IMPLEMENTATION OF AN LDPC DECODER USING FUNCTIONAL PROGRAMMING LANGUAGES

Brett W. Werling
Department of Electrical Engineering & Computer Science
University of Kansas
Lawrence, KS 66045
bwerling@ittc.ku.edu

Faculty Advisors:
Erik Perrins and Andy Gill

ABSTRACT

In this paper we present an implementation of a low density parity check (LDPC) decoder in the functional programming language Haskell. We describe the LDPC code in a very general sense and show how it is used in our implementation. We then discuss the advantages of using a functional programming language like Haskell to model this decoder, as well as the implications of doing so. Finally, we evaluate our model in terms of algorithm accuracy.

INTRODUCTION

In many communication systems, forward error correction (FEC) coding schemes are used to improve link reliability and increase the range of operation. As the telemetry world begins to move toward integrated network enhanced telemetry (iNET), FEC codes become an obvious choice. Recent work has shown that low density parity check (LDPC) codes are very promising, as they have been shown to be capable of closely approaching the channel capacity [1]. In addition to that, the iterative decoding algorithms used in LDPC codes are highly parallelizable in hardware. This is advantageous for the hardware designer because the speed of the algorithm can be increased without changing the algorithm itself. How does one go about building such a system in hardware?

When designing any system that is destined for the hardware world, it is necessary to create some sort of model that can be simulated. Tools such as MATLAB are great for building simulations, but taking a MATLAB implementation of something and converting it into hardware description is not easily done. The functional programming language known as Haskell [2] is quite different, however. There are extensions which allow for high-level simulation-oriented models, as well as tools aimed at very low-level structural models which can be directly converted to VHDL.[1]

Generally speaking, Haskell offers expressive facilities comparable to MATLAB. Both allow for the rapid creation of high-level models, but simulation is typically where their capabilities end. These simulations are useful for analytical purposes and can help the engineer understand the system better. Once

[1] VHDL - VHSIC (Very High Speed Integrated Circuits) Hardware Description Language

the engineer is satisfied with the results, the real hardware must be built. Typically a whole new approach must be taken because there is no concept of structure or the wiring together of components.

For simpler systems, Haskell's extension Lava [3] is an effective tool. It gives the engineer the ability to rapidly build the structure of a system at the gate level, taking advantage of Haskell's features along the way. Structures can be pieced together to create more complex systems, and a few function calls can act as a form of simulation. Lava can take this low-level representation and output it in VHDL form. Unfortunately, current versions of Lava only allow the engineer to build a system from the gate level, something which can already be done in VHDL.

There is a strong desire for a way to build a high-level model of a system and compile it down to a hardware description language. The first step in the process is to create a high-level model of a real system. In this paper, we take that step by working with an LDPC decoding algorithm that we intend to build in hardware. The following are our contributions:

- We build a model of an LDPC decoder in Haskell.
- We explore alternative representations of the data structures within the decoding algorithm.
- We put the power of Haskell to the test by using concepts and operations that are not immediately available in a traditional imperative programming language.
- We take a working implementation of the decoder that was written in MATLAB and compare it with our own.

LDPC CODE

The purpose of this section is not to explain LDPC codes in detail, but to present necessary items for other sections. For more information on the basics of LDPC codes, see [1].

The structure in the decoder that contains the code information is the parity check matrix, A, which contains only 1's and 0's. From A, we derive $\mathcal{M}$ and $\mathcal{N}$, both of which are used in our decoding algorithm. Using standard set notation, they are defined as

$$\mathcal{M}_n = \{m : A_{m,n} = 1\}, \quad \mathcal{M}_{n,m} = \mathcal{M}_n \backslash m, \quad \text{where } \backslash m \text{ means "excluding } m\text{"} \tag{1}$$

$$\mathcal{N}_m = \{n : A_{m,n} = 1\}, \quad \mathcal{N}_{m,n} = \mathcal{N}_m \backslash n, \quad \text{where } \backslash n \text{ means "excluding } n\text{"} \tag{2}$$

For this model, we have already been given a parity check matrix. We use the code with rate 2/3 and information block size of 4096 that has been developed for deep space use [4].

