September, 1998

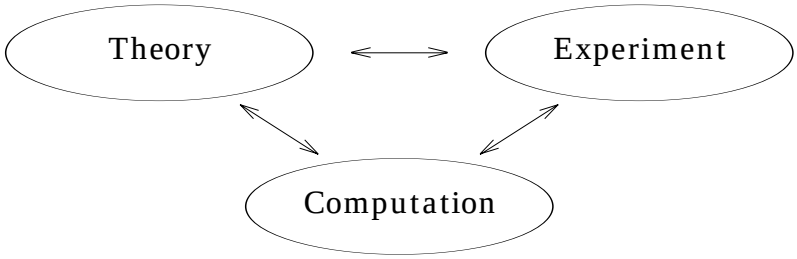
PHY307F/407F - Computational Physics
Course Notes

David Harrison

I. INTRODUCTION

Traditionally physics has been divided into two main branches: *theoretical* and *experimental*. Over time, the interaction between these two branches has become well-defined and the natural tension between theoreticians ("I don't roll around on the floor with a screwdriver in my teeth!") and experimentalists ("I don't spend all my time pushing a pencil around on a piece of paper!") has had a largely positive influence on the advancement of physics.

The effect of computers on the way physics is carried out has been sufficiently profound that some have proposed that the traditional *bifurcation* of the field into theory and experiment has now *trifurcated* into theoretical, experimental, and **computational** physics. Perhaps extreme, the suggestion nonetheless points out the huge impact this technology is having on the field. Pictorially, a trifurcated physics is:



In this course we give an overview of some areas of computation of particular importance to physics. The approach is broad rather than deep. In all cases, the computation is closely tied to real physics problems. The terminology of the course closely follows that of our undergraduate laboratories. Also in common with those labs, there is an emphasis on error analysis throughout.

A recurring theme is that although in principle the computers don't do anything that we can't do by hand, in practice the technology is sufficiently fast the it *feels* to us that we are doing something qualitatively different from what would be possible without the computers. This in turns leads to differences in the kind of questions that we ask about a physical system, and the sorts of approaches that we take in determining the answers to our questions.

In common with many of our laboratories, we begin with an Exercise.[1] The purpose of the exercise is to introduce you to the environment of the course in a setting with little conceptual difficulty. The exercise will consist of an investigation of techniques to solve linear equations.

The Exercise is followed by five experiments:

1. Fitting - Least squares algorithms; evaluation of the quality of a fit; polynomial fits to Pearson's K-Ar data with York's weights; non-linear fitting to nuclear spectra.

   This is a too brief survey of techniques for modelling real-world experimental data. A major emphasis is on the correct handling of data which has associated experimental errors, a case which is often ignored outside of physics departments. Measuring the quality of the fit and choosing between different models for a given set of data are also studied.

2. Physical Pendulum - solving the differential equations of the physical pendulum in a Hamiltonian formulation; Runge-Kutta and symplectic algorithms.

   The simple pendulum, where we limit the maximum amplitude θ so that $\sin(\theta) \approx \theta$, has a solution which is, of course, simple harmonic motion. Solving the pendulum without the assumption, however, to determine the angle as a function of time is not amenable to analytic approaches. In this experiment we will investigate techniques to numerically integrate the equations of motion to produce 'time series' of the position and momentum for times $t = t_0 + \Delta t$, where Δt is the 'timestep' of the time series. The answers will all be slightly incorrect, and a crucial aspect of the experiment will be to determine the errors associated with the different algorithms used to integrate the equations.

3. Heat Equation - solving the one dimensional heat equation; explicit and implicit methods and the Crank-Nicholson algorithm.

   The pendulum of the previous experiment involved solving ordinary differential equations. Here we investigate techniques for partial differential equations, concentrating on the heat equation. Schrödinger's equation is also a *parabolic* equation similar in some ways to the heat equation. The techniques studied here are also used for the wave equation and Poisson's equation. For

1. The *Hoop* experiment in the I Year Laboratory is such an exercise. The II Year Laboratory begins with a series of exercises before the actual experiments begin.

many real-world sets of initial conditions and/or boundary conditions, analytic solutions are difficult or impossible to achieve for these equations. We will discover that computational methods of solution must pay attention to the *stability* of the algorithm: many reasonable-looking algorithms just don't work.

