RZ 3459 (# 93845)
Electrical Engineering 14 pages

Research Report

Complexity analysis of Fourier-transform decoding of LDPC Codes over $GF(q)$

Edward Ratzer

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Complexity analysis of Fourier-transform decoding of LDPC Codes over GF(q)

Edward Ratzer

August 11, 2000

Abstract

The complexity of Fourier-transform decoding of Low-Density Parity-Check (LDPC) codes over a finite field GF(2^p) is studied. The Fourier algorithm is found to have improved performance over GF(2) when column weight > 3.

1 Introduction

LDPC codes are a class of linear block codes. A binary LDPC code is defined by its sparse parity-check matrix, $\mathbf{H}$, often constructed to have particular column and row weights. Codes constructed in this manner are known to have intrinsically good performance using a maximum likelihood decoder. See [5] for a comprehensive coverage of LDPC codes.

One can instead define an LDPC code over GF(q)[1] (see [4] for an introduction to finite fields). Each symbol in $\mathbf{H}$ and hence $\mathbf{s}$ and $\mathbf{t}$ (the message and the transmitted codeword respectively) is then a member of the finite field. However for many channels (including hard disc drives) binary transmissions are needed, hence this form of $\mathbf{t}$ is not very useful. We therefore want to be able to define the mapping GF(q) $\mapsto \{0,1\}^p$, $q = 2^p$ is chosen as we can then carry out this mapping without wasting transmitted data bits and this always forms a field.

Current decoding algorithms use $\mathbf{H}$ to define a bi-partite graph on which an iterative message-passing decoding algorithm can be used. However this algorithm is not known to be optimal, one reason is the presence of cycles. By using an $\mathbf{H}$ defined over a larger field than GF(2) we can eliminate some cycles that would otherwise be present in the equivalent $\mathbf{H}$ defined over GF(2) [1] – this can then be expected to increase the performance of the decoding algorithm.

By enlarging the field size the computation required at each node of the graph increases. Fourier transform decoding as suggested by Richardson and Urbanke [6] is expected to reduce this complexity. The aim of this work was to quantify this result.

2 Decoding on a graph

A parity-check matrix fragment:

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 0 & \cdots \\ 0 & 1 & 1 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \qquad (1)$$

will create the graph fragment shown in figure 1.

[1]Tanner [7] first suggested having more complex constructions than binary check nodes.

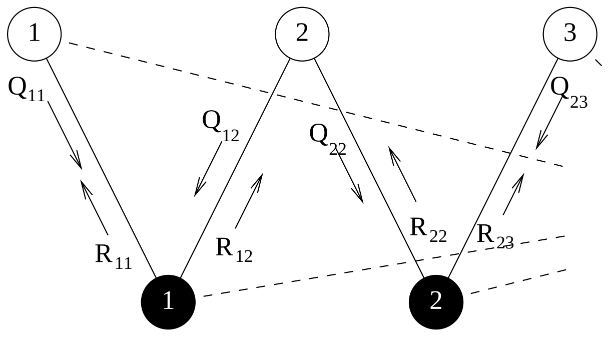


Figure 1: A graph fragment corresponding to Equation 1. The check nodes and symbol nodes are the filled and open circles respectively. The row and column weights can be seen to be 3 and 2 respectively

2.1 Check Node Computation

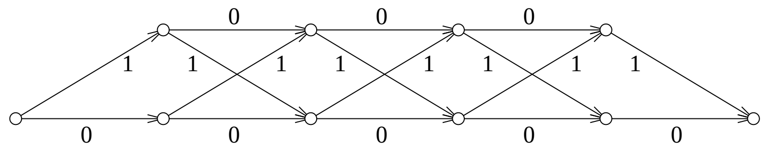
Each check node i sends to symbol node j a message $\mathrm{R}_{ij}(a)$ that is an approximation to the probability of check i being satisfied if the symbol node j is in state a:

$$\begin{aligned}
\mathrm{R}_{ij}(a) &= \Pr(\textstyle\sum_l H_{il}x_l = 0 | x_j = a) && (2)\\
&= \sum_{\boldsymbol{x}:x_j=a} \Pr(\textstyle\sum_l H_{il}x_l = 0|\boldsymbol{x}) \Pr(\boldsymbol{x}|x_j = a) && (3)\\
&\approx \sum_{\boldsymbol{x}:x_j=a} \Pr(\textstyle\sum_l H_{il}x_l = 0|\boldsymbol{x}) \prod_{k\neq j} \mathrm{Q}_{ik}(x_k) && (4)\\
&\approx \sum_{\boldsymbol{x}\in C_i:x_j=a} \prod_{k\neq j} \mathrm{Q}_{ik}(x_k) && (5)
\end{aligned}$$

