Walsh and Partition Functions Made Easy

Paul Field
Department of Computer Science
Queen Mary and Westfield College
Mile End Road
London E1 4NS
email: paulf@dcs.qmw.ac.uk

Abstract

This paper introduces Walsh and partition functions, coefficients and transforms in a genetic algorithm context. This has been done before, but this paper takes a light and basically non-mathematical approach to the subject. As well as describing the functions themselves, the paper deals with the idea of basis and what a "change of basis" means. The mathematics for all these concepts is presented, but the preceding nontechnical description should make its introduction relatively painless.

1 Introduction

"Two voices are there; one is of the sea,
one of the mountains; each a mighty voice"
— William Wordsworth

An important question for genetic algorithm research to answer is "What problems are hard for a genetic algorithm?". One of the major approaches to this has been the study of *deception*. What are deceptive functions? To use an old cliché, imagine that the genetic algorithm is a mountain climber trying to climb the highest mountain. It looks for signposts (sometimes called schemata) that tell it where this Everest is. Some signposts are very specific stating "You are at a very high spot", others are more general and say "The land to the North is generally higher than the land to the South". The genetic algorithm takes the advice of as many of these signposts as it can (there are an awful lot) and usually ends up at the highest point. With a deceptive function the signposts are still true, but they are very misleading. Reading "The land to the North is generally higher than the land to the South" will probably make the GA head off North. However, the word "generally" makes the signpost deceptive. There may be a lot of reasonably high ground to the North but to the South the optimum juts out from otherwise low ground.

We can use the idea of deception for two purposes. Firstly, to see how misleading an existing function is and secondly to create a benchmark function that misleads to a certain degree (i.e. has a certain number of misleading signposts). Notice that deceptive functions are not necessarily hard. The GA may head off in the wrong direction but then turn back and find the optimum. (see Goldberg's description of the minimal deceptive problem in Goldberg (1989c) and see Grefenstette (1993)).

This paper is not about deception as such. It is about Walsh and partition functions. These are the tools with which we can analyse functions for deception. Goldberg (1989a & 1989b) has already provided a good introduction to using Walsh functions for this. He also describes partition coefficients in Goldberg (1989c, Appendix E). All of these presentations are reasonably mathematical. Of course, Walsh and partition functions are mathematical ideas, but it is possible to explain many of the concepts involved without maths.

We are going to look at two views of Walsh and partition functions. For both we will begin with extremely simple examples. However, these examples contain everything necessary to understand the concepts and we will discuss their application to more complicated examples. At the end of each section we will translate the ideas into a mathematical form thus putting

the reader in a position to approach more serious papers on the topic.

During the next section we will spend a lot of time discussing partition functions and the reader might worry that Walsh functions have been forgotten. Fear not. There is only a tiny difference in the meaning of the source patterns for Walsh and partition waves. Everything else is identical. We leave the discussion of this difference until the end of the section.

Finally, we will be dealing with a GA that manipulates binary strings (as does much other GA research). However, don't be fooled into believing that GAs should only use binary strings. The "alphabet question" is still open and Walsh and partition coefficients have been generalised to nonbinary strings. See Mason (1991) for a "functional" description of nonbinary partition coefficients and nonbinary deception and see Field (1994) for the matrix forms of nonbinary partition and Walsh transforms (and their inverses), the relationship between them and the fast algorithm for calculating them.

2 Making Waves

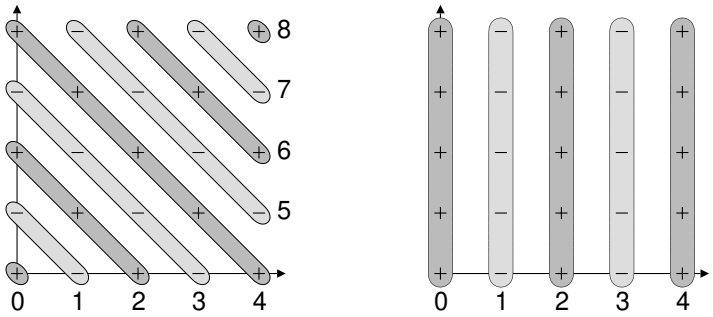

Figure 1: Two different waves viewed from above.

Partition functions are majestic waves imprisoned in a frozen sea, their mighty crests laughing at the tiny ships stranded atop them. OK, so actually they're more like not-so-majestic waves, too lazy to move from in front of the TV, unconcerned by the string fitnesses languishing on their crests like undersized toupees. Nevertheless, partition functions are like waves and we can use this analogy to understand what they do to the high-dimensional spaces that our feeble brains cannot imagine.

