Recovery of Jeppson Pipe Network Analysis Software

Bob Apthorpe, <bob.apthorpe@acorvid.com>
Acorvid Technical Services Corp., http://www.acorvid.com/

March 8, 2018

Abstract

Several pipe network flow analysis programs were recovered from a well-known engineering reference (Jeppson, 1976). The source code was manually extracted from the published text, transcription errors were corrected, and the overall code reformatted to compile as Fortran 90. Some of the programs relied on proprietary Sperry-Rand (UNIVAC) libraries and extensions to FORTRAN 66, leading to the development of compatibility libraries to avoid substantially rewriting the software. Verification test cases for each program were developed based on examples in the text. A number of significant errors in the published text were documented and corrected as part of the recovery and verification process. A consistent build environment for the software was created using both the `make` and `CMake` tools, allowing the code to be built with multiple compilers in multiple environments. The build and test process for the software was extensively documented and the code was put under version control and posted publicly at `https://bitbucket.org/apthorpe/jeppson_pipeflow`.

Keywords: Source code recovery, proprietary library removal, verification, build automation, UNIVAC, piping network analysis, water distribution

1 Executive Summary

1.1 Background

In January of 2018, a user on Stack Overflow posted a question about a missing UNIVAC math library used by a historical pipe network flow analysis code described in (Jeppson, 1974); see `https://stackoverflow.com/questions/48265245/univac-math-pack-subroutines-in-old-school-fortran-pre-77`. The text of (Jeppson, 1974) appears to be almost identical to that of (Jeppson, 1976), and after reviewing both texts, a total of 6 programs were found, each between 1 and 3 pages in length. Given the significance of the Jeppson text in instruction and as an engineering reference[1], we decided to attempt to recover the original code.

1.2 Recovery and Revitalization Plan

Three phases of software recovery and revitalization were planned. Phase 1 is the *recovery* phase (section 2) in which minimal changes are made to allow the code to compile and correctly run test cases. Basic build automation is created to ensure the build process is defined and repeatable and all software artifacts (source code, documentation, build configuration, *etc.*) is placed under revision control for safekeeping.

Phase 2 is the *modernization* phase, the goal of which is to convert as much of the FORTRAN 66 code format and idiom to well-formed Fortran 2008 (section 3). The goal is to clarify the logic of the code by replacing low-expressivity constructs with clearer modern alternatives and remove problematic and obsolete features which reduce long-term maintainability.

Phase 3 is the *modularization* phase where the applications are analyzed to find repetitive or common code elements and extract them to subroutines and functions and organize them into modules (section 2). This achieves the goal of code reuse and improves code quality by allowing *unit testing*, or testing individual code elements in isolation. This is contrasted with *integral testing* where code functionality is tested as part of the whole program. Both unit and integral testing have desirable attributes and detect different types of errors. There are historical and architectural reasons why unit testing is relatively difficult in Fortran, especially

[1] As of March 2018, Google Scholar shows at least 330 citations to (Jeppson, 1976); see `https://scholar.google.com/scholar?cites=15501786215094582199&as_sdt=5,44&sciodt=0,44&hl=en`

when applied to existing (vs new 'greenfield' code); phase 3 illustrates how unit testing can be applied to older code as part of a comprehensive revitalization plan.

In all phases, it is expected the code documentation, test, and build infrastructure will be improved as a side effect of achieving the goals specified in each phase. In some phases, specific goals are set for upgrading documentation, test, and build infrastructure; in general, improvements to these elements are expected to occur in support of other activities and may not be explicitly specified as goals.

The description of the three code modification phases is broken down into four elements:

- Initial Goals,
- Process,
- Reconnaissance, and
- Products and deliverables

Initial goals describes the intent of the modification phase given the knowledge at the beginning of the evolution. Initial goals may not be complete or achievable, but they represent a point-in-time intent and plan which will necessarily be modified by emergent conditions that arise during the evolution.

Process explains the intended actions and techniques which will be used to implement the initial goals. Some actions involve research or information gathering, others are specific modifications to the software or other development activities such as test generation or infrastructure creation.

Reconnaissance details the knowledge gained during review, analysis, and manipulation of the code. Information gained by investigating the code may change the project goals, create new goals in the current or later modification phases, defer current goals to later phases, or delete goals entirely. While reconnaissance implies information gathered prior to taking action on the code, this section focuses on interesting and significant knowledge gained about the code during the entire modification phase.

Products and deliverables lists the tangible artifacts which result from the code modification effort. These include source code, build, test, and documentation infrastructure, and utility scripts, coding, and data generated during the modification phase.