4. Simulation and Statistics - Monte Carlo modelling of high energy physics data.

   Data taken from the Stanford Linear Accelerator Center is used to study the decay of the $K^*(1780)^0$ meson from about 100,000 possible events. The theory provides only the probability of a given type of event occurring. Thus we will be using a Monte Carlo process to 'generate' simulated events that have the characteristics predicted by our theoretical model, and will compare the distributions of these generated events to the experimental data.

5. Visualisation - studying ozone data taken from the Total Ozone Measurement Spectrometer (TOMS) aboard the Nimbus 7 spacecraft.

   A day's ozone data taken by TOMS consists of a grid of 180 by 288 data points. Here we will explore techniques to discover patterns in the data by looking at it on a computer screen. Crucial will be the use of color to display values for the data. The data itself has missing sections and discontinuities, and we will investigate techniques of dealing with these.

Most practicing physicists who use computers in their everyday work (which is now most physicists) learned how to use the tool 'by the seat of their pants'. In many ways this has had a positive effect on the way we use this technology. However some of the downsides include:

- occasional published wrong results of calculations and/or data collection due to poor choice of algorithm or insufficient testing.
- a fair number of wasted cpu cycles because of a poor understanding of issues of efficiency.
- a large amount of wasted manpower because of little thought about issues of portability and maintainability of programs.

Thus, although this is definitely **not** a course in Computer Science, we will be emphasising some aspects of that discipline as it applies to the three problems just listed.

Our programming environment is based on *Mathematica* although occasionally we use *C* interacting with *Mathematica* when *Mathematica* itself is too slow. By the end of the course you will be fairly fluent in *Mathematica*. Although it has its own worldview, as do all computer languages, it is sufficiently rich that one can

do a nearly line-by-line translation of large parts of a typical program written in C, FORTRAN, Pascal, etc. into *Mathematica*.[2] Strengths of the language are its symbolic algebra and graphics capabilities, and the fact that it is interpreted so a separate compile step is not necessary. This choice also neatly sidesteps the *C* vs. *FORTRAN* war that continues to rage throughout physics.

II. PROGRAMMING STYLE

As mentioned in the introduction, two of the issues in computer programming are maintainability and portability. *Maintainability* refers to whether the code can be easily modified, updated, or extended at some future time, perhaps by someone other than the person who wrote the original code. *Portability* refers to whether the code can be easily moved to a different computer platform; this is a big issue in *C* and FORTRAN where flavours of the language and operating system dependencies abound.

Both of these issues are addressed by good programming style.

Here is an illustration. Consider the two small *Mathematica* functions on the next page that solve the quadratic equation $ax^2 + bx + c = 0$. They both return identical results, which are two answers, and each answer has a real and imaginary part.[3] Suppose that some value of $b^2 \gg 4ac$ is causing a *Catastrophic Cancellation* so the routines are returning wrong results. Which of the two would you prefer to modify and why?

A too brief list of some elements of good style follows:

1. Comment everything. When writing a piece of code, it is 'obvious' to you what is going on. It will not be obvious to someone else, or even to you two weeks later.
2. Choose variable names that are descriptive. For *Mathematica* programs, having them begin with a lower-case letter insures that they will not collide with built-in names.

---

2. In fact, if you are already fluent in a programming language I can perhaps identify it by looking at your first *Mathematica* program. In §V of these Notes there are some further remarks on this issue.
3. The built-in *Mathematica* procedure `Solve[]` also solves this equation. In addition, `Sqrt[]` can handle negative arguments automatically, although extra code would be needed to sort the answers into real and imaginary parts in the form of the example.

```
mySolve[a_, b_, c_] := Module[ {ans1, ans2, disc},
     disc = b^2 - 4*a*c;
     If[ disc >= 0,
          ans1 = { (-b + Sqrt[disc])/(2a), 0 };
          ans2 = { (-b - Sqrt[disc])/(2a), 0 },
     (* else *)
          ans1 = { -b/(2a), Sqrt[-disc]/(2a) };
          ans2 = { -b/(2a), -Sqrt[-disc]/(2a) };
     ];
     {ans1, ans2}
]
```

```
mySolve[a_, b_, c_] := Module[ {q, p},
Goto[m1];
Label[m0];
Goto[m2];
Label[m1];
     q = { (-b + Sqrt[b^2 - 4 a c])/(2a), 0 };
     p = { (-b - Sqrt[b^2 - 4 a c])/(2a), 0 };
     If[ b^2 - 4*a*c >= 0,
          Goto[m3],
          Goto[m0]
     ];
Label[m2];
     q = { -b/(2a), Sqrt[-(b^2 - 4 a c)]/(2a) };
     p = { -b/(2a), -Sqrt[-(b^2 - 4 a c)]/(2a) };
     Goto[m3];
Label[m3];
     {q, p}
]
```