Thinking of a fitness function often evokes a vision of mountains. We see a peak here, a valley there, but the view is quite local. We see how high or low individual points are. Instead we can see a function as a sea. The same range of high and low points can occur on the surface of the sea but the cause is different. Ripples summoned by the wind, the wake of a ship, the tug of the moon; each of these is a wave with a simple underlying patterns of ups and downs but, when combined, they form a complex surface. We must remember that real seas and waves are dynamic. A fitness function is a frozen sea, turned to ice in a split second or captured as a snapshot. Thus the waves are frozen into unmoving patterns too.

If we are going to view functions as made up from waves, we had better decide what we mean by a wave. A wave is a regular pattern of ups and downs on the surface of the water. More specifically, the wave is created by one or more points being pulled up[1]. These points are called the *source* of the wave. The wave's effect on any point can be determined by that point's distance from the source. Points at odd distances are pulled down and points at even distances are pulled up.

Figure 1 shows some examples. In these diagrams we're looking from above so crests (points pulled up) are shown as "+" and troughs (points pulled down) as "–". The left hand wave's source is the bottom left point. As you can see, the wave pulls points at a distance of 1 down, points at a distance of 2 up, points at a distance of 3 down and so on. The same rule creates the right hand wave of figure 1 but this time the source is all the points at the left hand edge.

[1] How do you "pull up" water? Easy. Either use your gravitational pull or, if you do not weigh several million tons, simply tie some string around the appropriate water molecules.

Notice that we aren't measuring distance "as the crow flies". Instead, we add together the number of units across and the number of units up.

Let's look at how this fits in with genetic algorithms and their strings. Imagine we have a problem whose solutions are represented by a 2 bit string. Each string is like a coordinate; the first bit tells us how far North we should travel and the second bit how far East. The coordinates tell us the positions of four buoys floating on the sea. The heights of these buoys correspond to the fitnesses of the GA's strings.

Partition functions are the waves which form a fitness function. In the 2 bit case there are four partition functions. They are shown in figure 2. Wave $E_{\#\#}$ is a swell. Its source is all four points. Wave $E_{\#1}$'s source is 01, 11 and the distance 1 points (10, 00) are pulled down. Wave $E_{1\#}$'s source is 10, 11 and the distance 1 points (00, 01) are pulled down. Finally, wave E_{11}'s source is 11, the distance 1 points (01, 10) are pulled down and the distance 2 point (00) is pulled up. Fortunately for us, these four waves are sufficient to produce any 2 bit fitness function[2].