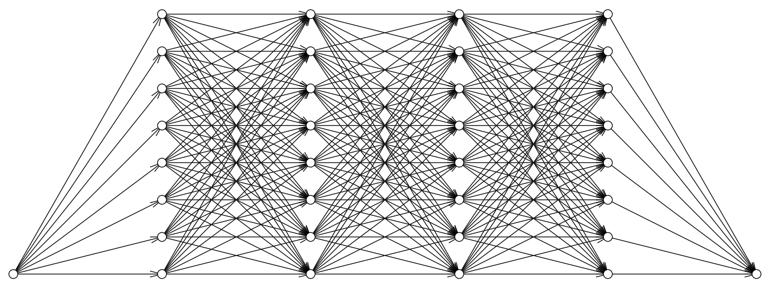
where $\mathrm{Q}_{ij}(a)$ are the messages received from the symbol nodes (an approximation to the probability that symbol node j is in state a according to the other check nodes) and C_i is the set of all the valid code words of the check node. This can then be evaluated by the forward-backward algorithm [2]. In the binary case one can simply use a small trellis, figure 2(a).

Over larger finite fields the trellis becomes more complicated, figure 2(b), making the process computationally demanding. However by using a Fourier representation one can reduce the complexity as follows:

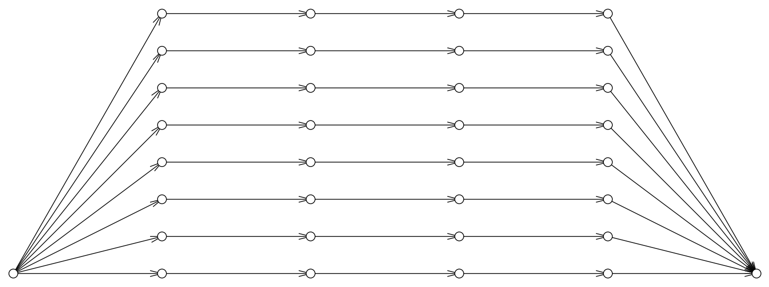
$$\begin{aligned}
\mathrm{R}_{ij}(a) &= \sum_{\boldsymbol{x}\in C_i} \delta(x_j = a) \prod_{k\neq j} \mathrm{Q}_{ik}(x_k) && (6)\\
&= \frac{|C_i|}{|\mathrm{GF}(q)^n|} \sum_{\boldsymbol{x}'\in C_i^\perp} \mathcal{F}[\delta(\cdot = a)](x'_j) \prod_{k\neq j} \mathcal{F}\left[\mathrm{Q}_{ik}\right](x'_k) && (7)\\
&= \tfrac{1}{q} \sum_{\boldsymbol{x}'\in C_i^\perp} \sum_{x_j} \langle x_j, x'_j\rangle \delta(x_j = a) \prod_{k\neq j} \mathcal{F}\left[\mathrm{Q}_{ik}\right](x'_k)\\
&= \tfrac{1}{q} \sum_{\boldsymbol{x}'\in C_i^\perp} \langle a, x'_j\rangle \prod_{k\neq j} \mathcal{F}\left[\mathrm{Q}_{ik}\right](x'_k)\\
\mathcal{F}[\mathrm{R}_{ij}](a') &= \tfrac{1}{q} \sum_a \langle a, a'\rangle \sum_{\boldsymbol{x}'\in C_i^\perp} \langle a, x'_j\rangle \prod_{k\neq j} \mathcal{F}\left[\mathrm{Q}_{ik}\right](x'_k)\\
&= \tfrac{1}{q} \sum_{\boldsymbol{x}'\in C_i^\perp} q\delta(x'_j = a') \prod_{k\neq j} \mathcal{F}\left[\mathrm{Q}_{ik}\right](x'_k)
\end{aligned}$$



(a) C over GF(2)



(b) C over GF(8)



(c) $C^{\perp}$ over GF(8)

Figure 2: Example trellises

$$\Rightarrow \mathcal{F}[\mathrm{R}_{ij}](a') \quad = \quad \sum_{\boldsymbol{x}' \in C_i^{\perp} : x'_j = a'} \prod_{k \neq j} \mathcal{F}[\mathrm{Q}_{ik}](x'_k) \tag{8}$$

where in going from equation 6 to equation 7 we have applied the Poisson summation formula as suggested by Forney [3] to get to a summation over the dual code ($C_i^{\perp}$) and we can split up the Fourier transform as each term is in a separate dimension. Note equation 8 has the same structure as equation 5 and hence can similarly be evaluated by the sum-product algorithm, but instead over the dual code with Fourier transformed messages.

