CWP-045

# COMPUTATIONAL ASPECTS OF THREE DIMENSIONAL SEISMIC INVERSION

by

Patrick W. Quist

Partially supported by the Consortium Project of the Center for Wave Phenomena and by the Selected Research Opportunities Program of the Office of Naval Research

Center for Wave Phenomena
Department of Mathematics
Colorado School of Mines
Golden, Colorado 80401
(303)273-3557

TABLE OF CONTENTS

## ABSTRACT

Very briefly, data inversion is: given the recorded data, deciding what caused the data. For the seismic industry, inversion is the process of taking data which varies with time and converting it into data which varies with depth. After inversion, the peaks on a trace provide the depth at which a change occurs in the subsurface while the amplitude of the peaks measure the magnitude of that change.

The Center for Wave Phenomena has developed a computer program to invert three dimensional seismic data. Inverting seismic data takes enormous amounts of CPU time and "core" memory. Except for small test cases, this type of work is impractical on a machine such as the CSM Gould 9750 or VAX 8600. For my thesis, I ported the CWP inversion code to a Cray X-MP computer in Minneapolis for testing and optimization. A Cray computer has both a very fast central processor and a large memory, so problems of this type can be successfully solved. Even with the Cray's capabilities, though, the code should be efficiently designed and implemented.

Another aspect of this thesis is bringing the inverted data back to Colorado School of Mines to either

# LIST OF FIGURES

## ACKNOWLEDGEMENT

The author gratefully acknowledges the support of the Office of Naval Research, Mathematics Division, through its Selected Research Opportunity Program, and the Consortium Project on Seismic Inverse Methods for Complex Structures at the Center for Wave Phenomena, Colorado School of Mines. Consortium members are Amoco Production Company, Conoco, Inc., Geophysical Exploration Company of Norway A/S, Marathon Oil Company, Mobil Research and Development Corp., Phillips Petroleum Company, Sun Exploration and Research, Texaco USA, Union Oil Company of California, and Western Geophysical.

this code is nearly optimum. It is only nearly optimum, since some time could be saved by moving a square root call, but that is more of a production worry than an academic worry.

Chapter 4 will change focus from the inversion code to returning the data back to Colorado School of Mines. The inverted data set may be quite large, so some data compactification schemes are examined. To this end, a few basic information theory concepts will be presented. The data set size can be significantly reduced which will directly decrease the transfer cost.

parallel (or even straight), and the number of traces per line may vary from line to line. However, the number of time samples per trace will usually be constant within a survey, as it will be in this paper.

Geologic data, such as cross sections or borehole data, usually varies with depth, whereas seismic data varies with time, making it difficult to correlate the two. Seismic data can be converted to depth data through the process of inversion (or migration). The purpose of inversion is to replace the time dependence of the input data with a depth dependence. After inversion, the position of the peaks on a trace indicates at what depth there is a change in some measurement of interest (typically propagation speed) while the amplitude of the peaks relates to the magnitude of that change.

One of the major assumptions connected with seismic data acquisition is that all of the recorded energy came from directly below the recording line. This assumption does not allow for off-line contributions which arise in an inhomogenous world. One of the aims of three dimensional inversion is to sort the input data in such a way that off-line contributions are accounted for and their information used.

a series of layers with jump discontinuities in the velocity (or impedance) at these layers." (Cohen and Hagin, 1985)

The discrete version of formula (1) is

$$\beta(x) \simeq C_1 \Sigma_{k=1}^{K} \Sigma_{j=1}^{J} A(\rho_{jk}, z) W(\xi_j, \eta_k, \tau(x, \xi_j, \eta_k)),$$

where $C_1$ is a constant arising from the discretization procedure, k = 1, ... ,K (K = number of input lines), and j = 1, ... ,J (J = the number of traces per input line). K and J can each be of the order n, so the summation is of order $n^2$. This summation is repeated for each output point, x. Since x = (x,y,z) and each of these directions can be of the order n, there are $n^3$ output points. This yields an operations count of $n^5$. This operations count is an easily obtainable upper bound. Generally, the number of input points will be less than $n^2$, so the operations count will be less than $n^5$.

## CZ3D2 Structure

The inversion program is called CZ3D2, and was written by B. Sumner and Drs. Hagin and Cohen. It has an accessory program worth mentioning, which was written by B. Sumner. The program is CZ3D1 and it filters the