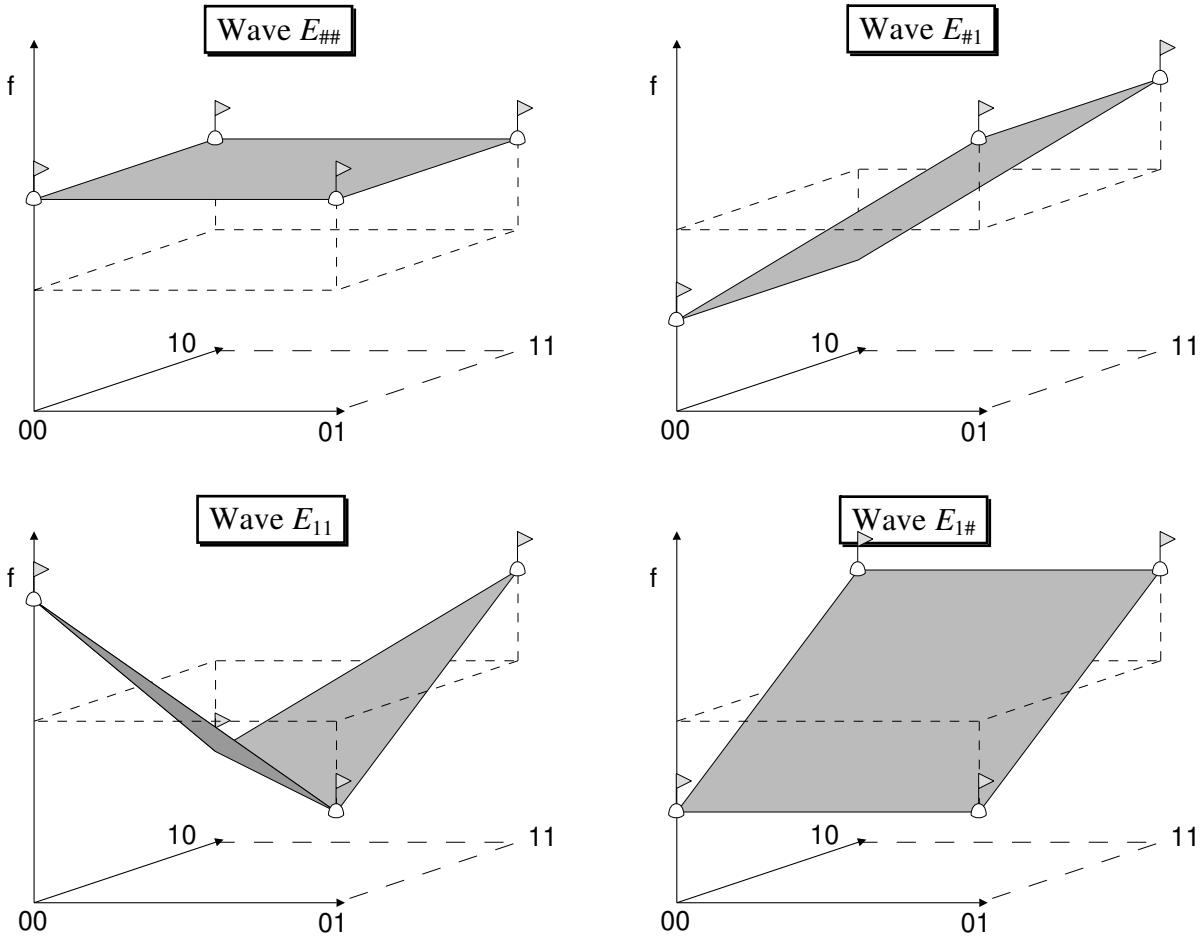

Figure 2: Time for the seasickness pills

Why are the partition "waves" named so strangely and how do we know what their sources are? One question answers the other. A partition waves's symbol has a *source pattern* as a subscript. This pattern is made from 1s and #s and describes the points in the source. A 1 in the pattern will only match a 1 in a string, whereas a # matches both 1 and 0. For example, the pattern 1# matches 10 and 11, ## matches all the strings and 11 only matches 11.

To work out whether a point is pulled up or down we need to know how far it is from the source. Remember we are measuring distance as "amount across plus amount up" which, for

[2] We won't be discussing Walsh functions until the end but it's worth noting that if you invert $E_{\#1}$ and $E_{1\#}$ then the four waves will be Walsh functions rather than partition functions.

strings, translates into "change in bit 1 plus change in bit 2". We ignore bits where the source pattern has a #. So 10 is one away from #1 and no distance from 1#. 00 is two away from 11. We'll see more detailed examples in the next section.

So far we have discussed points being pulled up and down. But how far up and down are they pulled? Each wave needs to have a strength. $\varepsilon_{\#\#}$ is the strength of $E_{\#\#}$, $\varepsilon_{1\#}$ is the strength of $E_{1\#}$ and so on. If a wave's strength is 2, then points are either pulled up by 2 or down by 2. If the strength is 0.5 then points are either pulled up 0.5 or down 0.5. Finally, if the strength is negative, for example –2, then the wave is inverted. Points that should go up actually go down by 2 and points that should go down actually go up by 2.

There has been far too little jargon so far. You have already seen waves (the *E*s) referred to as *partition functions*. In addition, their strengths (the εs) are called *partition coefficients*. We will intermix the terms fairly freely but will move to the technical terms as we get more mathematical.

So to summarise. *We can think of a 2 bit fitness function as being made up of four partition functions (waves) of various strengths.* The waves are the same whatever the function so to describe a function we just need the four partition coefficients (wave strengths). We have two equally valid ways of viewing the function: four string fitnesses or four partition coefficients.

Before we go on there is one question that the astute reader might be worried about. How do we know that we can make *any* 2 bit fitness function from these waves? We don't. Well, not using the waves analogy anyway. Just trust me for the moment and in section 3 we'll see a different way of looking at partition functions that shows that any fitness function can be described using them.

2.1 A GA is like a harpist. If it only plays on four strings it is considered uninteresting.

I said that waves would help us to explain how partition coefficients describe high-dimensional fitness functions. You may be thinking that we're not gaining anything; we'll still have to imagine waves in 20 dimensional space. Not at all. You already know everything about 20 dimensional waves from the 2 dimensional case. Waves have a strength. Waves are formed from a source. The distance of a point from the source determines whether it is pushed up or down. All we need to do is specify sources and distances in 20 dimensions. Easy eh? Ah. Well, before anyone feels betrayed and complains that their local stationer does not stock 20 dimensional rulers, I will point out that these two things are very easy. Honest.

In the last section we had a two bit problem and we imagined waves on the sea (a two dimensional surface). When I talk about a 20 dimensional solution space I am merely using confusing language to try and sound intelligent. All I mean is "all the solutions, as represented by strings containing 20 bits". So, bits equals dimensions.

How do we specify, say, a 10 dimensional partition wave? Just make a 10 symbol long source pattern. Remember the patterns are made from 1s and #s. For example, the source pattern 1#1##1#111 generates the wave $E_{1\#1\#\#1\#11}$.