3. Clear and readable code is better than tricky algorithms or implementations. If performance issues require tricks for greater speed, they can be put in later. First get it right, then make it fast *if necessary*. The programmer's rule is: KISS.[4]

4. Good languages have a number of control structures; in *Mathematica* these include `If, For, Which, While,` etc. Use the construct that naturally matches the operation being performed instead of bending another one to an unsuitable task. In general, the `Goto` is bad for you.

5. Hide the inner workings of functions from the outside world. Then, if necessary they can be re-written with no side effects.

6. If the same lines of code are being used throughout, put them in a separate function.

7. Some languages, such as *Mathematica* and C, allow the unrestricted use of 'white space' (spaces, tabs, newlines). Use the indentation to make the logical flow of the program clear. Always use the same style of indentation.

8. Typically control structures get nested, as in:

```
If[ test1 ,
     operation1 ;
     If[ test2 ,
          operation2 ;
          If[ test3 ,
               operation3 ;
               ...
          ];
     ];
];
```

   When the nesting gets deep, it is time to put some parts into separate functions.

9. Use many small functions instead of one great big one. This may not be an issue for the relatively small programs you will be writing in this course. Also, although *Mathematica* is controlled by its vendor, Wolfram Research, and so has only a single standard, compiler writers for C, FORTRAN, and

4. Keep It Simple Stupid.

Pascal can't resist adding extensions to the language which are different for every version and vendor. Try to avoid using them, but if you must, isolate them into well-documented functions. Similarly, avoid or isolate operating system dependencies. Note that this advice depends on recognising the extensions and operating system dependencies in the programming environment, which is often the hard part for beginners.

10. Comment everything. When writing a piece of code, it is 'obvious' to you what is going on. It will not be obvious to someone else, or even to you two weeks later.

Mechanics of writing a program

When beginning to write a program, there is a tendency to pull up an editor and start banging away. For most people this is a bad idea. Instead the first draft should be done on a piece of paper. At the least, on the paper should be all internal variable names and blocks of 'pseudo-code'. It can include a flow chart and/or actual complete code including all the semi-colons, commas, etc.

III. REFERENCES

Here is an annotated list of some books discussing the material of this course. In addition you will wish to acquire a copy of the locally written document **Mathematica & UPSCALE**; it is a guide to local features of our version of *Mathematica*; it also includes a bibliography of books on the *Mathematica* programming language.

- Philip R. Bevington, **Data reduction and error analysis for the physical sciences** (McGraw-Hill, 1969).

  A classic text, one of the first on numerical methods. Many people still reach for this one first when confronted with a new problem.

- Brice Carnahan, H.A. Luther and James O. Wilkes, **Applied Numerical Methods** (Wiley, 1969).

  Despite its reliance on FORTRAN and showing its age through things like flowcharts, this book discusses a wide range of topics in numerical methods at about the level of this course. Further, it ties those discussions to the solution of 'real world' problems in physics and engineering.

- Gene H. Golub and James M. Ortega, **Scientific Computing and Differential Equations** (Academic, 1992, ISBN: 0-12-289255).

  Seriously considered as a textbook for this course. It *is* the text for the

Engineering course *CSC383S - Numerical Methods*.

- W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, **Numerical Recipes in C** (Cambridge, 1988, ISBN: 0-521-35465-X).
  W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, **Numerical Recipes** (Cambridge Univ., 1986, ISBN 0-521-30811-9).

  A goldmine of techniques and discussion. The first listed version is in C, obviously, the second in FORTRAN and Pascal. Another candidate for the textbook in this course.

- Samuel S.M. Wong, **Computational Methods in Physics and Engineering** (Prentice-Hall, 1992, ISBN: 0-13-155953-2).

  Professor Wong, of this Department, has been involved in numerical methods for a long time; this book presents his accumulated knowledge at a level close to this course. He also was active in the establishment of the *UPSCALE* facility in this Department.

