. Since $f_i$ is only defined within $\{0, \ldots, r\}$ we extend it by setting:

$$f_n = f_0, \quad \forall n < 0$$

$$f_n = f_r, \quad \forall n > r$$

In general, to fit a data set

$$C = \{(x_i, f_i) : i \in \{1, \ldots, N\}\}$$

to an expression

$$y(x) = \sum_{k=1}^{M} \alpha_k X_k(x)$$

where $X_k(x)$ are some basis functions, we find the vector $\mathbf{a} = (\alpha_1, \ldots, \alpha_M)$ for which the *chi-squared metric* is minimised:

$$\chi^2 = \sum_{n=1}^{N} [f_n - y(x_n)]^2 \qquad \text{minimised.}$$

It can be proved that the needed vector can be calculated like this: If we define the *design matrix* as

$$A_{ij} = X_j(x_i)$$

then the vector $\mathbf{a} = (\alpha_1, \ldots, \alpha_M)$ of the LSF coefficients is

$$\mathbf{a} = (\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot \mathbf{A}^T \cdot \mathbf{f}$$

with $\mathbf{f} = (f_1, \ldots, f_N)$

To derive expressions for the $\beta_n$ smoothing coefficients, consider how we would obtain $g_0$. We want to fit a polynomial of degree M,

$$G_0(i) = \alpha_0 + \alpha_1 i + \ldots + \alpha_M i^M$$

to the set of data points $\{(-n_L, f_{-n_L}), \ldots, (n_R, f_{n_R})\}$ Then $g_0$ will be the value of that polynomial at $i = 0$, that is $\alpha_0$.

The design matrix is

$$A_{ij} = i^j \quad \forall i \in \{-n_L, \ldots, n_R\},\ \forall j \in \{0, \ldots, M\}$$

and if $\mathbf{f} = (f_{-n_L}, \ldots, f_{n_R})$ then

$$g_0 = \alpha_0 = \sum_{n=-n_L}^{n_R} \{(\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot \mathbf{A}^T\}_{0n} f_n$$

$(\mathbf{A}^T \cdot \mathbf{A})^{-1}$ is the inverse of

$$\{(\mathbf{A}^T \cdot \mathbf{A})\}_{ij} = \sum_{k=-n_L}^{n_R} \mathbf{A}_{ki}\mathbf{A}_{kj} = \sum_{k=-n_L}^{n_R} k^{i+j}$$

and you can see that the wanted magic coefficients are

$$\beta_n = \{(\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot \mathbf{A}^T\}_{0n} = \sum_{m=0}^{M} \{(\mathbf{A}^T \cdot \mathbf{A})^{-1}\}_{0m} A^T_{mn} = \sum_{m=0}^{M} \{(\mathbf{A}^T \cdot \mathbf{A})^{-1}\}_{0m} n^m$$

It's easy to see that this result is independent from the fact that we specifically calculated $g_0$.

In our implementation, we use *quadratic* smoothing filters, with $n_R = n_L = w$ and we refer to $w$ as the smoothing filter window. A filter with window $w$ involves the linear combination of $2w + 1$ terms. We began using smoothing filters as part of the pdf estimation algorithms. However, we found that preprocessing property profiles with a smoothing filter can have positive effects on solving the classification problem.

## References

1. W.H. Press, *et al.*, "Numerical Recipies in C", (Cambridge University Press,1992).
2. A.D. Whalen, "Detection of Signals in Noise", (Academic Press,N.Y., 1971).
3. K.G. Beauchamp,"Transforms for Engineers",(Clarendon Press, Oxford, 1987).
4. D.S Stoffer,"Walsh-Fourier Analysis and its Statistical Applications", J. Am. Stat. Asso., 86:461-479,1991.
5. D.S Stoffer,"Walsh-Fourier Analysis of Discrete-Valued Time Series", J. Time Series Anal., 8:449-467,1987
6. P.K. Ponnuswamy and M. Gromiha, Int. J. Peptide Protein Res., 42:326-341, 1993
7. D. Eisenberg, "Three-Dimensional Structure of Membrane and Surface Proteins", Ann. Rev. Biochem., 53:595-623, 1984
8. D.Eisenberg, R.M. Weiss, and T.C. Terwilliger, "The Hydrophobic Moment Detects Periodicity in Protein Hydrophobicity", Proc. Natl. Acad. Sci. USA, 81:140-144, 1984
9. R.M. Sweet and D.Eisenberg, "Correlation of Sequence Hydrophobicity Measures Similarity in Three-Dimensional Protein Structure", J. Mol. Biol., 171:479-488, 1983
10. K.A Dill, "dominant Forces in Protein Folding", Biochem.,29,7133-7155, 1990.