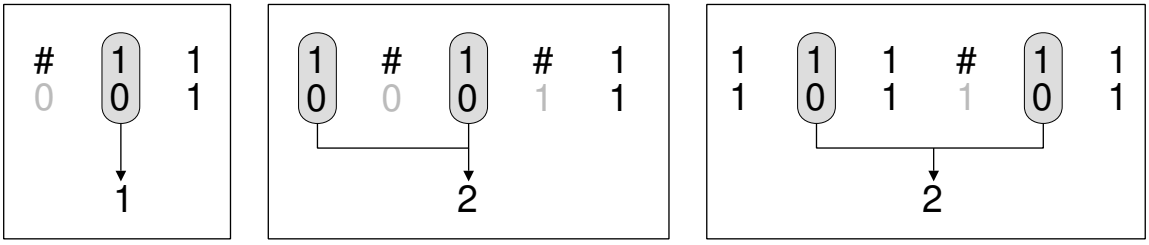

Figure 3: The distance of points from wave sources

What about the distance of a point from the source? Simply count the number of bit positions at which the source pattern and the string differ (ignoring #s). Figure 3 shows three

examples.

That's all there is to it but we'll look at an example we can sort-of visualise just to see the ideas in action.

As in the 2 bit case, a 3 bit string is like a coordinate. The first bit is the distance up, the second the distance back and the third the distance across. Notice that moving along an edge is moving a distance of 1 unit (and thus moves to a string with a single bit difference). This gives us a cube, but leaves us with no dimension for fitness. We could use blobs of colour on the corners of the cube to represent fitness but, for easy comparison with figure 2, we will cheat and use the "up" dimension to stand for fitness *as well as* the first bit.

In the 3 bit case there are 8 source patterns that can be made and so eight partition functions: $E_{\#\#\#}$, $E_{\#\#1}$, $E_{\#1\#}$, $E_{\#11}$, $E_{1\#\#}$, $E_{1\#1}$, $E_{11\#}$, E_{111}. We will look at E_{111} and $E_{\#11}$ as examples. The following table shows the distance of points from the sources and thus the effect the wave has on a point. Figure 4 shows the waves graphically.

Point	000	001	010	011	100	101	110	111
Distance from #11	2	1	1	0	2	1	1	0
Effect of $E_{\#11}$	Up	Down	Down	Up	Up	Down	Down	Up
Distance from 111	3	2	2	1	2	1	1	0
Effect of E_{111}	Down	Up	Up	Down	Up	Down	Down	Up

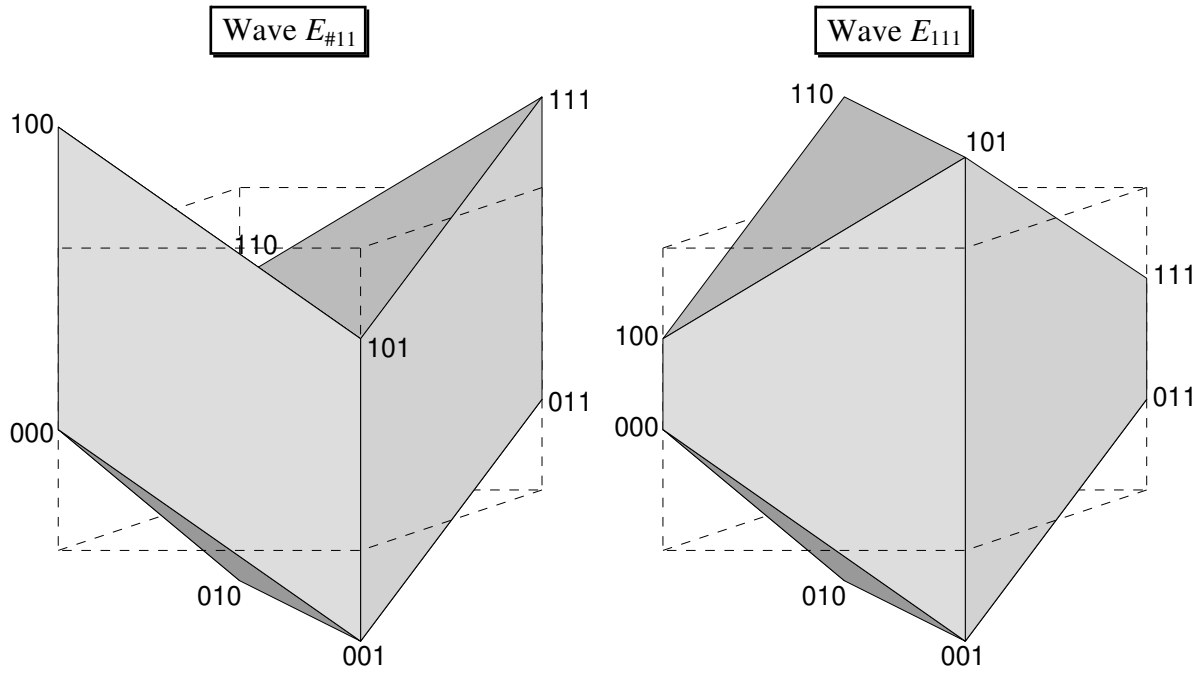

Figure 4: How $E_{\#11}$ and E_{111} affect a 3 bit "fitness cube"