IV. MARKING SCHEME

The Exercise will be marked *Pass/Incomplete*, and must be passed before work on the experiments may begin. Each experiment will be marked out of **18** points, so the five experiments will account for 90% of your mark in the course. The remaining 10% will be determined by an oral examination at the end of the course.

Although alternatives are available, past experience has shown that the most comfortable and efficient environment for you to do your work is the *Mathematica* Notebook. This can be invoked either from the main menu within the *UPSCALE* menu-system or from a shell prompt by typing `mathematica`. *UPSCALE* has a growing number of pre-prepared notebooks available. One advantage of invoking from within the menu system is that you will be prompted for the Notebook you wish to load, without having to know file or directory names.

The Exercise and Experiments are written as Notebooks, and you will be able to access them both on-line and as hardcopy. They serve the same function as the "Guide Sheets" of our more traditional laboratories, and typically contain introductory text, references, sample *Mathematica* commands, and code listings.

Once you have loaded the notebook for the Exercise or one of the Experiments, you can save it in your own areas. Thus you can:

1. Substitute for our text and explanation with your own description of what you did, what happened, and what you think it means. Thus, the notebook will serve the same function as a lab book in our traditional labs.

2. Include sample *Mathematica* commands and their output. When you are asked to write a *Mathematica* procedure, the code should be included in the notebook. Typically, the text that you write explaining what you are doing, what happens and why, "flows" around the *Mathematica* input and output cells.

When the exercise or experiment is complete, we will copy your notebook into our own areas for evaluation.

The evaluation of your work will be in part based on your descriptions of your work and in part on the code that you write. As a rough guideline, the table shows how the marks for each part of an experiment will be assessed.

Components of mark

What	Description	Percent
Content	Is the experiment completely done and understood? Are the conclusions correct and meaningful? Are all questions answered? Are there suggestions for further exploration?	35%
Clarity	Are the materials easy to read and understand?	7%
Completeness	Do the written materials tell me everything that you did?	8%
Correctness	Do your programs work in all cases, including boundary cases? Do they handle bad data gracefully? Are they efficient?	30%
Style	Good variable names? Clear and obvious use of variables? Clear and logical structure? Indentation reflects the logical structure of the program? Can be easily understood and modified? Well commented?	20%

V. THE MATHEMATICA PROGRAMMING ENVIRONMENT

Between the references, Guide Sheets, and the **Mathematica & UPSCALE** document, most of the information required to get up to speed with *Mathematica* is provided. A few exceptions and further discussion occurs in this section.

V.A Debugging

Traditionally the first program to be written is one that prints *hello world* on the screen and exits. At its simplest, you may type:[5]

```
In[1]:= HelloWorld[] := Print["hello world"]
```

and then you can type:

```
In[2]:= HelloWorld[]
hello world

In[3]:=
```

Suppose that we want to add an argument to our definition that governs the number of times it does the print. We can add the following definition:

```
In[3]:= HelloWorld[num_Integer] := Module[ {i},
     For[i = 1, i <= num, i++,
          Printf["hello world"]
     ]
]
```

This second definition is executed only if a single integer argument is given to the call.[6] However, probably because I coded too much C as a child, I have inadvertently typed `Printf` instead of `Print` so the second form produces no output:

```
In[4]:= HelloWorld[]
hello world

In[5]:= HelloWorld[3]

In[6]:=
```

Although it is easy for you to see the mistake after I have pointed it out, usually such errors are very difficult to spot, especially for their perpetrator. In this case, unless you can spot the mistake you may not be sure whether the second definition of `HelloWorld` is even being executed, or perhaps there is something wrong with the `For` specification or ....

One of the disadvantages of *Mathematica* and most other interpreted languages is that in C, FORTRAN, or Pascal the above kind of error would be caught by the compiler. *Mathematica* catches many errors at run-time and attempts to provide a useful error message; it didn't catch this one and actually doesn't think it is an error at all. In addition, powerful symbolic debuggers are increasingly

---

5. The `In[1]:=` is the prompt from *Mathematica*; you would not type that in.
6. This ability to overload the name space is one of *Mathematica*'s strengths. Object-oriented compiled languages like C++ also have this ability.

available for the compiled languages which allow one to step through the code line by line; *Mathematica* has no such tool. Thus, it is common to sprinkle `Print` statements in a misbehaving *Mathematica* program.