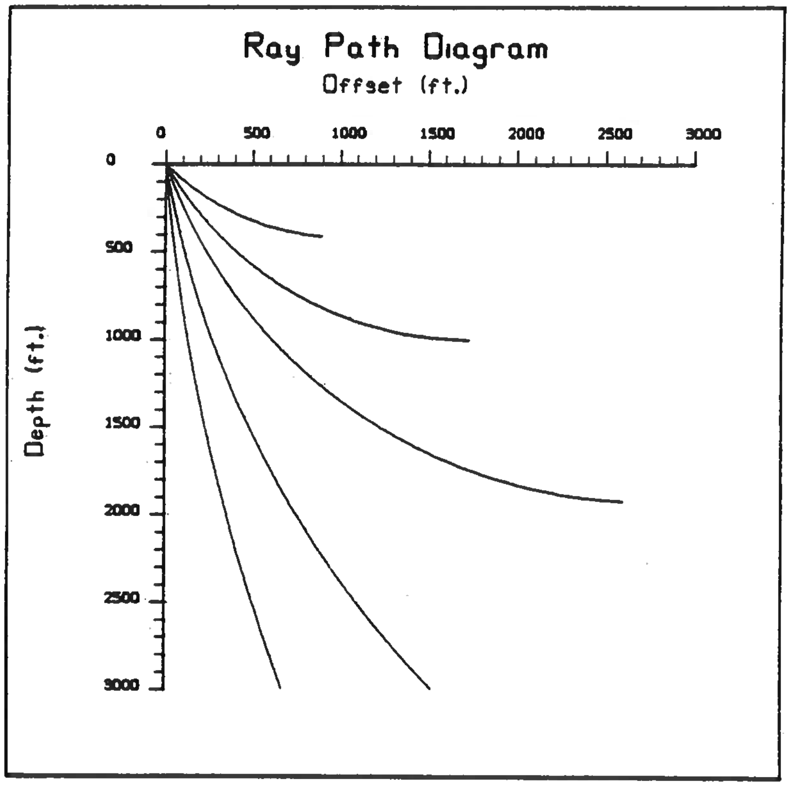

Figure 1.1
Ray path diagram

The values, $\tau$ and A, are tabled in an array which is referenced by depth and horizontal offset. The maximum offset possible for each depth is an important quantity, as it will determine loop indices in the second section of the program. The graph below (Fig. 1.2) illustrates the basic character of how maximum offset varies with depth. The graph will rise to a peak since the longer the energy travels, the further it can travel. The tailing off is due causality, since the recording lasts for a finite that the integration requires 5 nested loops. The order of computation is:

```
Loop on input lines
   Read the input line

   Loop on output lines
      Read and/or write output line

      Loop on depth
         Read correct table record

         Loop on the output traces

            Loop on the input traces
               sum
            end loop

         end loop

      end loop

   end loop

end loop.
```

When the program was written, care was taken to minimize the number of I/O operations. During any run, there are four data files which require either reading, writing, or both. The first filed used is the input data. Since this data typically will be stored on a magnetic tape, the authors wanted to cycle through it only once. Hence the leading loop of the 5 loops cycles though the input lines, reading the information for one line into a two dimensional array.

The third file necessary for the inversion is the table of travel times and amplitudes calculated in the top section. This table is referenced by depth and offset, with records within the file containing the information for one or more depths. The third loop, which iterates on the depth variable, reads a new record from this file if the current depth is not covered by the information previously read. This information is stored in a one dimensional array, which is then referenced by offset.

The final file is the output data. Like the input data, it will typically be stored on magnetic tape. This file is written after the computation is complete, so it does not affect the loop structure.

The only loops not yet mentioned are the two innermost loops, which are also the most important of the five, using over 90% of the run time. These loops sum along the the traces of both the input and output lines. See the diagram below. For this diagram, the input line, output line, depth, and output trace are all fixed. The current output trace is at the center of the circle. The depth determines the maximum range of input (the radius of the circle), hence the integration range of the input line (points a and b) may be calculated. The deepest loop then steps across the input line from a to b, calculating

## Input Model Data

This should be enough of an overview of CZ3D2 to acquaint the reader with the program layout, so the next issue is the data used throughout this thesis. No field data was used, but rather, computer generated data was used. The generating program was written by B. Sumner and is named SYNTH1. That program calculates the seismic data for one line based upon an input model. That line is then replicated to simulate three dimensional data.

Two models are used for tests in this paper. The first is a single horizontal reflector at 1000 ft with a 1000 ft/sec jump in propagation velocity across that reflector, while the second has three dipping reflectors with various jumps in the propagation speed. See Figure 1.5.

This one diagram depicts both models. The single plane model is the same as the three dipping planes model except that the second and third reflectors are not present. On this diagram, the horizontal and vertical scales are not equal so the dips are exaggerated. The propagation speed and density of each layer is labeled within the layers. The panel to the right shows the input reference velocity which CZ3D2 used to calculate the travel times and amplitudes for the three dipping planes model. Notice that the reference velocity matches the model near Trace 1, but by Trace 100, the reference velocity does not match. Where the reference velocity is a poor estimate of the actual velocity, the result produced by CZ3D2 will not be as good.

At the bottom of the diagram, two data sets are listed, identifying the input grid and output window for each set. They are named hugeinv and biginv for the relative size of the output data file at the time which they were first run. The hugeinv set will be used in the Cray results section and the data transfer section. For the Cray tests, two varations of hugeinv will be used: a 10 line and one line output window. The biginv data set will be mentioned in the data transfer section only.

operate at 6.67 Mips. The Cray X-MP has a cycle time of 9.5 nanoseconds, resulting in a maximum instruction rate of 105 Mips. This fast clock requires specially designed chips and cooling, each of which adds to the cost of a Cray computer. Also, short wire lengths between components is necessary to lessen the transfer time between the components. The short connections lead to the aesthetically pleasing circular shape of Cray computers.