The trellis for the dual code of each check node becomes trivial (the dual code is one-dimensional), figure 2(c), due to the multiplication properties of the finite field. Hence the addition in equation 8 is only over one term. There are only now q connections per symbol node in the middle of the trellis, rather than q^2 as previously. This leads to an important reduction in complexity as one increases the field size.

2.2 Symbol Node Computation

We need to evaluate both the messages to be passed back to the check nodes $\mathrm{Q}_{ij}(a)$ and a tentative decoding for that symbol node. $\mathrm{Q}_{ij}(a)$ is an approximation to the probability that symbol node j is in state a according to check nodes other than i:

$$\begin{aligned} \mathrm{Q}_{ij}(a) &= \Pr(x_j = a | \text{check nodes other than } i \text{ satisfied}) && (9) \\ &= Z_{ij} \Pr(x_j = a) \Pr(\text{check nodes other that } i \text{ satisfied} | x_j = a) && (10) \\ &\approx Z_{ij} \mathrm{f}_j(a) \prod_{k \neq i} \mathrm{R}_{kj}(a) && (11) \end{aligned}$$

where $\mathrm{f}_j(a)$ is the prior probability that node j is in state a and Z_{ij} is a normalizing constant.

The tentative decoding for that symbol in a similar manner is:

$$\begin{aligned} \hat{t}_j &= \max_a \left(\Pr(x_j = a | \text{check nodes satisfied}) \right) && (12) \\ &\approx \max_a \left(\mathrm{f}_j(a) \prod_k \mathrm{R}_{kj}(a) \right) && (13) \end{aligned}$$

2.3 Iteration

$\mathrm{Q}_{ij}(a)$ are first initialized to the channel probabilities of symbol node j being in state a. Then messages are passed on the bi-partite graph in chosen manner (typically all check nodes are updated, followed by all symbol nodes and this is then repeated) until a successful decoding results or a maximum number of steps has been taken.

3 Complexity of the GF(2) Non-Fourier algorithm

We will aim to express things in terms of parameters of the $\mathbf{H}$ matrix. This will be an $M \times N$ matrix – this then forms a graph with M check nodes and N symbol nodes. This matrix will also be defined to have fixed column and row weights of m and n. In other words each check node is connected to n symbol nodes and each symbol node is connected to m check nodes.

3.1 Check Node

We will start by evaluating the number of calculations needed at a single check node i. To do this the forward-backward algorithm will be used on a trellis like figure 2(a).

The forward α and backward β probabilities will be defined as follows:

$$\alpha_{ij}(a) = \Pr\left(\sum_{k\le j}\mathbf{H}_{ik}x_k = a\right) \tag{14}$$

$$\beta_{ij}(a) = \Pr\left(\sum_{k> j}\mathbf{H}_{ik}x_k = a\right) \tag{15}$$

a can be seen as representing the level on the trellis in this case. These probabilities can then be easily calculated on the trellis by iteration, for example:

$$\alpha_{ij}(1) = \mathrm{Q}_{ij}(0)\alpha_{i,j-1}(1) + \mathrm{Q}_{ij}(1)\alpha_{i,j-1}(0) \tag{16}$$

$$\beta_{ij}(1) = \mathrm{Q}_{i,j+1}(0)\beta_{i,j+1}(1) + \mathrm{Q}_{i,j+1}(1)\beta_{i,j+1}(0) \tag{17}$$

$\alpha_{ij}(0)$ and $\beta_{ij}(0)$ do not need to be evaluated explicitly due to the normalization condition. Therefore we can instead express equations 16 and 17 as:

$$\alpha_{ij}(1) = (\mathrm{Q}_{ij}(0) - \mathrm{Q}_{ij}(1))\alpha_{i,j-1}(1) + \mathrm{Q}_{ij}(1) \tag{18}$$

$$\beta_{ij}(1) = (\mathrm{Q}_{i,j+1}(0) - \mathrm{Q}_{i,j+1}(1))\beta_{i,j+1}(1) + \mathrm{Q}_{i,j+1}(1) \tag{19}$$

The boundary conditions are:

$$\alpha_{i0}(0) = \beta_{in}(0) = 1 \tag{20}$$

$$\alpha_{i0}(1) = \beta_{in}(1) = 0 \tag{21}$$

We need the α and β values on $n-2$ non-trivial states, this takes $2(n-2)$ multiplications and $4(n-2)$ additions in total.