Mathematica does provide a `Trace` command that can be of great use in debugging. Here is some sample output from our misbehaving `HelloWorld` program:

```
In[6]:= Trace[ HelloWorld[3] ]

Out[6]= {HelloWorld[3], Module[{i$},

>     For[i$ = 1, i$ <= 3, i$++, Printf[hello world]]],

>    For[i$2 = 1, i$2 <= 3, i$2++, Printf[hello world]], {i$2 = 1, 1},

>    {{i$2, 1}, 1 <= 3, True}, {i$2++, {i$2, 1}, {i$2 = 2, 2}, 1},

>    {{i$2, 2}, 2 <= 3, True}, {i$2++, {i$2, 2}, {i$2 = 3, 3}, 2},

>    {{i$2, 3}, 3 <= 3, True}, {i$2++, {i$2, 3}, {i$2 = 4, 4}, 3},

>    {{i$2, 4}, 4 <= 3, False}, Null}

In[7]:=
```

Examination shows that the procedure *is* being executed, that the `For` loop seems to be running correctly, and we eliminate possibilities until we are led to the offending statement. After correcting the mistake, we can run:

```
In[7]:= HelloWorld[3]
hello world
hello world
hello world

In[8]:=
```

The `Trace` command has many options that allow a large amount of control over what and when you trace. Learning at least some of the simpler of these options is probably worth your time; see the **Mathematica** book, §2.5.10, for more details.

V.B Function Arguments

One of the issues in evaluating various computing languages is whether their functions receive their arguments by value or by address. The distinction can perhaps be made clear by the following pseudo-code:

```
...
x = 3
y = func[x]
...
func[x]
     x = x + 4
     return x
...
```

We have a block of code that assigns the value *3* to the variable *x* and calls the function *func* with *x* as the argument. The function adds 4 to its argument and returns the result, which in this case is assigned to the variable *y*. In a language like *C* all the function *func* knows is the **value** of its argument; thus after the call to the function the value of *x* as seen by the main part of the program is unchanged. In a language like FORTRAN, the function actually gets the **address** where the variable *x* is stored; thus after the call the value of *x* is changed.

Passing arguments by value is usually considered to be superior because it automatically insures that the inner workings of the function are hidden from the rest of the program. This means that the function can be re-written or modified without fear that some other part of the code will be broken by the change. Unfortunately, for *C* arguments that are an array or string are passed by address, while other arguments are passed by value. I tend to consider this a wart in *C*, but one that all C programmers including me use regularly. The code in **Numerical Recipes in C** uses this relentlessly to change the values of an input argument.

In this sense *Mathematica* is much more 'pure'; it won't even allow you to assign a value to an input argument. For example:

```
In[1]:= f[x_] := (x = N[x]; Sqrt[x])

In[2]:= f[2]

Set::setraw: Cannot assign to raw object 2.

Out[2]= Sqrt[2]
```

If you want to use pattern names as local variables in the same way as procedure parameters are used in programming languages like *C* and *Pascal*, for example, you can do it by using an initialized local variable:

```
In[3]:= g[x_] := Module[ {xx = x}, xx = N[xx]; Sqrt[xx] ]
```

```
In[4]:= g[2]

Out[4]= 1.41421
```

This means that although *Mathematica* is very flexible in the sense that you can program it procedurally or functionally, can overload the name space, etc., doing a translation of the data structures and calling conventions of many FORTRAN or C programs requires a redesign.

V.C Portability

We should make a few remarks about portability and *Mathematica*. As already mentioned, since *Mathematica* is controlled by its vendor, Wolfram Research, there is only a single standard for the language; this differentiates it from *C* and *FORTRAN*, where many flavours exist. Wolfram also works hard to insure that the release of a new version of the software doesn't break existing applications, although they are not perfect in this regard. Nonetheless, since *Mathematica* runs on many different computing platforms it is possible to write a program that will work on one machine but not on another. For example, to load the *DiracDelta* package you type:

```
In[1]:= << NumericalMath`DiracDelta`
```

On a UNIX platform, one could also load `NumericalMath/DiracDelta.m` since the file exists in the *NumericalMath* directory and is named *DiracDelta.m*. This would not work in a DOS environment since the directory delimiter is a backslash \, not the forward slash, and on other hardware the filename extension may or not be *.m*. So Wolfram provides the grave ` as a generic directory delimiter and filename extension which is interpreted correctly on all platforms.