2.2 This is far too simple. Make it look hard so I can impress my friends.

It's time to write all this mathematically. First we need some symbols. We'll use s for a string and p for a source pattern. l is the length of the string (and pattern). We already have partition functions (waves) E_p and partition coefficients (wave strengths) ε_p. We need to write the equation of a partition function and the fitness function the partition coefficients describe.

When passed a string, a partition function returns +1 if the string's fitness should increase, −1 if it should decrease. To work this out we find the distance of the string from the source and return +1 if it is even and −1 if it is odd. We know how to find the distance; we have to

compare the source pattern p and the string s a position at a time and count the differences. However, we can be crafty. We don't need the distance, only whether it is odd or even. We start with it being even (distance is 0) and then toggle it whenever we spot a difference between the pattern and the string. Remember our function should return +1 for even and −1 for odd, so we'll use this all the way through. We can toggle this state by multiplying by −1 (1 (even) × −1 = −1 (odd) and vice versa) and we can leave the state unchanged if we multiply by 1. We can use this to write a partition function as:

$$E_p(s) = \prod_{i=1}^{l} d(p_i, s_i)$$

The function d compares the string and pattern at one position (i) and returns a 1 (i.e. don't change the state) if they match and −1 (i.e. toggle the state) if they don't. All the Π does is multiply together the values returned by d for every position along the pattern and string.

Now we need to define d. Basically, the string and pattern match at a particular position if they are the same (at that position) or there is a # in the pattern (at that position). Remembering that we return 1 for a match and −1 otherwise we can write:

$$d(p_i, s_i) = \begin{cases} 1 & \text{if } p_i = s_i \text{ or } p_i = \# \\ -1 & \text{otherwise} \end{cases}$$

Let's see an example of this for the wave $E_{\#\#11}$:

$$\begin{aligned} E_{\#\#11}(0101) &= d(\#,0) \times d(\#,1) \times d(1,0) \times d(1,1) \\ &= 1 \times 1 \times -1 \times 1 \\ &= -1 \end{aligned}$$

Finally, writing the set of all source patterns as P, we can add all the partition functions (times their strengths, the partition coefficients) together to form a fitness function:

$$f(s) = \sum_{p \in P} \varepsilon_p E_p(s)$$

Or in English: To find the fitness of a string add up the effects of all the waves on that string.

2.3 I've paid good money for Walsh functions and that's what I want

Walsh functions are the same as partition functions except that their labels mean slightly different things. Both of the functions' labels specify the wave's source but the bits in a Walsh function's label match the *opposite* bits in the string. For example, the partition function $E_{1\#\#}$ has 100, 101, 110, 111 as its source. The same label on a Walsh function, i.e. $\psi_{1\#\#}$, means 000, 001, 010, 011 is its source. As another example, $\psi_{11\#}$ has 000, 001 as its source. This is rather counter-intuitive which is why we concentrated on partition functions. That's the only difference so let's see how it affects the equations.

The function d matches a string and a pattern at a particular position so this is the only thing that has to change. The change is small. We match if the bit in the pattern *doesn't equal* the bit in the string (rather than if it does):

$$d'(p_i, s_i) = \begin{cases} 1 & \text{if } p_i \neq s_i \text{ or } p_i = \# \\ -1 & \text{otherwise} \end{cases}$$

Before a hundred researchers accost me for selling them defective merchandise, Walsh functions and coefficients are usually labelled with a number rather than a pattern. This is just a different notation. Convert the number to binary and replace the 0s with #s and you have the pattern. So, the Walsh coefficient w_5 is $w_{1\#1}$ in our notation.

The Walsh function equations are written in a variety of ways although the majority of them boil down to multiplying 1s and −1s for each bit position in the string. Usually there is some sort of "gimmickry" involved in generating the 1s and −1s whereas the description in this paper makes the bit comparison operation explicit. On the other hand these formulations are very compact. Let's look at one of the "cleanest" (in my opinion) from Bridges and Goldberg (1991). They define Walsh functions as:

$$\psi_j(s) = \prod_{i=1}^{l} (-1)^{j_i s_i}$$

This takes advantage of j being a binary number rather than a pattern. If we enumerate d' and $(-1)^{j_i s_i}$ we can see that they are identical:

j_i (p_i)	s_i	$d'(p_i, s_i)$	$(-1)^{j_i s_i}$
0 (#)	0	1	1
0 (#)	1	1	1
1	0	1	1
1	1	−1	−1

3 A change of basis?

Partition coefficients are one of many ways of looking at a function. Are there other useful perspectives on functions? We need formalise the idea of a "way of looking at something" to find this out and also to see how we can change the way we look at function (i.e. from string fitnesses to partition coefficients and vice versa).