LDPC DECODER ALGORITHM

We focus on the iterative log likelihood decoding algorithm for binary LDPC codes [1, p. 652]. Consider the decoder as a system with four inputs (A, $\mathbf{r}$, L, L_c) and one output ($\hat{\mathbf{c}}$). A is the parity check matrix of size $M \times N$, $\mathbf{r}$ is the received vector of length N, L is the maximum number of iterations before decoding failure, and L_c is the channel reliability. The output $\hat{\mathbf{c}}$ is a vector of length N containing the decoded[2] binary values.

[2]In some instances it is possible that the parity check passes and an incorrect codeword is found.

Within the algorithm there are two main structures, η and λ. The matrix η is of size $M \times N$ and contains the check nodes, and the vector λ is of length N and contains the bit nodes. The loop counter l keeps track of the algorithm iteration on which we are working. For example, $\eta_{m,n}^{[5]}$ means that $l = 5$. The initial states of η, λ, and l are

$$\eta_{m,n}^{[0]} = 0 \text{ if } A_{m,n} = 1 \text{ for } m = (0, 1, \ldots, M-1),\ n = (0, 1, \ldots, N-1)$$
$$\lambda_n^{[0]} = L_c r_n \text{ for } n = (0, 1, \ldots, N-1)$$
$$l = 1$$

The decoding algorithm can be described as a loop that is split into three steps, shown below.

Step 1 - Update the check nodes: For each (m,n) with $A_{m,n} = 1$: Compute

$$\eta_{m,n}^{[l]} = -2 \tanh^{-1} \left(\prod_{j \in \mathcal{N}_{m,n}} \tanh \left(\frac{\lambda_j^{[l-1]} - \eta_{m,j}^{[l-1]}}{2} \right) \right) \tag{3}$$

Step 2 - Update the bits nodes: For $n = (0, 1, \ldots, N-1)$: Compute

$$\lambda_n^{[l]} = L_c r_n + \sum_{m \in \mathcal{M}_n} \eta_{m,n}^{[l]} \tag{4}$$

Step 3 - Perform the check: For $n = (0, 1, \ldots, N-1)$:

$$\hat{c}_n = 1 \text{ if } \lambda_n^{[l]} > 0, \text{ otherwise } \hat{c}_n = 0$$

$$\begin{gathered} \text{If}(A\hat{\mathbf{c}}) \bmod 2 = 0 \text{ then } \textbf{Stop} \text{ and output } \hat{\mathbf{c}}. \\ \text{Otherwise, if } l < L, \text{ increment } l \text{ and jump to } \textbf{Step 1}. \\ \text{Otherwise, declare a coding failure, } \textbf{Stop}, \text{ and output } \hat{\mathbf{c}}. \end{gathered} \tag{5}$$

PARITY CHECK MATRIX

The parity check matrix we are given is of size 3072×7168, therefore having 22,020,096 elements. Using a brute-force implementation of the algorithm with 32-bit integers would require at least 672 MB of data memory. Even though memory is not at a premium on most systems anymore, that is a lot of information to store and process. In order to figure out a way around this, we first need to understand what kind of information is stored in A.

We know that A contains only 0's and 1's, and because this is an LDPC code, we know that A is sparse [1]. In fact, out of the 22,020,096 elements within our instance of A, only 23,552 of them have the value 1. From (5) we see that the only mathematical operation A is directly involved in is a matrix multiplication (mod 2). This means that we are only concerned with the 1's in A, as any 0 that gets multiplied will zero out its fellow operand. Thus, we only need to know the locations of A that contain

a 1. For example, say we have a parity check matrix $\hat{A}$ of size 5×6, shown below. If $\hat{w}$ represents $\hat{A}$ as an array of (row, column) pairs of locations of 1's, we have

$$\hat{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \implies \begin{array}{l} \hat{w}[0] = (0,1) \\ \hat{w}[1] = (0,4) \\ \hat{w}[2] = (1,3) \\ \hat{w}[3] = (1,5) \\ \hat{w}[4] = (2,0) \\ \hat{w}[5] = (2,3) \\ \hat{w}[6] = (4,1) \\ \hat{w}[7] = (4,2) \\ \hat{w}[8] = (4,5) \end{array}$$