V.D Functional and Procedural Programming

Finally, above the distinction between *functional* and *procedural* programming was mentioned. In order to illustrate the distinction, the following program calculates the **mean** of a list of numbers in a procedural style:

```
MeanP1[list_] := Module[ {i, length, sum},
     length = Length[list];
     sum = 0;
     For[i = 1, i <= length, i++,
          sum = sum + list[[i]]
     ];
     sum/length
]
```

The code in this procedure looks almost identical with the way it would be coded in a traditional compiled language like *C* or *FORTRAN*. The same result can be written in a functional style:

```
MeanF1[list_] := Apply[Plus, list] / Length[list]
```

Although efficiency is not a major issue in this course, *Mathematica* is much more efficient executing functional code: in this example for a 10,000 point data set `MeanP1[]` took 3.1 seconds to execute while `MeanF1[]` took 0.56 seconds. If you are familiar with the `Apply[]` function, which is a *Mathematica* built-in, the functional form is probably more readable than the procedural one; it is certainly shorter.

The two versions produce identical results, including nonsense results if they are not given 'sensible' input. Continuing to explore the distinction between these two styles of programming, we modify `MeanP1[]` so that it checks to make sure it receives a list of numbers and the list is not empty.

```
MeanP2[list_] := Module[ {i, length, sum},
        If[ !VectorQ[list],
                Return[$Failed]
        ];
        length = Length[list];
        If[ length == 0,
                Return[$Failed]
        ];
        sum = 0;
        For[i = 1, i <= length, i++,
                sum = sum + list[[i]]
        ];
        sum/length
]
```

Making the same checks functionally:

```
MeanF2[list_] := Apply[Plus, list] / Length[list]  /;
                VectorQ[list] && Length[list] > 0
```

Now, if the procedural version is given a list which is not a vector or has zero length it returns the symbol `$Failed`. The functional version will not even be evaluated.

```
In[1]:= wrongdata = {{1,2},{3,4}};

In[2]:= MeanP2[wrongdata]
```

```
Out[2]= $Failed

In[3]:= MeanF2[wrongdata]

Out[3]= MeanF2[wrongdata]
```

Since procedural languages like *C* and *FORTRAN* dominate Physics as of this writing, it is a difficult question as to whether you will benefit more by programming *Mathematica* procedurally and thereby improve your ability to think about problems that way, or rather program and think about computation functionally. The whole question of procedural versus functional programming tends to be a religious issue in some circles.

Original version by David Harrison, August 1993
This rev: 1.15, date (m/d/y): 09/10/98

On rounding errors

The following are some details of the U.S. Government Accounting Office Report GAO/IMTEC-92-96.

The Patriot missile defence unit *(battery)* uses a 24 bit arithmetic which causes the representation of real time and velocities to incur roundoff errors; these errors become substantial when the Patriot battery ran for 8 or more consecutive hours.

As part of the search and targeting procedure, the Patriot radar system computes a "Range Gate" that is used to track and attack the target. As the calculations of real time and velocities incur roundoff errors, the range gate shifts by substantial margins, especially after 8 or more hours of continuous run.

The following data on the effect of extended run time on Patriot operations from Appendix II of the report would be of interest to numerical analysts everywhere.

Hours	Real time (seconds)	Calculated Time (seconds)	Inaccuracy (seconds)	Approx. Shift in range gate (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0275	55
20	72000	71999.9313	.0687	137
48	172800	172799.8352	.2472	330
72	259200	259199.7528	.2472	494
100	360000	359999.6667	.3333	687

The air fields and sea ports of Dhahran were protected by six Patriot batteries. Alpha battery was to protect the Dhahran air base. On February 25, 1991, Alpha battery had been in operation for over 100 consecutive hours. That’s the day an incoming Scud struck an Army barracks and killed 28 American soldiers.

On February 26, the next day, the modified software, which compensated for the inaccurate time calculation, arrived in Dharan.

Clearly the design of the software did not match the way it was used in the field. It is interesting to muse about whether this error was just a lack of communication, or whether the people who wrote the software just didn’t think about the possibility of a rounding error.

Information provided by Murli Gupta in a Internet posting of June 10, 1992.