## Functional Units

The Cray X-MP has various configurations. It is available with 1,2, or 4 processors and with 1,2,4,8, or 16 million words of memory. The various combinations are denoted by X-MP/ab where 'a' is the number of processors and 'b' is the memory size. For example, this project was completed on an X-MP/48. Though the four processors were not used for this project, a future project could incorporate the multi-processing power to reduce the overall run time of the inversion program.

Each of the processors has 12 functional units, and of these 12, only the 3 floating point units will be discussed. These units do the following operations: addition, multiplication, and reciprocal approximations. The other functional units have similar design, but they

$28.5\text{x}10^{-4}$ seconds. In vector mode, a result is obtained every clock period after the 6 cp necessary for the first pair to pass through the functional unit. This is called start-up time. The same vectors then require only 50006 cp, or $4.75\text{x}10^{-4}$ sec. Ignoring the start-up time, this is precisely 6 times faster than the scalar mode, and that can be generalized for any functional unit. The number of segments in a functional unit is equal to the start-up time of that unit and will decrease the amount of time needed for that operation by the same factor. The functional unit times for the Cray X-MP floating point functional units is listed below.

| | |
|---|---|
| Addition | 6 cp |
| Multiplication | 7 cp |
| Reciprocal approximation | 14 cp |

A functional unit may pass its results onto a different unit rather than returning them to the registers. This is called chaining and like segmentation, increases the effective speed of a Cray computer. Each of the units can operate at a maximum rate of 1 operation per clock period, or 105 Mflops. However, when the results are chained the effective speed can increase to 210 or even 315 Mflops, since each of the units in the chain produce one operation per clock period.

| | Before SSD Assignment (sec) | With 1 file on SSD (sec) |
|---|---|---|
| TIME EXECUTING IN CPU - | 53.1329 | 52.8161 |
| TIME WAITING TO EXECUTE - | 136.9485 | 137.8345 |
| TIME WAITING FOR I/O - | 567.0504 | 8.8585 |

Table 2.1
SSD I/O wait time improvement of 70 to 1

Note that the the CPU time and execution wait time are comparable, but that the I/O wait time dropped dramatically, from 567 sec down to 9 sec. This example is extreme, but fairly representative of SSD capabilities. It should noted that I/O wait time is highly dependent upon the system load at the run time and these figures will vary. Also, the SSD is a resource which must be allocated, so on occasions the job sat in a queue waiting for available space.

## Theoretical Rates

Before moving on to how well CZ3D2 performed on the Cray, it is interesting to see some example operating rates on a tightly controlled program. This next section will present results of simple vector operations obtained under dedicated time on the Cray. It also will introduce a loops may not be significant, these seven loops were nested in one outer loop with an iteration count of 100. The first call allows the user to name the loop, so the name of each operation is listed under the Name column. Because the utility itself has an overhead of 300 - 600 clock periods, so the number of clock periods spent within the loop should be long compared to 600. For the test, each of the vectors was dimensioned to 50,000, thus easily passing the clock period constraint.

For the ease of reading, the FLOP results have been broken into three tables. Table 2.3 shows the loop name, number of passes through that loop, and the timing information for each loop. Table 2.4 presents the number floating point operations for each the loops; and Table 2.5 lists the operating rates of each of the loops.

These tables have a few striking features which will be discussed in the next few paragraphs. First, the mega-flop rates in Table 2.5 are somewhat slower than expected. Second, a single square root requires 15 operations to obtain that result. Also, the minimum function and conditional were slower than expected.

For the moment, concentrate on the F_ADD entry of Table 2.3 and recall the operation time estimated for that operation in the functional units section ($4.75^{-4}$ sec). This estimated time was the same operation as done by F_ADD, except for the 100 passes through the F_ADD loop, so multiplying that number by 100 makes the times comparable. The time obtained on the Cray is $6.36 \times 10^{-2}$ sec, which is considerably longer than the expected time of $4.75 \times 10^{-2}$. The difference lies in that our calculated time ignores the time needed to reference the vectors. The FLOP utility, however, has that time available to it, so it uses the total time to calculate its mega-flop rate. That is the correct way to calculate the rate since preparation time should be included in the overall rate.

It also explains why the mega-flop rates in Table 2.5 are somewhat lower than expected. The author expected rates closer to the maximum of 105 Mflops for the floating addition and multiplication and closer to 210 Mflops for root and that loop is the most time consuming part of the inversion.