Notice that this representation requires us to store 18 values. This is a decrease from the 30 that are required for the original $\hat{A}$, but we can optimize even further. If we use the row number as the index of an array, we only need to store the columns containing a 1. If $\hat{x}$ is an array representing $\hat{A}$, then

$$\begin{array}{l} \hat{x}[0] = \{1,4\} \\ \hat{x}[1] = \{3,5\} \\ \hat{x}[2] = \{0,3\} \\ \hat{x}[3] = \text{empty} \\ \hat{x}[4] = \{1,2,5\} \end{array}$$

We have reduced our number of stored values from 18 to 9, which is exactly the number of 1's in $\hat{A}$. If we follow the same procedure for A, the number of values drops from 22,020,096 to 23,552. That is a decrease of approximately 99.89% from a brute-force implementation. Even though we have a compact version of A, we still must check to make sure it meets our needs for the algorithm. Besides the parity check in (5), the only other main use of A is in (1) and (2), the derivation of $\mathcal{M}$ and $\mathcal{N}$.

Obtaining $\mathcal{N}$ is trivial. Let's say $\hat{\mathcal{N}}$ is the result of applying (2) to $\hat{A}$. We can see that $\hat{\mathcal{N}}_0 = \{1,4\}$. We can obtain the same result by accessing $\hat{x}[0]$. A simple array lookup in $\hat{x}$ provides us with the information needed for both $\hat{\mathcal{N}}_m$ and $\hat{\mathcal{N}}_{m,n}$.

Finding $\mathcal{M}$ is not as easy, at least not with our current data structure for A. Since we are only focusing on representing A by its rows, we have a harder time accessing a given column. A complex function can be written to emulate (1) and find $\mathcal{M}$ on a row-by-row basis, but such a function would dramatically increase the complexity of each algorithm loop. The way we solve this is by splitting our representation of A into two parts: one for the rows, and one for the columns. If we apply this and have the array $\hat{x}_1$ indexed by the rows of $\hat{A}$ and the array $\hat{x}_2$ indexed by the columns of $\hat{A}$, then we have

$$\begin{array}{l} \hat{x}_1[0] = \{1,4\} \\ \hat{x}_1[1] = \{3,5\} \\ \hat{x}_1[2] = \{0,3\} \\ \hat{x}_1[3] = \text{empty} \\ \hat{x}_1[4] = \{1,2,5\} \end{array} \qquad \begin{array}{l} \hat{x}_2[0] = \{2\} \\ \hat{x}_2[1] = \{0,4\} \\ \hat{x}_2[2] = \{4\} \\ \hat{x}_2[3] = \{1,2\} \\ \hat{x}_2[4] = \{0\} \\ \hat{x}_2[5] = \{1,4\} \end{array}$$

We have now doubled the number of elements that were in our compact version of $\hat{A}$. That puts the number of values needed to store A at 47,104, but this is still an approximately 99.79% decrease from

the brute-force method. All of the necessary information is still in the new data structure, but we have something that takes up little space and can be processed very quickly.

CHECK NODE MATRIX

The check node matrix η holds vital information for each iteration of the decoding algorithm. We can see in (3) that the only values in η we operate on are those that correspond to the locations of the 1's within A. Knowing this, we can start with our efficient representation of A as a base for η.

Examining (3), we notice that each value in η is the result of the $\tanh^{-1}(\cdot)$ operation multiplied by a scalar, and therefore must be a floating point number. Combining that with our knowledge of the relationship between η and A, we can create a single data structure containing all the important code information. If every value within our array representation of A is paired with a floating point value, we then have a representation for η. For instance, let's reuse our example parity check matrix $\hat{A}$ to show this new structure. If $\hat{y}_1$ and $\hat{y}_2$ are arrays representing $\hat{\eta}$, and if $\hat{\eta}_{m,n}$ is the floating point value in $\hat{\eta}$ at (m,n), then