We can then obtain the R_{ij} values by multiplication and addition:

$$\mathrm{R}_{ij}(1) = \alpha_{i,j-1}(1)\beta_{ij}(0) + \alpha_{i,j-1}(0)\beta_{ij}(1) \tag{22}$$

$$= \alpha_{i,j-1}(1) + \beta_{ij}(1) - 2\alpha_{i,j-1}(1)\beta_{ij}(1) \tag{23}$$

$$\mathrm{R}_{ij}(0) = 1 - \mathrm{R}_{ij}(1) \tag{24}$$

To do this most efficiently we only need to apply the full equation 23 to $n-2$ states – over the two remaining states the boundary conditions make the equation trivial. Equation 24 needs to be applied to all n states. Calculation of all R_{ij} values hence takes $2(n-2)$ multiplications and $3n-4$ additions. It should be noted though that $(n-2)$ of these multiplications are easy as they are just multiplication by 2 – on a computer this can quickly be done just using a bit shift or addition of 1 to the exponent of a floating point number.

Therefore each check node in total requires $4(n-2)$ multiplications and $7n-12$ additions.

3.2 Symbol Node

We will again use forward-backward algorithm to evaluate Q_{ij} and $\hat{t}_j$. An additional state will be used to deal with the f_j term, as shown in figure 3. We define the forward and backward probabilities to be:

$$\alpha'_{ij}(a) = \mathrm{f}_j(a)\prod_{k\le i} R_{kj}(a) \tag{25}$$

$$\beta'_{ij}(a) = \prod_{k>i} R_{kj}(a) \tag{26}$$

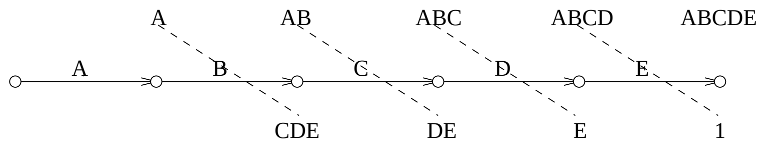


Figure 3: The forward-backward algorithm at an $m = 4$ symbol node. The linear structure for just one particular member of GF(q) is shown. 'A' represents the additional state for f_j and 'B'-'E' the $\mathrm{R}_{kj}(a)$ messages. The top row shows the α' values and the bottom row the β' values. The dotted lines represent the needed multiplications to obtain values of $\mathrm{Q}_{ij}(a)$ from these.

Again we will break the computation down into steps:

1. **Evaluation of $\boldsymbol{\alpha}'$ and $\boldsymbol{\beta}'$** We need to know all the values of α' (taking $2m$ multiplications) and mq values of β' (taking $2(m-2)$ multiplications). This takes a total of $4(m-1)$ multiplications.
2. **Evaluation of unnormalized $\mathbf{Q}_{ij}$** We simply multiply values of α' and β' separated by one, as shown in figure 3, this takes $2(m-1)$ multiplications.
3. **Normalization** This is done by adding up each pair of probabilities and then dividing each probability by the result – this takes m additions, m multiplicative inverses and $2m$ multiplications (note the inverses and multiplications could be replaced by $2m$ divisions but we will again assume that multiplication is preferable).
4. **Evaluation of $\hat{t}_j$** We just need to compare $\alpha'_{mj}(1)$ and $\alpha'_{mj}(0)$ and find the maximum. This takes 1 comparison.

This comes to a total of $2(4m-3)$ multiplications, m additions, m multiplicative inverses and 1 comparison.

Alternatively one could multiply all the R_{kj} terms and then use division to remove the unneeded ones – this takes less operations in total however it is likely that division will be significantly slower than multiplication and hence be slower overall.

3.3 Syndrome check

After calculating all the $\hat{t}_j$ we then want to check whether this is a valid decoding. To do this we just test whether

$$\hat{\mathbf{t}}\mathbf{H}^{\top} = \mathbf{0}.$$

H is sparse and hence the entire matrix multiplication does not need to be carried out, instead we can just concentrate on the non-'0' entries. This will take Mn binary multiplications, $M(n-1)$ binary additions and M comparisons.

3.4 Total complexity per iteration

In each iteration we will update all the check nodes, all the symbol nodes and then do a syndrome check. There are M check nodes and N symbol nodes – hence we can calculate the total number of operations per iteration, this is shown in table 1.