Note that the two square root loops have the highest rate of results, yet they took the most time. Recall from Table 2.4 that a square root requires 15 operations, whereas the other loops have only one or two operations per pass. The high number of operations raises the speed of obtaining a square root, but it is still a time consuming result.

produced the following results on the Gould and Cray.

```
 Line no.:   40 Trace no.:   40
   Z        Beta            Delta C               |
 ------------------------------------------------------------
  950.0 -0.2363918E-01  -0.2309326E+03   *  |
  962.5 -0.4238997E-02  -0.4221103E+02      *
  975.0  0.3477888E-01   0.3603203E+03      |****
  987.5  0.7448769E-01   0.8048262E+03      |*********
 1000.0  0.9243470E-01   0.1018491E+04      |***********
 1012.5  0.7705796E-01   0.8349163E+03      |*********
 1025.0  0.3677832E-01   0.3818259E+03      |****
 1037.5 -0.6571915E-02  -0.6529008E+02     *|
 1050.0 -0.3130101E-01  -0.3035098E+03 *    |
```

Table 3.1
Gould Inversion Results

```
 Line no.:   40 Trace no.:   40
   Z        Beta            Delta C               |
 ------------------------------------------------------------
  950.0 -0.2363822E-01  -0.2309236E+03   *  |
  962.5 -0.4238809E-02  -0.4220917E+02      *
  975.0  0.3477962E-01   0.3603283E+03      |****
  987.5  0.7448657E-01   0.8048135E+03      |*********
 1000.0  0.9243472E-01   0.1018491E+04      |***********
 1012.5  0.7705859E-01   0.8349240E+03      |*********
 1025.0  0.3677790E-01   0.3818216E+03      |****
 1037.5 -0.6572199E-02  -0.6529287E+02     *|
 1050.0 -0.3130036E-01  -0.3035038E+03 *    |
```

Table 3.2
Cray Inversion Results

These tables are produced by an accessory program to CZ3D2 named CZ3D3 which was written by B. Sumner. The line and trace number of the output are identified at the bin, whether it is a user statement label or a compiler generated label.

## SPY Results

Since SPY relies on statistics, it should be used only with large runs, so the following table was produced by SPY using the hugeinv output window. The table was edited to show the first few lines of the printout, the innermost loops of the integration, and the subroutines.

Referring to Table 3.3, the first three labels are compiler generated labels, while the labels suffixed with -A or -B are the tops and bottoms of do loops in the program. For instance, the 420A label is the top of the depth loop while the 420B is the bottom of that loop. Also, not all of the user statement labels appear in the printout. For example, the 410A label has no 410B counterpart because in the CZ3D2 code, the 410 continue statement directly follows the 400 label, so the 410B label would be included in the 400B bin. The HITS column is just the total number of times that SPY found the program control in that bin. Similarly, the SECONDS column is the number of seconds spent in the bin. The column headed by %SUB lists the percent of the run used for that module alone, while the column headed by %PRG applies to the total run time. The %CUM stands for cumulative percent and is just the running sum of the %PRG column.

The part to note from this table is that the bins containing the labels 400A and 400B use 64.6% of the program run. These labels pertain to the innermost loop of CZ3D2. If the bin named 410A is included with those two then the percentage of the run spent in these bins increases to 72.7% and the section of the program included is increased to the two innermost loops. Since these loops dominate the run, they are the most likely candidates for any optimization.

Also notice that the cumulative percentage column does not sum to 100%. This is because SPY distinguishes between program and system control. In this case, CZ3D2 accounted for 73.9% of the run time and system calls comprised the other 26.1%. See Table 3.4.

In the above table, the system calls are listed along with CZ3D2 and its subroutines. Some FORTRAN intrinsic function such as SQRT(), COS(), and ALOG() are not counted with the program run, but have their own entries in the table. The system calls begin with a $. Notice that the subroutines, listed at the top of the table beneath CZ3D2, required virtually no time. Part of this is due to the wide time slice that SPY uses (500 microseconds), but mostly, the subroutines are a minor part of the program run time. It is not worth the effort to attempt any optimization within these modules, because any speedup in them would not appreciably affect the total run time.

Looking again at the table, most of the system routines use little or no time to execute. The exceptions to this are the square root operation and the I/O routines $RCW, $RU, $WRTUTIL, and $WU, none of which can be avoided. The square root is necessary for the integration, so little can be done about it. Also, the square root call is suffixed by -V, indicating that it is a vector square root, providing the best performance available.

Overall, SPY provided a good profile of the program and helped narrow the optimization focus. The SPY utility showed that the two innermost loops dominate the run consideration is effectively utilizing the Cray hardware. Of course, few if any programs will efficiently take advantage of all of the vector capabilities. The user must then decide whether the rate FLOP provides is acceptable given the mathematical constraints within the program.

The initial test of FLOP on CZ3D2 used the three dipping planes model with 10 output lines, 100 traces per line, and 239 depth points (the hugeinv output window). The FLOP results for this model and output window follow (edited for easier reading).

The first result to notice from this table is the dominance of CZ3D2 in comparison to the subroutines. CZ3D2 accounts for essentially 100% of the run time and floating point operations. This result agrees with the SPY result that the subroutines are not worth optimizing, even though they run relatively slow (7.95 to 12.8 Mflops). The 80 mega-flop rate is acceptable, but not outstanding.