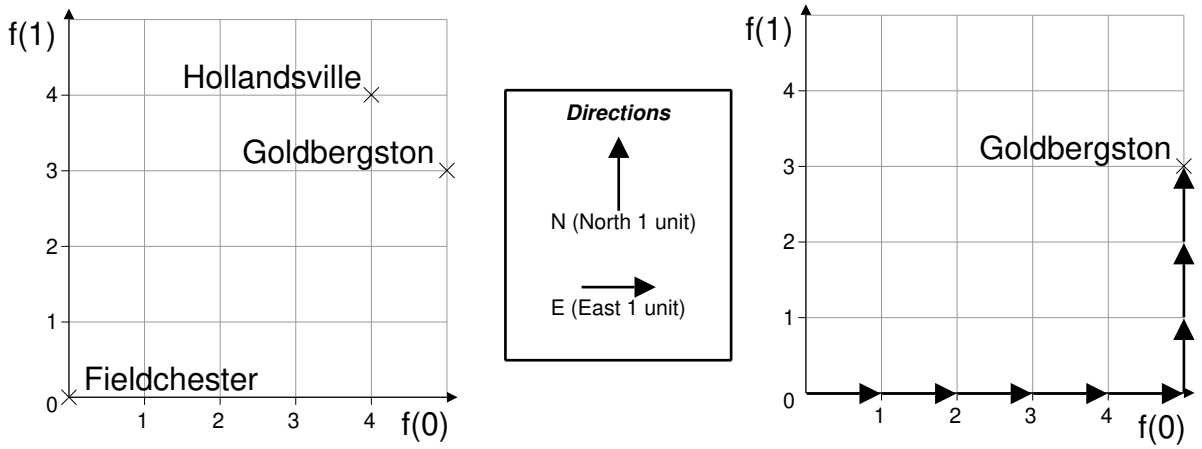

Figure 5: Car Journey to Goldbergston

Loosely speaking, a *basis* is a way of looking at things. A vague definition but we'll fill in the details as we go along. The "things" we are interested in looking at are functions. We'll start by imagining a "map" of functions telling us how to get from place to place (i.e. function to function). Figure 5 has the map of 1 bit functions on its left. There are an infinite number of functions on this map but a few places of interest have been specially marked. If we want to travel between points we need a way to describe our journey so we use the directions N and E. All our journeys will be from the immodestly named Fieldchester (the origin). So, to travel to Goldbergston we travel 5E and 3N. Notice that all directions have a length associated with them, so we don't have to say "4·5 kilometres East", we just say "4·5E". We have two ways to specify where Goldbergston is. Using map coordinates it is at (5,3) and using the journey from Fieldchester it is at 5E + 3N. There appears to be little

difference between the two but we'll soon see an example where there the distinction is clearer.

"Why don't we have S and W as well as N and E?" you cry. We don't need them. S is simply −N and W is −E. What about NE or SE? Again, they are redundant. NE is N+E, SE is E−N (or −N+E, the order of directions is unimportant). There are two important things to notice about the directions N and E. We can get to any point on the map using them (in jargon we say the directions *span the space* of functions) but they are all necessary; if we discard E then there is no way to travel East. We've already seen how other directions could be made from N and E, but N can't make E and E can't make N (in jargon we say the directions are *linearly independent*—if we used the directions N, E and SE they would not be linearly independent because SE can be made from N and E). If we have directions with these two properties they are called a *basis*.

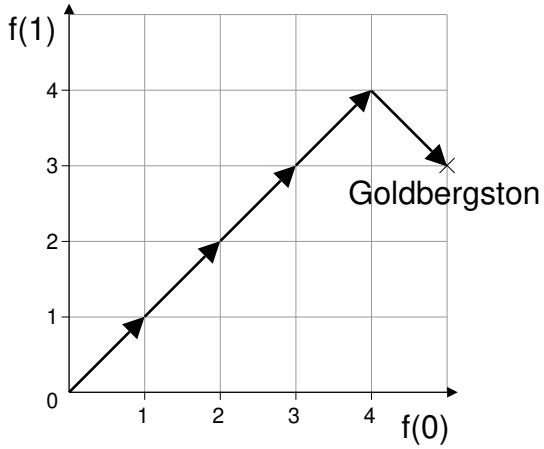

Figure 6: Boat Journey to Goldbergston

The N, E basis is the usual way of viewing functions. We "journey" to the function by specifying the fitness of each string i.e. the fitness of 1 (the number of Ns) and the fitness of 0 (the number of Es). Section 2 has already shown us that partition coefficients are an equally valid way of describing a function. So how do they fit in? Rather simply. The partition functions ($E_{\#}$ and E_1 for a 1 bit function) are the directions NE and SE respectively. The partition coefficients form the journey to the function i.e. how many NEs and SEs to travel. And what about Walsh functions? $\psi_{\#}$ and ψ_1 are NE and NW respectively.