$$\hat{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \implies \begin{array}{l} \hat{y}_1[0] = \{(1, \hat{\eta}_{0,1}), (4, \hat{\eta}_{0,4})\} \\ \hat{y}_1[1] = \{(3, \hat{\eta}_{1,3}), (5, \hat{\eta}_{1,5})\} \\ \hat{y}_1[2] = \{(0, \hat{\eta}_{2,0}), (3, \hat{\eta}_{2,3})\} \\ \hat{y}_1[3] = \text{empty} \\ \hat{y}_1[4] = \{(1, \hat{\eta}_{4,1}), (2, \hat{\eta}_{4,2}), (5, \hat{\eta}_{4,5})\} \end{array} \quad \begin{array}{l} \hat{y}_2[0] = \{(2, \hat{\eta}_{2,0})\} \\ \hat{y}_2[1] = \{(0, \hat{\eta}_{0,1}), (4, \hat{\eta}_{4,1})\} \\ \hat{y}_2[2] = \{(4, \hat{\eta}_{4,2})\} \\ \hat{y}_2[3] = \{(1, \hat{\eta}_{1,3}), (2, \hat{\eta}_{2,3})\} \\ \hat{y}_2[4] = \{(0, \hat{\eta}_{0,4})\} \\ \hat{y}_2[5] = \{(1, \hat{\eta}_{1,5}), (4, \hat{\eta}_{4,5})\} \end{array}$$

Obviously there is some redundancy introduced by storing each floating point value twice. After re-examining (3), the values of η can be updated either row-by-row or column-by-column. Since both methods will give us the exact same result, we can discard one of the sets of floating point values. In this paper, we remove the floating point values from the column representation, leaving our final representation of $\hat{\eta}$ as

$$\begin{array}{l} \hat{y}_1[0] = \{(1, \hat{\eta}_{0,1}), (4, \hat{\eta}_{0,4})\} \\ \hat{y}_1[1] = \{(3, \hat{\eta}_{1,3}), (5, \hat{\eta}_{1,5})\} \\ \hat{y}_1[2] = \{(0, \hat{\eta}_{2,0}), (3, \hat{\eta}_{2,3})\} \\ \hat{y}_1[3] = \text{empty} \\ \hat{y}_1[4] = \{(1, \hat{\eta}_{4,1}), (2, \hat{\eta}_{4,2}), (5, \hat{\eta}_{4,5})\} \end{array} \quad \begin{array}{l} \hat{y}_2[0] = \{2\} \\ \hat{y}_2[1] = \{0, 4\} \\ \hat{y}_2[2] = \{4\} \\ \hat{y}_2[3] = \{1, 2\} \\ \hat{y}_2[4] = \{0\} \\ \hat{y}_2[5] = \{1, 4\} \end{array} \tag{6}$$

This addition of a second array in the data structure of η provides us with all we need to discard A from processing. After the arrays y_1 and y_2 have been initialized from A, we can completely remove A from memory for the current decode operation.

BIT NODE VECTOR

Similarly to the check node matrix η, the bit node vector λ holds valuable information for the decoder. After initialization, λ contains the product of the received vector and a scalar. The received vector is modulated using binary phase-shift keying (BPSK) and therefore contains floating point values. We see in (4) that we are required to store something for every value of n in λ_n. Because of that, we cannot do much to optimize our representation of λ. In this paper we represent λ using an array of floating point numbers, which allows us easy access to elements.

HASKELL IMPLEMENTATION

In this paper, we do not intend to teach Haskell, nor do we provide all of the code used in our model. Our intent is to provide a distant view of the code in order to focus on the architecture of the implementation. The Haskell language itself is described in great detail in [2], as well as numerous websites.

Storing the Check Node Matrix, η

```
data EtaMatrix =
  EtaMatrix (Int,Int) (Array Int [(Int,Float)]) (Array Int [Int])
```

To start, the structure for η contains a pair of integers that hold the number of rows and columns in the matrix A. These values are mainly used for indexing purposes. As mentioned previously, our data structure for η consists of two very large arrays. Every location of each array contains a list, something very trivial to store and process in Haskell.

The first array corresponds to y_1. It is indexed by the rows of η (and therefore A), and each location contains a list of tuples[3]. The first element of the tuple is an integer that holds the column number within η. The second element of the tuple is the actual floating point value we are concerned with for its (row, column) pair in η.