SPY indicated that the two innermost loops accounted for most of the run time, so having a calculation rate for those loops became important. Unfortunately, the FLOP utility bogged down when isolating the loops with ten output lines, so the output window was cut to one output line, 100 traces 239 depth points on each trace. The one output line FLOP results without the loop isolation follow. The FLOP results with the loops isolated will be presented later.

floating point operations also dropped, but again not quite by the full factor of ten. The drop was from $2.25 \times 10^{10}$ to $2.40 \times 10^{9}$ floating point operations. The mega-flop rate for the two runs was comparable, 80.63 to 82.58. Keep in mind that FLOP is a statistical utility, so that the differences mentioned could be either within the FLOP error margin, or due to the variance of the load on the Cray.

Compare the results for the subroutines. Notice that they do not exhibit the reduction of the output window. The run times are comparable, the operations count is the same, and the rates are roughly the same for each subroutine. That is because they are affected more by the input reference velocity than the width of the output window and the model was held constant for the comparison. Had the reference velocity or the depth window changed, then the number of operations and the run time for each of the subroutines would have changed, but these were held constant. The percentage of the run attributed to the subroutines increases for the one line of output, but they are still negligible. Henceforth, the subroutine entries will be edited out of the FLOP results, as they do not add any new information. Also, the one line output window will be used for the rest of the FLOP results.

and the program as a whole. This indicates that any optimization efforts should concentrate only on these loops.

A few attempts were made to optimize these loops. First, switching the loops was examined and found to be a detriment to the execution time. The other attempts tried to avoid a conditional within the innermost loop. These tries neither helped nor hurt the performance of CZ3D2. The next few paragraphs will explain each of these results.

Currently, an output trace (the outer of the two loops) selects all the input traces which sum into it. Switching loops would mean an input trace summing into all of the output traces for which it affects. In either case, the innermost loop will have loop indices which depend upon the other loop. That being the situation, the innermost loop will iterate at least once and at most 78 times. (The upper bound is particular to the model used. Generally, the maximum number of iterations occurs when the input and output lines are coincident. Then the maximum number of iterations is equal to:

$$2 * \text{INT} [ \text{MAX}(\text{range}(z)) / \text{MIN}(dx, dy) ],$$

where range(z) is an array containing the maximum range of

| NAME | CALLED | TIME(SEC) | AVE-TIME | %AGE | ACCUM% |
|---|---|---|---|---|---|
| ORIGLOOP | 11873 | 2.62E+01 | 2.21E-03 | 92.62 | 92.62 |
| DEEP2 | 11873 | 3.13E+01 | 2.64E-03 | 93.79 | 93.79 |
| NOIF_MIN | 11873 | 2.63E+01 | 2.21E-03 | 92.46 | 92.46 |
| TRUNC | 11873 | 2.61E+01 | 2.20E-03 | 92.32 | 92.32 |

| NAME | ADDS | MULTS | RECIPS | FLOPS |
|---|---|---|---|---|
| ORIGLOOP | 1.16E+09 | 1.04E+09 | 1.82E+08 | 2.39E+09 |
| DEEP2 | 1.16E+09 | 1.04E+09 | 1.82E+08 | 2.39E+09 |
| NOIF_MIN | 1.16E+09 | 9.95E+08 | 1.81E+08 | 2.34E+09 |
| TRUNC | 1.16E+09 | 9.95E+08 | 1.81E+08 | 2.34E+09 |

| NAME | MEM/FLOP | MMEM/SEC | MFLOPS |
|---|---|---|---|
| ORIGLOOP | 0.27 | 24.97 | 91.07 |
| DEEP2 | 0.47 | 35.57 | 76.26 |
| NOIF_MIN | 0.28 | 24.91 | 89.00 |
| TRUNC | 0.28 | 25.09 | 89.65 |

Table 3.8
Composite of Four FLOP Runs

The other attempts at optimization were not as drastic, but they had about as much success. First note that number of multiplications and reciprocals dropped slightly for the NOIF_MIN and TRUNC entries. That is because a division was replaced by a multiplication of its reciprocal within the innermost loop. One division counts as 3 floating point multiplications and one reciprocal, so both counts should drop.

The rate for each of these attempts is close to the original rate of 91 mega-flops. These changes were to how the horizontal offset is calculated. In calculating the geometry set up by the c(z) assumption. The present code utilizes that geometry when tabling the travel times and amplitudes, but does not for the inversion. The seismic data structure is outwardly rectangular, so any cylindrical symmetry within it is well masked. The rest of this thesis will concentrate on transferring the data from a remote computing facility to the users location.

affect the graphing of the data. The "compression" phase uses a Huffman algorithm to encode the data in such a way that the file size is smaller. Huffman encoding preserves the information content, but then the data cannot be used until it is decoded. Each of the methods can be illuminated by information theory; so some concepts from that theory will follow.

## INFORMATION THEORY

Information theory defines quantities such as information, entropy, and redundancy of a message. An alphabet is the group of symbols, or possible symbols within a message. Let M be the number of elements in the alphabet. For example, the letters A, B, C, D, and E comprise one alphabet (with $M = 5$) while the numbers 1-9 another. A meta-symbol is one or more of the symbols combined. The probability associated with a meta-symbol is the sum of the individual symbol probabilities and will be interpreted as the probability of one of the individual symbols occurring next. For this paper, the alphabet is the 256 combinations of 8 bits. Note that each symbol in an alphabet can itself be considered a message and will have a probability of occurrence associated with it.