Table 1: Complexity per iteration for non-Fourier algorithm

Operation	Number
Real multiplications	$4M(n-2)+2N(4m-3)$
Real additions	$M(7n-12)+Nm$
Binary multiplications	Mn
Binary additions	$M(n-1)$
Multiplicative inverses	Nm
Comparisons	$N+M$

(a) Complexity in terms of parameters of the **H** matrix

Operation	Number
Real multiplications	$4(3m+2R)-14$
Real additions	$8m+12(R-1)$
Binary multiplications	m
Binary additions	$m+R-1$
Multiplicative Inverses	m
Comparisons	$2-R$

(b) Complexity per transmitted bit in terms of rate

However it is useful to express things in terms the rate of the resultant code. Assuming linearly independent rows in **H** this gives:

$$\text{Rate} = R = \frac{N-M}{N} \Rightarrow M = N(1-R) \tag{27}$$

Also the total number of connections in the graph must be the same both from the point of view of the check nodes and the symbol nodes; we can therefore remove the dependency on the row weight:

$$Mn = Nm \Rightarrow n = \frac{m}{1-R} \tag{28}$$

Then to get a complexity per transmitted bit per iteration, as shown in table 2(b), we need to divide by N.

It is worth remembering that m might be a function of the blocklength so the complexity might not scale linearly with block length. Also meaningful comparisons are only possible if we assume the number of iterations required are similar for all codes.

4 GF(q) Fourier-Decoding Implementation

As a test of concept, the Fourier-transform decoding over GF(2^p) was implemented in form of a MatLab program.

4.1 Representation of GF(2^p)

Internally in the program each element of the finite field was given two representations:

$$\mathbf{F} : F(h_1, h_2) = \sum_{g_1, g_2} \mathbf{F}_{(h_1,h_2),(g_1,g_2)} f(g_1, g_2)$$

$$\begin{array}{l} \\ (1,-1) \\ (1,1) \\ (-1,1) \\ (-1,-1) \end{array} \begin{array}{c} \begin{array}{cccc} (1,-1) & (1,1) & (-1,1) & (-1,-1) \end{array} \\ \begin{pmatrix} -1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

Figure 4: Fourier transformation matrix **F**

Logarithmic The elements excluding the '0' element were given in terms of powers of a generating element for this cyclic sub-group. Hence multiplication of elements becomes simple. The '0' element was represented by q - this was so the representation could be used as the index of an array in MatLab.

GF(2)^p A look-up table (indexed by the previous representation) of vectors was used. By choosing the vectors appropriately addition could then be carried out using vector addition modulo 2.

4.2 Fourier transformations

The Fourier transformation is defined as:

$$F(h_1, h_2, \cdots) \equiv \mathcal{F}[f](h_1, h_2, \cdots) \triangleq \sum_{(g_1, g_2, \cdots) \in \mathrm{GF}(2)^p} (-1)^{h_1 g_1 + h_2 g_2 + \cdots} f(g_1, g_2, \cdots) \tag{29}$$

where we have mapped $\mathrm{GF}(2^p) \mapsto \mathrm{GF}(2)^p$ and used the character of GF(2) that maps the elements on to $\{-1, 1\}$.

We can view this process as the application of a matrix similar to a Hadamard matrix, for example figure 4.

4.3 Binary communication

It is assumed that the sender, receiver and channel all want data in a binary form. Therefore the $\mathrm{GF}(2)^p$ representation was used, first to initially convert the data to be sent to elements of $\mathrm{GF}(2^p)$ and then also in reverse after encoding to get back to binary form. A similar process was done during the decoding however one now needs to keep track of probabilities. For example if symbol a_0 is represented by `00` then $\Pr(\boldsymbol{x} = a_0) = \prod_{i=1,2} \Pr(x_i = \mathtt{0})$. The individual bit probabilities can be obtained by a Bayesian calculation based on the channel properties and the received data.

4.4 Messages

In the binary case only $\mathrm{R}_{ij}(0)$ and $\mathrm{Q}_{ij}(0)$ were sent to save memory as, for example, $\mathrm{R}_{ij}(1) = 1 - \mathrm{R}_{ij}(0)$. Over $\mathrm{GF}(q)$ we need to send a vector of probabilities – all the elements were sent for simplicity of programming.

The update routine used was kept the same as in the original program. $\mathrm{Q}_{ij}(a)$ were first initialized to the channel probabilities of node j being in state a. All the $\mathrm{R}_{ij}(a)$ were updated, followed by all the $\mathrm{Q}_{ij}(a)$. This was then repeated until a successful decoding resulted or the maximum number of iterations had been reached.