Figure 6 shows the journey to Goldbergston this time travelling on the partition waves.

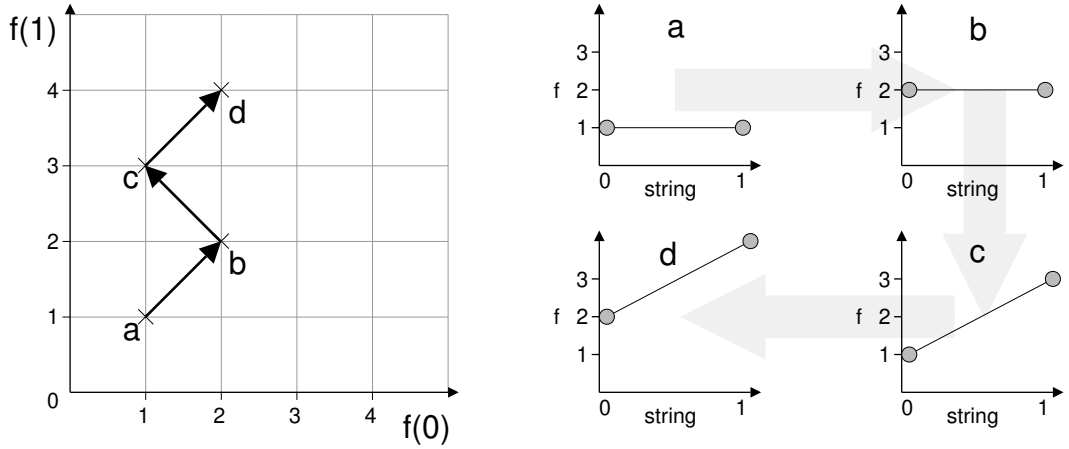

Figure 7: Four picture postcards from a journey

We can now specify Goldbergston as 4NE + SE if we want to. This is a different way of looking at the problem but what does it mean? Before, the number of Ns also happened to be the fitness of the string 1. What do NE and SE stand for?

Figure 7 shows the effects of moving NE and SE. Moving from a to b and from c to d (both NE) causes the average fitness of all points to increase by one. Moving from b to c causes a "twist" (compare with the wave $E_{\#1}$ in figure 2). The fitnesses move relative to each other but the average fitness stays the same. As we said, NE is $E_{\#}$ and SE is E_1.

It's time to formalise these ideas so that we can deal with larger than 1 bit functions and also find out how to change basis.

3.1 Sailing on the matrices

We can use matrix notation to formalise our ideas of bases. We will only have to use matrix multiplication and, at the end, matrix inversion.

We can write each direction as a vector. The number at the top is the effect the direction has on the *y* coordinate of a point and the lower number is the effect on the *x* coordinate. :

$$\text{N} = \begin{bmatrix}1\\0\end{bmatrix} \qquad \text{E} = \begin{bmatrix}0\\1\end{bmatrix} \qquad \text{NE} = \begin{bmatrix}1\\1\end{bmatrix} \qquad \text{NW} = \begin{bmatrix}1\\-1\end{bmatrix} \qquad \text{SE} = \begin{bmatrix}-1\\1\end{bmatrix}$$

We can place these vectors side-by-side to form a matrix for each basis:

$$\begin{matrix}\text{Fitness basis}\\(\text{N, E})\end{matrix} : \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \qquad \begin{matrix}\text{Partition basis}\\(\text{NE, SE})\end{matrix} : \begin{bmatrix}1 & -1\\1 & 1\end{bmatrix} \qquad \begin{matrix}\text{Walsh basis}\\(\text{NE, NW})\end{matrix} : \begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$$

Given that we have a function described using one basis how do we convert it to another? For GA researchers this is important. We want to analyse functions by converting them to partition coefficients and create new functions from partition coefficients.