The information of a symbol, then, is defined to be

$$I = -\log_2 P(i),$$

where P(i) is the probability of the $i^{th}$ symbol of the alphabet occurring next in a given message. The negative logarithm function was picked because it has the value 0.0 at 1.0 and increases without bound as the argument approaches zero, while the base 2 was chosen so that the information of a symbol is measured in bits.

While information is measured for individual symbols, the entropy applies to the entire message. Entropy is just the average information content of the message, or

$$H = -\Sigma_{i=1}^{M} \quad P(i)\log_2 P(i).$$

In this formula, the sum runs over all of the symbols in the alphabet. Like information, entropy is measured in bits, so entropy is also the average number of bits which have useful information.

Another quantity, $H_{max}$, is needed before redundancy can be defined. $H_{max}$, the maximum entropy, is a least upper bound on the number of bits needed to encode M symbols. Mathematically, this is simply

$$H_{max} = \log_2 M.$$

to 8 bit integer data. That operation is followed by packing 4 data points into one word of computer memory. This phase shows that the above-mentioned classification is not perfect. During the truncation, there is a loss of information, so it is clearly reducing the entropy of the data. However, the packing of 4 data points into one word of memory reorganizes the data into a more compact form, so it also reduces the redundancy of the data. Both traits exist in this phase. Overall though, the loss of information dominates the phase so it falls into the entropy reduction category.

The 8 bit data format was decided upon for two reasons. First, 8 bits is equivalent to slightly more than 2 significant figures, which is typically about the accuracy of the inversion algorithm and of the original time data. Secondly, a plot of the 8 bit data does not look any different than a plot of 16 bit or 32 bit data. See the following figure. That plot shows the same 5 traces of inverted data, but at various levels of accuracy. The 2 bit and 4 bit displays exhibit clipping and a lack of resolution between the peaks. The 8 bit to 32 bit displays, though, are essentially identical. The human eye would be hard pressed to distinguish between the displays, particularly when most seismic traces are