4.5 Creation of H and G

To use the program we needed to generate valid **H** and **G** (the generator for the code) matrices. Initially **H** was created by using a binary construction to obtain regular toy LDPC codes (of size 14×49) and replacing all the links by random q-ary values. A generator matrix **G** in systematic form was obtained from **H** by applying Gaussian elimination.

5 Complexity of the Fourier Decoding algorithm

In in similar manner to section 3 we will calculate the complexity of the Fourier decoding algorithm, but we generalize the result to be over field size q. The rest of the notation will be the same as used before. The major changes are the Fourier transformation and the way the probability distributions are normalized. All of the messages will now be unnormalized and the normalization will be done once per iteration in the check nodes. This leads to a reduction in complexity as explained below.

5.1 Check Node

Again the forward-backward algorithm will be used to carry out the calculations in Fourier space, our trellis now looks like figure 2(c). The trellis is not time-invariant and hence one needs to be careful with how the forward and backward probabilities are defined. We label the kth symbol of dual code word j for check node i as d_{ij}^k. The forward and backward probabilities then are defined as:

$$\alpha''_{ij}(a) = \prod_{k \leq j} \mathcal{F}[\mathrm{Q}_{ik}](d_{ia}^k) \tag{30}$$

$$\beta''_{ij}(a) = \prod_{k > j} \mathcal{F}[\mathrm{Q}_{ik}](d_{ia}^k) \tag{31}$$

We can then recursively evaluate these in the similar manner to section 3.2. To obtain the Fourier transformed R_{ij} we then just have to multiply these:

$$\mathcal{F}[\mathrm{R}_{ij}](d_{ia}^j) = \alpha''_{i,j-1}(a)\beta''_{ij}(a) \tag{32}$$

The algorithm breaks down into several steps:

1. **Fourier Transformation** Using the method in figure 4 we are multiplying a $n \times q$ matrix (there are n rows as we will Fourier transform the messages from all connected symbol nodes at the same time) by a $q \times q$ matrix (**F**, which can be pre-calculated) and so we would expect that nq^2 multiplications and $nq^2 - nq$ additions are needed. However the matrix **F** consists of only 1 and -1 entries so we can expand the calculation out into instead $nq^2 - nq$ additions or subtractions only.

   The biggest saving can be gained though by seeing that our matrix is a permutation of a Hadamard matrix. We can then split the calculation up into pairs and additions and subtractions. For example in figure 5 it can be seen that four additions or subtractions are repeated and hence need only be done once. In general this then gives us $nq \log_2 q$ additions or subtractions[2]. This

[2]This was not implemented in the MatLab sample program so the size of the finite field could more easily be changed, instead a fast code-generating program was developed for later incorporation into C++.

$$\mathbf{F} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \qquad \begin{array}{rcl} F(0) & = & [f(0)+f(1)]+[f(2)+f(3)] \\ F(1) & = & [f(0)-f(1)]+[f(2)-f(3)] \\ F(2) & = & [f(0)+f(1)]-[f(2)+f(3)] \\ F(3) & = & [f(0)-f(1)]-[f(2)-f(3)] \end{array}$$

Figure 5: An example Fourier transform (after [1])

result stems from the fact that at each level of pairing there are q additions or subtractions and there are $\log_2 q$ levels of pairing.

It should be noted that this does not increase the memory usage as the inputs (or previous level of calculations) can be discarded once each pair of an addition and subtraction is done[3].

2. **Normalization** We will normalize each row in the resultant $n \times q$ matrix now as this is the most efficient time to do it. This is because the '0' element of each Fourier transform is just the required normalization factor for that row and hence we do not need to reevaluate it. This takes n multiplicative inverses and $n(q-1)$ multiplications.

3. **Evaluation of $\boldsymbol{\alpha}''$ and $\boldsymbol{\beta}''$** The forward and backward probabilities each require $(n-2)(q-1)$ multiplications (the first and last state is either known or not needed in later calculations and the '0' row of the trellis is trivial as all α'' and β'' values are just 1).

4. **Calculation of Fourier Transformed Probabilities** This multiplication of α'' and β'' requires $(n-2)(q-1)$ multiplications (the first and last states are just β'' or α'' respectively and the '0' row is again trivial).

5. **Inverse Fourier Transform** The Fourier transformation used is proportional to an involution (a self-inverse). Thus to obtain unnormalized probabilities this requires the same $nq \log_2 q$ additions or subtractions as we did before. The normalizing factor is just q for all the probabilities – this scaling factor does not affect anything apart from the scale of the symbol node calculations and can be left in the resultant messages.