This structure is intended to show how the goals of a software modification plan are changed by information learned during the project. Often project *post*

mortem reviews will omit false starts, missing information, and changes to plans, giving the impression that a project was planned perfectly and implemented completely. This presents a false sense of certainty and describes a software development process that deviates substantially from actual practice. If such reviews are to have any value as institutional knowledge and as teaching tools, they must describe *work-as-practiced*, not *work-as-imagined*. This is especially important in safety-critical work since the difference between *work-as-practiced* and *work-as-imagined* makes work planning and management much more difficult when balancing operational, financial, and safety goals. At best, this only leads to unrealistic assumptions about schedule and budget; at worst it can lead to workers inadvertently being set up to fail, resulting in poor code quality or project failure, potentially with disastrous consequences. The safety significance of deviations between work-as-imagined and work-as-practiced is discussed at length in (Hollnagel, 2014).

Finally, key insights gained from this project are summarized in section 5.

1.3 Refactoring

Note that the code modifications described in phases 1 through 3 do not add any new functionality or change the input file format or user interface of the software. This process is known as *refactoring* and it is a critical part of revitalizing and clarifying software to improve its long-term maintainability(Fowler and Beck, 1999). Here our intent is limited to recovery and internal modernization rather than extending or changing the functionality of the software. The recovery and revitalization process set the stage for functional changes to the software by reconstituting the build and test infrastructure, clarifying code logic and removing problematic constructs to let developers focus on feature addition instead of working around brittle code constructs or unnecessary complexity to ensure code stability is maintained during development.

2 Phase 1: Recovery

2.1 Initial Goals

The recovery project had several goals:

- recover each program presented in the text such that it could be compiled in a modern environment with a commonly available Fortran compiler which

supported Fortran 95 or later (*e.g.* `gfortran` or Intel's `ifort`),

- provide test cases so the code could be verified as operating correctly,
- replace or rewrite proprietary libraries without substantially rewriting the application code,
- provide build infrastructure to simplify building the code in a variety of environments
- configure auto-documentation tools such as Doxygen so the internal structure of the code may be easily reviewed,
- place the code under configuration control to safeguard it during recovery (*i.e.* allow changes to be reverted during recovery to prevent wasted effort and to document the recovery process)
- document the build and test process to assist others in understanding and using the software

These were considered the baseline goals for the first phase of the project from which specific tasks and products would be derived. Later phases of the project

2.2 Process

The plan for recovering each program involved copying and reformatting raw text into legal ASCII text[2] format source code files, iterative removal of syntax errors, generation of test input, and iterative removal of runtime and input errors.

Since (Jeppson, 1974) was in electronic format and contained searchable text, the scanned source code text was copied from the PDF file into text files and compared against the typeset source code in (Jeppson, 1976). This allowed for disambiguation of characters such as `1` (one), `l` (lower-case 'L'), `I` (upper-case 'I'), and `/` (forward slash or stroke) and `0` (zero) and `O` (upper-case 'O'). These characters pose a specific problem for optical character recognition (OCR) systems, especially when typeset in a proportional font not designed specifically for OCR processing.

[2]The FORTRAN language dates to 1956 and predates the ASCII character standard by approximately a decade; ASCII was defined in 1966.

Once obvious typographical errors had been removed from the code, it would be formatted as fixed-format Fortran (*i.e.* FORTRAN 77 style) and compiled, editing until all syntax errors had been removed.

After successful compilation, the source code would be minimally transformed from fixed to free format, replacing comment and continuation characters, adding missing structural element (`program` and `end program` elements), converting Hollerith strings[3], and ensuring line labels [4] were uniquely associated with either a `format` or `continue` statement. This was done to clarify logic and conform with modern Fortran development practice.

Test cases would then be created, with code input either taken directly from the original text or derived from the supplied examples. Input was carefully compared against the read format specified in the code to ensure the input could be read properly.

The code would be run against the test cases and the results would be compared against those reported in the text. While results were not expected to match digit-for-digit, three significant figures of accuracy was deemed sufficient in most cases for verifying the code.

The process described is an idealization and it was known that other emergent tasks would be required to complete the recovery of each program. Proprietary coding structures would need to be identified and removed or replaced. Missing dependencies such as library routines would need to be replaced as well. Since the code would be repeatedly compiled to remove syntax errors, build scripts would be required.

To test the code, compiler options would need to be set based on code behavior. Legacy code may take advantage of proprietary compiler features or non-standard defaults, and some features of modern Fortran compilers must be adjusted (*e.g.* floating point exception handling). A subtle aspect of code recovery is predicting and discovering the implied assumptions about the legacy computing environment. Often the original build environment has been lost and assumptions about local variable persistence, automatic initialization, *etc.* may not be valid in the new build environment.