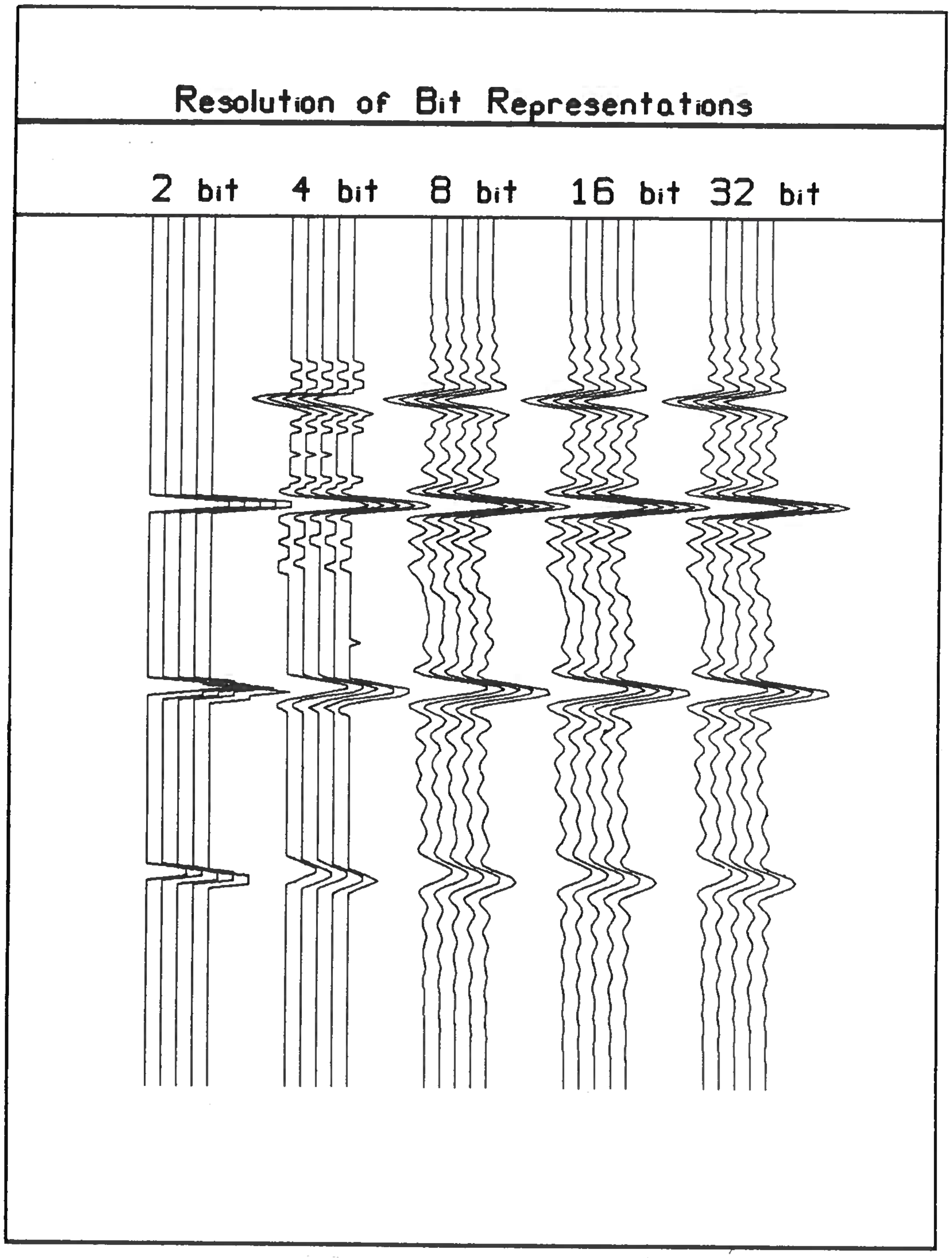

Figure 4.2
Resolution of seismic data at various accuracies

The Huffman method builds a code according to the probability of a character occurring (Huffman, 1950). For the next few paragraphs, consider a hypothetical message of 50 characters composed only of the characters A, B, C, D, and E. Let them have the frequencies and associated probabilities listed below.

| Letter | Frequency | Probability |
|---|---|---|
| A | 20 | 0.40 |
| B | 12 | 0.24 |
| C | 10 | 0.20 |
| D | 6 | 0.12 |
| E | 2 | 0.04 |
|  | 50 | 1.00 totals |

Table 4.1
Hypothetical message character distribution

A code for this distribution could be built as follows. Find the characters with the two lowest probabilities (D and E here). Assign the character with the lower of the two probabilities to be 0 and the greater to be 1. Add their probabilities to get a probability of either of them occurring, thus creating a meta-character. Now, search the amended table for the next two lowest probabilities (C and meta-character DE) assigning 0 and 1 as before. Continue this process until the table has only one entry. See Figure 4.3.

400 bits (roughly a four to one compression). Note that this code is not unique. In building it, the lower of two probabilities was assigned 0. The rule might just have easily been the lower probability receives a 1. That rule would have have resulted in an equivalent, but different code.

A message is decoded as it is received, provided that the key is present. The first bit is received, and tested whether it is a 0 or 1 . If it is a 0, then the first character of the message is an A. Otherwise it is the meta-character BCDE. The next bit is examined. Again, if it is a 0, the character is known (a B). Continue examining bits until a character is determined. Then start over until the whole message is deciphered.

The above example introduces the basic method. That algorithm was developed in 1952 and there have been some improvements since. The current approach finds common substrings in the file, then finds an optimal code for the substring. Also, the algorithm codes in blocks, so that the new codes may change within a file, thus keeping the codes optimal over a shorter range. One other improvement over the original is that the code key does not need to be stored.

$$H_{max} = 8.000$$

$$H = 5.426$$

$$R = 2.574.$$

It should be noted that the entropy, H, has dropped while the redundancy, R, has increased. Sending the packed data through the Huffman algorithm resulted in a further 2.5 to 1 compression, for a complete compaction of 10 to 1. The following table lists the size, in bytes, of the data set at each stage of the compression.

| | original | packed | compressed |
|---|---|---|---|
| Seismic data | 964000* | 240000 | 95387 |
| Header information | 1972 | 1976 | 1039 |
| Total | 965972 | 241976 | 96426 |

*This figure includes record separators between traces (1000 traces)(8 bytes/trace) = 8000 bytes

Table 4.2
Sizes (in bytes) of the Hugeinv Data Set

The original data is in one file even though the chart indicates two files. During the packing process the seismic data is split away from the header information so that more compression may be achieved during the Huffman

The Huffman algorithm can compress the packed data but not the raw data because of the different distributions of the characters within each of the files. The Huffman routine reads the input data one byte at a time, so when the 32 bit floating point data is sent through the routine, no patterns emerge. Taking 8 bits from a floating point number results in a very flat distribution. See the figure below. The two peaks near 100 and 300 octal respectively, are due to the way floating point numbers are stored on the Gould and to the restricted range of the seismic data. The leading bit is a signbit and since our data takes on both negative and positive values, two peaks are possible. Following the signbit are 7 bits that are used for the exponent of the floating point number. The seismic data is scaled to be in the range [-1.0, 1.0], thus limiting the the possible exponents to just a few. Even though these peaks are present, they are not large enough for the Huffman algorithm to take advantage of them.

Seismic data, though, has a distribution centered about zero. See Figure 4.4. Most of the seismic data lies between $151_8$ and $214_8$. There is data outside of this range, but it is very infrequent. Truncating seismic data to eight bits preserves the distribution of that data for data set. That data set is the three dipping planes model with an output window of 10 output lines, 100 traces per line, and 239 points per trace. The run time of each routine is the sum of the user seconds and the system seconds, while the real time is the elapsed time for the computer to run the program.

| | User time (sec) | System time (sec) | Real time (sec) |
|---|---|---|---|
| Packing of data | 15.7 | 2.2 | 20 |
| Huffman encoding | 3.4 | 0.6 | 4 |
| Huffman decoding | 2.6 | 0.5 | 8 |
| Unpacking of data | 8.9 | 1.6 | 11 |
| Total time | 30.6 | 4.9 | 43 |