This leads to a total of $2nq \log_2 q$ additions or subtractions and $2(2n-3)(q-1)$ multiplications and n multiplicative inverses per check node.

5.2 Symbol Node

We will use the same technique as before (figure 3 and section 3.2) but now over q field elements rather than just 2. Also the probabilities are now unnormalized but this does not change the overall calculation.

1. **Evaluation of $\boldsymbol{\alpha}'$ and $\boldsymbol{\beta}'$** This takes $2(m-1)q$ multiplications.

2. **Evaluation of unnormalized $\mathbf{Q}_{ij}$** This takes $(m-1)q$ multiplications.

3. **Evaluation of $\hat{t}_j$** $q-1$ comparisons are needed.

This comes to a total of $3(m-1)q$ multiplications and $q-1$ comparisons per symbol node.

[3]Thanks to D.J.C. MacKay for pointing this out.

5.3 Syndrome check

We will do exactly the same as in section 3.3. However our additions and multiplications are now over the field GF(q) rather than being binary. This comes to Mn GF(q) multiplications, $M(n-1)$ GF(q) additions and M comparisons.

5.4 Total complexity per iteration

As in the binary case the total complexity per iteration is shown in table 2(a) and (b).

Table 2: Complexity per iteration

Operations	Number
Real multiplications	$2M(2n-3)(q-1)+3N(m-1)q$
Real additions	$2Mnq\log_2 q$
GF(q) multiplications	Mn
GF(q) additions	$M(n-1)$
Multiplicative inverses	Mn
Comparisons	$N(q-1)+M$

(a) Complexity in terms of parameters of the **H** matrix

Operation	Number
Real multiplications	$\frac{(7m-9)q+6(R(q-1)+1)-4m}{\log_2 q}$
Real additions	$2mq$
GF(q) multiplications	$\frac{m}{\log_2 q}$
GF(q) additions	$\frac{m+R-1}{\log_2 q}$
Multiplicative Inverses	$\frac{m}{\log_2 q}$
Comparisons	$\frac{q-R}{\log_2 q}$

(b) Complexity per transmitted bit in terms of rate

As before note that meaningful comparisons are only possible if we assume the number of iterations are similar for all codes.

6 Discussion

We can now directly compare the Fourier algorithm with the non-Fourier algorithm in the case of $q=2$. This is shown in table 3. It can be seen that the only difference occurs for real multiplications and additions.

For the Fourier algorithm to be more efficient in multiplications and additions we therefore respectively require:

$$m > 1-R \tag{33}$$

$$m > 3(1-R) \tag{34}$$

It is likely that both of these will be satisfied. Any LDPC code will satisfy equation 33 as m always has to be greater than 1. Equation 34 will often be satisfied as m

Table 3: Comparison of the complexity of the Fourier and non-Fourier decoding algorithms over GF(2)

Operation	Non-Fourier number	Fourier number
Real multiplications	$4(3m + 2R) - 14$	$10m - 12 + 6R$
Real additions	$8m + 12(R - 1)$	$4m$
Binary multiplications	m	m
Binary additions	$m + R - 1$	$m + R{-}1$
Multiplicative inverses	m	m
Comparisons	$2 - R$	$2 - R$

is often greater than 3. Therefore Fourier decoding algorithm is liable to be more efficient even over GF(2).

As q is increased we can hope for similar behaviour as the Fourier algorithm is $\mathcal{O}(q)$ whereas a traditional algorithm is $\mathcal{O}(q^2/\log_2 q)$ (as shown by the complexity of the trellis).

This work does not address the comparative performance of these codes however [1] (and Appendix A) shows that codes over larger finite fields seem to perform better than binary codes. Once reliable performance guides are obtained it would be informative to then study the payoff between performance and complexity.

Another direction for future work would be studying approximations of the decoding techniques. The traditional decoding algorithm has had several approximations applied to it which reduce the complexity without impacting a code's performance too much – finding similar approximations that work in the Fourier domain would further increase the usefulness of Fourier decoding.

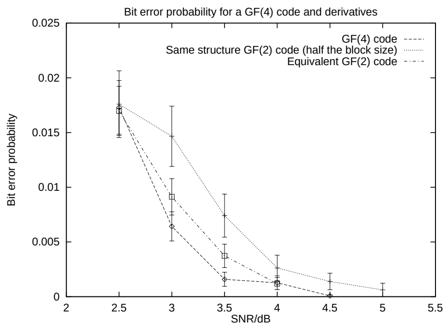
7 Conclusion

Moving to Fourier transform decoding reduces the complexity of the decoding algorithm in the binary case and also allows easier use of larger finite fields as the decoding algorithm only scales as q.