The second array corresponds to y_2. It is indexed by the columns of A, and each location contains a list of integers of row values for each column.

Storing the Bit Node Vector, λ

```
newtype Signal a = Signal (Array Int a)
```

For λ, we use a more general structure. Since it only contains an array of floating point values, we can name a data type called `Signal` that holds an arbitrary type `a`. This allows us to use the same data type for $\hat{\mathbf{c}}$, as it contains an array of integers.

Updating the Check Node Matrix

```
updateEta :: Signal Float -> EtaMatrix -> EtaMatrix
```

Above is the type definition for the function `updateEta`, which is our implementation of (3). The function takes a `Signal Float` ($\lambda^{[l-1]}$) as well as an `EtaMatrix` ($\eta^{[l-1]}$), and returns an `EtaMatrix` ($\eta^{[l]}$). The code for `updateEta` can be found in the Appendix.

To perform the operations shown in (3) in a typical imperative programming language, one would likely use something like nested loops. These loops can become complicated, mostly because there are so many things to keep track of. Haskell removes some of this complexity with its built-in functionality. Lists and tuples are basics in Haskell, and operations such as `map`, `product`, `zip`, and list comprehensions allow easy processing of those basics. Access to $\mathcal{N}$ is made easy by our storage of the information in η as lists.

[3] A tuple is a basic structure in Haskell that can contain multiple types of data.

Updating the Bit Node Vector

```
updateLam :: EtaMatrix -> Signal Float -> Signal Float
```

Above is the type definition for the function `updateLam`, which is our implementation of (4). The function takes an `EtaMatrix` ($\eta^{[l]}$) and a `Signal Float` ($L_c\mathbf{r}$), and returns a `Signal Float` ($\lambda^{[l]}$). The code for `updateLam` can be found in the Appendix.

In this function we make use of the built-in `sum`, as well as list comprehensions. Access to $\mathcal{M}$ is made easy by our storage of the information in η as lists.

Performing the Parity Check

```
eMatMultSigMod2 :: (Num a, Integral a) =>
                   EtaMatrix -> Signal a -> Signal a
```

The above function takes an `EtaMatrix` ($\eta^{[l]}$) and a `Signal a` ($\hat{\mathbf{c}}$), returning a `Signal a` (the product vector, $\mathbf{p}$). The code for `eMatMultSigMod2` can be found in the Appendix.

What we actually need for this function is A instead of η, but since A's information is embedded in η, we can use it. This function represents the main operation in (5), the matrix multiplication (mod 2). Since we are working with a parity check matrix, the operation can be performed using the following steps.

1. For row i in A, find the columns containing 1's.
2. Grab only the locations in $\hat{\mathbf{c}}$ with the results from the previous step as the indexes.
3. Sum up the values from the previous step, apply (mod 2), and store the result in p_i.

The first step is easily accomplished using the y_1 array, the second step is a combination of array lookups, and the third step is done using `sum` and `mod`. Once $\mathbf{p}$ is created, another call to `sum` lets us know if the parity check has passed.

MODEL ACCURACY

In order for us to conclude that our implementation of the decoding algorithm was correct, we compared it to a working model. The model given (built in MATLAB) was simulated over varying values of the signal-to-noise ratio (SNR). In the simulation, the maximum number of decode iterations was set to 200, and a random frame was encoded, mixed with noise (according to the particular SNR), and decoded. This was done 512 times for each SNR value, and the bit errors and frame errors were recorded.

To test our implementation, we ran the same simulation but replaced the MATLAB decoder with our Haskell version. Figure 1 shows the results of the simulation. These results very closely resemble those in [4], which tells us that our decoder works properly.