A basis matrix translates a journey into map coordinates. Take the partition journey to Goldbergston as an example:

$$\text{Map coords} = \text{Basis matrix} \times \text{Journey}$$

$$\begin{bmatrix}y\\x\end{bmatrix} = \begin{bmatrix}1 & -1\\1 & 1\end{bmatrix}\begin{bmatrix}4 & \text{(Number of NE)}\\1 & \text{(Number of SE)}\end{bmatrix}$$

$$= \begin{bmatrix}3\\5\end{bmatrix}$$

To translate from map coordinates to a journey, the inverse of the basis matrix can be used. Combining these two ideas allows us to change basis.

To change the basis of a function from B_1 to B_2, you first convert to "map coordinates" by multiplying by B_1 and then convert to the new basis by multiplying by B_2^{-1}, the inverse of the new basis matrix. This conversion can be done by a single *transform matrix*. The following table shows various transform matrices and the calculation that forms them:

From	To	Calculation	Transform Matrix
Partition	Fitness	$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}^{-1}\begin{bmatrix}1 & -1\\1 & 1\end{bmatrix}$	$\begin{bmatrix}1 & -1\\1 & 1\end{bmatrix}$
Fitness	Partition	$\begin{bmatrix}1 & -1\\1 & 1\end{bmatrix}^{-1}\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$	$\begin{bmatrix}\frac{1}{2} & \frac{1}{2}\\-\frac{1}{2} & \frac{1}{2}\end{bmatrix}$
Partition	Walsh	$\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}^{-1}\begin{bmatrix}1 & -1\\1 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$

So far all of these matrices have been for single bit functions. The solution of such functions is not regarded as a pressing area of research. We also haven't tied this up with the "wave" view of the previous section. Each wave is also a "direction" (i.e. vector) in the fitness space.

So each column of the basis matrix is formed by calculating values of a particular wave across all points in the space. For example, here's the partition to fitness transform matrix (which equals the partition basis matrix) in use on a 2 bit problem:

$$\begin{bmatrix} f(00) \\ f(01) \\ f(10) \\ f(11) \end{bmatrix} = \begin{bmatrix} E_{\#\#}(00) & E_{\#1}(00) & E_{1\#}(00) & E_{11}(00) \\ E_{\#\#}(01) & E_{\#1}(01) & E_{1\#}(01) & E_{11}(01) \\ E_{\#\#}(10) & E_{\#1}(10) & E_{1\#}(10) & E_{11}(10) \\ E_{\#\#}(11) & E_{\#1}(11) & E_{1\#}(11) & E_{11}(11) \end{bmatrix} \begin{bmatrix} \varepsilon_{\#\#} \\ \varepsilon_{\#1} \\ \varepsilon_{1\#} \\ \varepsilon_{11} \end{bmatrix} = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \varepsilon_{\#\#} \\ \varepsilon_{\#1} \\ \varepsilon_{1\#} \\ \varepsilon_{11} \end{bmatrix}$$

The transform matrices for any sized problem can be calculated in the same way as the 1 bit matrices were.

4 Conclusion

We've seen two perspectives on partition functions. By viewing them as waves we saw that they provide an alternative way to describe a function. Waves also helped us to understand their effects in multidimensional space. Then, we took a wider view and saw how partition coefficients specify a journey that takes us to a particular function. Partition functions specify the directions we could use, but by choosing other directions we could specify a function in a different way (i.e. using Walsh functions). By choosing "directions" that have some meaning in terms of characteristics of the functions (such as "average value") we can potentially analyse and build functions in many different ways.

Acknowledgements

Thanks to Don Beal who has commented on more drafts than a double-glazing salesman.

References

Bridges, C.L. and Goldberg, D.E. (1991), The Nonuniform Walsh-Schema Transform in Rawlins G. (Ed), *Foundations of Genetic Algorithms,* Morgan Kaufmann

Field, P. (1994), Nonbinary Transforms for Genetic Algorithm Problems. Presented at the *AISB 1994 Workshop on Evolutionary Computing*, Univeristy Of Leeds, England. Proceedings to be published by Springer-Verlag.

Goldberg, D.E. (1989a), Genetic Algorithms and Walsh Functions: Part I, A Gentle Introduction, *Complex Systems* 3, pp.129-152

Goldberg, D.E. (1989b), Genetic Algorithms and Walsh Functions: Part II, Deception and Its Analysis, *Complex Systems* 3, pp.153-171

Goldberg, D.E. (1989c), *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley

Grefenstette, J.J. (1992), Deception Considered Harmful. In Whitley (Ed), *Foundations of Genetic Algorithms 2,* Morgan Kaufmann

Mason, A.J. (1991), Partition Coefficients, Static Deception and Deceptive Problems for Non-Binary Alphabets. In Belew, R.K. and Booker, L.B. (Eds) (1991), *Proceedings of the Forth International Conference on Genetic Algorithms*, Morgan Kaufmann

Strang, G. (1980), *Linear Algebra and its Applications (2nd Ed)*, Academic Press