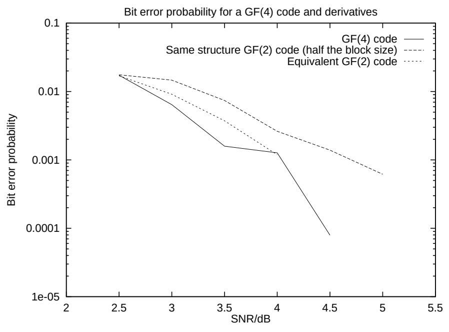
A Preliminary Performance Guide

For the case $m = 2$, a few small tests using some toy LDPC code were run in MatLab to get a feeling of how these codes perform.

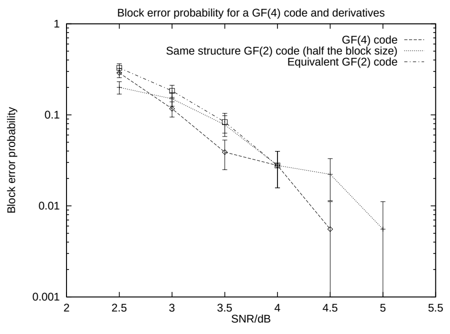
Three codes were studied. A GF(4) code was taken as the starting point. This was then compared to two different GF(2) codes, one with the same graph (and hence half the block size) and one which would result in the same set of binary code words being transmitted (each GF(4) symbol in $\mathbf{H}$ was replaced with the 2×2 matrix representation of the element). The results are shown in figure 6. These derive from small runs in MatLab with a maximum of 10 iterations; they should not be trusted too much at very low error rates. It does give the indication though that the GF(4) code is outperforming the two other derivatives. The lower performance of the same code defined over GF(2) suggests that the decoding algorithm performance increases as it is defined over larger finite field (as a caveat see [1] for how performance depends on m). The same structure code is probably performing worse as it has a smaller block size.



(a) Linear Scale Bit Error Probability



(b) Log Scale Bit Error Probability



(c) Log Scale Block Error Probability

Figure 6: Preliminary $m = 2$ performance curves

B Minimum Distance

A guide to the performance of a code is the minimum distance[4]; this is the shortest Hamming distance between any two code words. As LDPC codes are linear codes we can instead find the minimum weight code word (the code with the smallest Hamming distance from the zero code word). All code words have the property that $\mathbf{t}\mathbf{H}^{\top} = \mathbf{0}$. This allows us to instead look for the smallest set of linearly independent columns in $\mathbf{H}$.

For the toy example matrix over the binary field the minimum distance is 4, for the same matrix with random symbols from GF(4) the minimum distance is also 4. The code generated by the GF(4) matrix is twice as long in binary terms as the other code – therefore having the same minimum distance is not an ideal property. Each code can hence only reliably correct 1 bit flipped in the entire block.

Finding the minimum weight codeword is still a computationally hard problem and over GF(4) takes a long time to complete when a simple search is used. An upper bound can be found by looking at the $\mathbf{H}$ matrix in row-reduced echelon form. One can combine the minimum weight column—excluding columns of weight 1 or less—with a set of columns of weight 1 to form linearly dependent columns. Therefore the bound is simply $1 +$ the weight of this column. For the two matrices cited above this upper bound was reached.

References

[1] M.C. Davey. *Error-Correction Using Low-Density Parity-Check Codes.* PhD thesis, University of Cambridge, 1999. Available on the web at `http://wol.ra.phy.cam.ac.uk/mcdavey`.

[2] R. Durbin et al. *Biological sequence analysis: probabilistic models of proteins and nucleic acids.* Cambridge University Press, 1998.

[3] G. David Forney, Jr. Codes on graphs: Generalised state realizations. Submitted to IEEE Transactions on Information Theory, 1998.

[4] Lidl and Niederreiter. *Introduction to Finite Fields and their applications.* Cambridge University Press, revised edition, 1994.

[5] D.J.C. MacKay. Good error correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory*, 45(2):399–431, 1999.

[6] T. Richardson and R. Urbanke. The capacity of low-density parity check codes under message-passing decoding. Submitted to IEEE Transactions on Information Theory, available from `http://cm.bell-labs.com/cm/ms/former/tjr/pub.html`, 1998.

[7] R.M. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547, 1981.

[4]LDPC codes can often be decoded beyond their minimum distance [5].