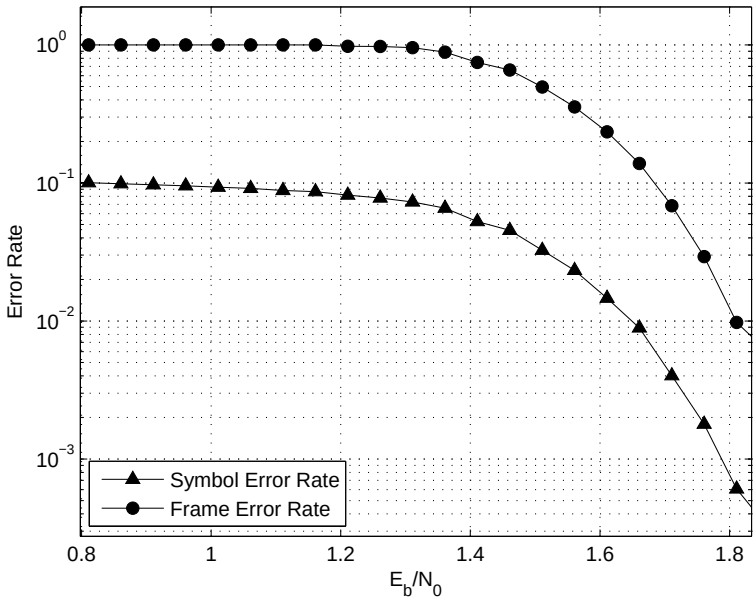

Figure 1: Simulated Performance of LDPC Code with Rate 2/3 and Block Length of 4096

CONCLUSIONS / RELATED WORK

We have successfully modeled an LDPC decoder in the functional programming language Haskell. We explored the capabilities of the language along with its powerful compiler to create a fairly efficient representation. We can now use our Haskell LDPC decoder in an attempt to take a simulation-like representation of a system and generate hardware from it. Along the way, we developed data structures that can be used for this algorithm in a traditional hardware design process. Once we get working hardware implementations from both methods, we can compare the two in order to see if our new method is advantageous.

ACKNOWLEDGEMENTS

The authors would like to thank the Test Resource Management Center (TRMC) Test and Evaluation/Science and Technology (T&E/S&T) Program for their support. This work was funded by the T&E/S&T Program through the U.S. Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI), contract number W900KK-09-C-0018 for High-Rate High-Speed Forward Error Correction Architectures for Aeronautical Telemetry (HFEC).

APPENDIX

A. *Update Function for the Check Node Matrix*

```
updateEta :: Signal Float -> EtaMatrix -> EtaMatrix
updateEta lamPrev etaPrev =
  EtaMatrix
  (r, c)
  (array (0,r-1)
    [(i,(zip (etaPrev &#@ i) (helpUpdate i))) | i <- [0..(r-1)]])
  (array (0,c-1)
    [(i,(etaPrev &#@\ i)) | i <- [0..(c-1)]])
  where
    r = eMatRows etaPrev
    c = eMatCols etaPrev
    helpUpdate m =
      [(-2) * (atanh (product (map tanh (
        [((lamPrev $@ j) - (etaPrev &@@ (m,j))) / (-2)
          | j <- (fNmn etaPrev (m,n))])))))
      | n <- (etaPrev &#@ m)]
```

B. *Update Function for the Bit Node Vector*

```
updateLam :: EtaMatrix -> Signal Float -> Signal Float
updateLam eta lcr =
  Signal
  (array (0,(sigLen lcr)-1)
  [(n, ((lcr $@ n) + (sum [(eta &@@ (m,n)) | m <- (fMn eta n)])))
    | n <- [0..(sigLen lcr)-1]])
```

C. *Multiplication Function for the Parity Check*

```
eMatMultSigMod2 :: (Num a, Integral a) =>
                   EtaMatrix -> Signal a -> Signal a
eMatMultSigMod2 em s =
  Signal
  (array (0,(eMatRows em)-1)
    [(i,(mod (sum (sigAtIdxs s (em &#@ i))) 2))
      | i <- [0..(eMatRows em)-1]])
```

REFERENCES

[1] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. New Jersey: Wiley, 2005.

[2] S. Peyton Jones, ed., *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware Design in Haskell," in *International Conference on Functional Programming*, pp. 174–184, 1998.

[4] Consultive Committee for Space Data Systems (CCSDS), "Low density parity check codes for use in near-Earth and deep space applications (131.1-O-2 Orange Book)," Sep. 2007.