---

[3] 'Hollerith'-formatted text is prefixed by the length of the text and the letter 'H'. For example, the text string `'cat'` is written `3Hcat` as a Hollerith string.

[4] Used to denote `goto` and `end=` target statements as well as `do` loop boundaries

2.3 Reconnaissance

The code in Chapter 5 of the Jeppson texts ("Linear Theory Method") was investigated first since it was relatively long and complex (see Figures 1 through 7). This code presented a number of challenges. While full text of the source code was available in the PDF version of (Jeppson, 1974), the OCR process had difficulty recognizing some of the text; even under magnification, parts of the PDF were still illegible (Figure 8). Luckily the print version of (Jeppson, 1976) was legible and was generally sufficient to clarify any ambiguities stemming from the illegibility of (Jeppson, 1974).

```
      INTEGER JN(40,5),NN(40),JB(20),IFLOW(40),LP(8,20),
     $JC(50)
      REAL D(50),L(50),A(50,51),QJ(20),E(50),KP(50),V(2  ),
     $Q(50),EXPP(50),AR(50),ARL(50)
   30 READ(5,110,END=99) NP,NJ, NL,MAX,NUNIT,ERR,VIS,
     $DELQ1
  110 FORMAT(5I5,3F10.5)
C  NP-NO. OF PIPES. NJ-NO. OF JUNCTIONS, NL-NO. OF
C  LOOPS, MAX-NO. OF ITERATIONS ALLOWED, IF NUNIT=0
C  D AND E IN INCHES AND L IN FEET,IF NUNIT=1-D AND
C  E IN FEET AND L IN FEET, IF NUNIT=2 D AND E IN
C  METERS AND L IN METERS, IF NUNIT=3 D AND E IN CM
```

Figure 1: Original source code for linear method solver (1 of 7)

```
C  AND L IN METERS.
  100 FORMAT(16I5)
      NPP=NP+1
      NJ1=NJ-1
      READ(5,101) (D(I),I=1,NP)
      READ(5,101) (L(I),I=1,NP)
      READ(5,101) (E(I),I=1,NP)
  101 FORMAT(8F10.5)
      DO 48 I=1,NP
   48 E(I)=E(I)/D(I)
      IF (NUNIT-1) 40,41,42
   40 WRITE(6,102) (D(I),I=1,NP)
  102 FORMAT('OPIPE DIAMETERS (INCHES)'/,(1H ,16F8.1))
      DO 43 I=1,  NP
   43 D(I)=D(I)/12.
      GO TO 44
   41 WRITE(6,112) (D(I),I=1,NP)
  112 FORMAT( 'OPIPE DIAMETERS (FEET)'/,(1H ,16F8.3))
   44 WRITE (6,103) (L(I),I=1,NP)
  103 FORMAT('OLENGTHS OF PIPE (FEET)'/,(1H ,16F8.0))
      G2=64.4
      GO TO 50
   42 IF (NUNIT .EQ. 2) GO TO 45
      DO 46 I=1,NP
   46 D(I)=.01*D(I)
   45 WRITE(6,113) (D(I),I=1,NP)
  113 FORMAT('O PIPE DIAMETERS (METERS)'/,(1H ,16F8.4))
      WRITE(6,114) (L(I),I=1,NP)
  114 FORMAT('O LENGTH OF PIPES (METERS)'/,(1H ,16F8.1))
      G2=19.62
      WRITE(6,115) (E(I),I=1,NP)
  115 FORMAT('O RELATIVE ROUGHNESS OF PIPES'/,
     $(1H ,16F8.6))
```