Table 4.4
Run time chart for the compression
(Gould 9750)

No attempt was made to optimize the run time of the packing and unpacking routines since they will be rarely used. Also, the savings already made possible by the programs far outweighs any possible benefits of optimizing them. The Huffman routines, however, are more general and will be used much more extensively, so somebody has optimized them. The total compression time, then, is the sum of all four routines (43 sec). That number should be compared to the anticipated savings in transmission time

# CONCLUSION

The authors of CZ3D2, B. Sumner and Drs. Cohen and Hagin wrote a good program which performed as they expected. It ported from the Gould 9750 here at CSM to a Cray X-MP/48 in Minneapolis quite easily. For a while, the I/O wait time presented a problem, as it took up to ten times longer than the actual run time, but taking advantage of a high speed storage unit solved that problem. The device is called a Solid-state Storage Device and it reduced the I/O wait time to virtually no time.

The operating rate of CZ3D2 turned out to be in the 80 mega-flops. After testing the program, and trying a few changes, that rate was found to be acceptable. Particularly when contrasted with the test results presented in Chapter 2. Those results were obtained using long vector lengths (50,000 elements) and under dedicated time, yet the floating point addition did not obtain the 80 mega-flops that CZ3D2 did. CZ3D2 did not have the advantage of long vectors, as its inner loop had a minimum iteration of 1 and a maximum iteration of 78. CZ3D2 did have very few memory references per floating point operation (0.29), which helps its calculation rate.

By truncating the output data of CZ3D2 from 32 bit floating point data to 8 bit integer data, a great savings in transmission time can be achieved. Following the truncation, the packed data should be run through a Huffman encoding algorithm. This two stage compactification yields a data file size reduction of approximately 10 to 1. It is only approximate because the Huffman algorithm is highly dependent upon the input data.

This compactification has some drawbacks to it, however. The phone transfer may cause some problems. If even one bit is changed, the Huffman algorithm will not be able to properly decode the data. The phone transfers which were done utilized the local lines only. For those transfers, no problems occurred, but the long distance lines have more background noise which may cause problems. A future project may examine this. Also, modems with error checking exist, so they should be used when files are transferred.

## Appendix A

## TEST PROGRAM for FLOP

This is the program which was tested in the Theoretical Rates section. It was run under dedicated time, thus eliminating system conflicts with other programs on the Cray. The idea was to see the rates of various common operations. The operations are:

| Operation | Name |
|---|---|
| A(j) = B(j) + C(j) | F_ADD |
| A(j) = B(j) * C(j) | F_MULT |
| A(j) = B(j) * C(j) + const. | F_MULT_S |
| A(j) = sqrt( C(j) ) | BAS_SQRT |
| A(j) = sqrt( B(j) * C(j) ) | CHNDSQRT |
| A(j) = MIN( B(j), C(j) ) | MIN_FUNC |
| if ( B(j) .LT. C(j) ) then A(j) = B(j) | IF_TEST |

The FLOP routine allows the user to name the isolated section, so the name column identifies the name I gave each loop. 'F_ADD' is a floating point addition, 'F_MULT' is a floating point multiplication, 'F_MULT_S' is the multiplication followed by an addition of a scalar. The two square roots are 'BAS_SQRT' for a basic square root and 'CHNDSQRT' for a multiplication chaining into a square root. The other two names should be self-explanatory. The program follows.

```
          CALL PERFON('CHNDSQRT'R)
          DO 500 J = ONE, VECLEN
             A(J) = SQRT( B(J) * C(J) )
500       CONTINUE
          CALL PERFOFF

          CALL PERFON('BAS_SQRT'R)
          DO 600 J = ONE, VECLEN
             A(J) = SQRT( C(J) )
600       CONTINUE
          CALL PERFOFF

          CALL PERFON('MIN_FUNC'R)
          DO 700 J = ONE, VECLEN
             A(J) = MIN( B(J) , C(J) )
700       CONTINUE
          CALL PERFOFF

          CALL PERFON('IF_TEST'R)
          DO 800 J = ONE, VECLEN
             IF (B(J) .LT. C(J) ) A(J) = B(J)
800       CONTINUE
          CALL PERFOFF

900    CONTINUE

       STOP
       END
```

```
00000000 00000000 00000000 10010111
```

The other data points may be packed into that word by shift and bit-wise OR operations. First the data point is shifted to the left, filling the vacant field on the right with zeroes.

```
00000000 00000000 10010111 00000000
```

Then a bit-wise OR operation puts an unshifted data point into the rightmost field. This procedure will work as long as the data points do not overlap and as long as zeroes, rather than ones, fill the fields which do not contain data points. The program takes care of these considerations by 1) using only positive values for the 8 bit data, and 2) shifting the data before the OR operation.

PKBIT breaks the data into a header file and a packed seismic data file. The header file contains information such as number of lines, traces, depth points, etc. It also contains the background velocity information and the maximum amplitude by which the traces were scaled. The seismic data is in a separate data file so that the Huffman routine may yield more compression.

The unpacking routine works similarly, but it shifts