Figure 2: Original source code for linear method solver (2 of 7)

```
C  INFLOW--IF 0 NO INFLOW, IF 1 THEN NEXT CARD GIVES
C  MAGNITUDE IN GPM. IF 2 NEXT CARD GIVES MAGNITUDE
C  IN CFS. IF 3 NEXT CARD GIVES MAGNITUDE IN CMS.
C  NNJ--NO. OF PIPES AT JUNCTIONS--POSITIVE FOR INFLOW
C  NEGATIVE FOR OUTFLOW. JN--THE NUMBERS OF PIPES
C  AT JUNCTION, IF FLOW ENTERS MINUS- IF FLOW LEAVES
C  THE PIPE NUMBER IS POSITIVE.
      DO 70 I=1,NP
      AR(I)=.78539392*D(I)**2
   70 ARL(I)=L(I)/(G2*D(I)*AR(I)**2)
      II=1
      DO 1 I=1,NJ
      READ(5,100) IFLOW(I),NNJ,(JN(I,J),J=1,NNJ)
      NN(I)=NNJ
      IF(IFLOW(I) - 1) 1,2,3
    2 READ(5,101) QJ(II)
      QJ(II)=QJ(II)/449.
      JB(II)=I
      GO TO 4
    3 READ(5,101) QJ(II)
      BJ(II)=I
    4 II=II+1
    1 CONTINUE
C  NUMBER OF PIPES IN EACH LOOP (SIGN INCLUDED)
      DO 35 I=1,NL
      READ(5,100) NNJ,(LP(J,I),J=1,NNJ)
   35 LP(8,I)=NNJ
      DO 5 I=1,NP
      IF (NUNIT .GT. 1) GO TO 66
      KP(I)=.0009517*L(I)/D(I)**4.87
      GO TO 5
   66 KP(I)=.00212*L(I)/D(I)**4.87
    5 CONTINUE
      ELOG=9.35*ALOG10(2.71828183)
      SUM=100.
      NCT=0
   20 II=1
      DO 6 I=1,NJ1
      DO 7 J=1,NP
```

Figure 3: Original source code for linear method solver (3 of 7)

```
    7 A(I,J)=0.
      NNJ=NN(I)
      DO 8 J=1,NNJ
      IJ=JN(I,J)
      IF(IJ .GT. 0) GO TO 9
      IIJ=ABS(IJ)
      A(I,IIJ)=-1.
      GO TO 8
    9 A(I,IJ)=1.
    8 CONTINUE
      IF(IFLOW(I) .EQ. 0) GO TO 10
      A(I,NPP)=QJ(II)
      II=II+1
      GO TO 6
   10 A(I,NPP)=0.
    6 CONTINUE
      DO 11 I=NJ,NP
      DO 22 J=1,NP
   22 A(I,J)=0.
      II=I-NJ1
      NNJ=LP(8,II)
      DO 12 J=1,NNJ
      IJ=LP(J,II)
      IIJ=ABS(IJ)
      IF(IJ .LT. 0) GO TO 13
      A(I,IIJ)=KP(IIJ)
      GO TO 12
   13 A(I,IIJ)=-KP(IIJ)
   12 CONTINUE
   11 A(I,NPP)=0.
      V(1)=4.
C  SYSTEM SUBROUTINE FROM UNIVAC MATH-PACK TO
C  SOLVE LINEAR SYSTEM OF EQ.
      CALL GJR(A,51,50,NP,NPP,$98,JC,V)
      IF (NCT .GT. 0) SUM=0.
      DO 51 I=1,NP
      BB=A(I,NPP)
```

Figure 4: Original source code for linear method solver (4 of 7)

```
      IF(NCT) 60,60,61
   60 QM=BB
      GO TO 62
   61 QM=.5*(Q(I)+BB)
      SUM=SUM+ABS(Q(I)-BB)
   62 Q(I)=QM
      DELQ=QM*DELQ1
      QM=ABS(QM)
      V1=(QM-DELQ)/AR(I)
      IF(V1 .LT. .001) V1=.002
      V2=(QM+DELQ)/AR(I)
      VE=QM/AR(I)
      RE1=V1*D(I)/VIS
      RE2=V2*D(I)/VIS
      IF(RE2 .GT. 2.1E3) GO TO 53
      F1=64./RE1
      F2=64./RE2
      EXPP(I)=1.
      KP(I)=64.4*VIS*ARL(I)/D(I)
      GO TO 51
   53 MM=0
      F=1./(1.14-2.*ALOG10(E(I)))**2
      PAR=VE*SQRT(.125*F)*D(I)*E(I)/VIS
      IF(PAR .GT. 65.) GO TO 54
      RE=RE1
   57 MCT=0
   52 FS=SQRT(F)
      FZ=.5/(F*FS)
      ARG=E(I)+9.35/(RE*FS)
      FF=1./FS-1.14+2.*ALOG10(ARG)
      DF=FZ+ELOG*FZ/(ARG*RE)
      DIF=FF/DF
      F=F+DIF
      MCT=MCT+1
      IF(ABS(DIF) .GT. .00001 .AND. MCT .LT. 15) GO TO 52
```

Figure 5: Original source code for linear method solver (5 of 7)

```
      IF(MM .EQ. 1) GO TO 55
      MM=1
      RE=RE2
      F1=F
      GO TO 57
   55 F2=F
      BE=(ALOG(F1)-ALOG(F2))/(ALOG(QM+DELQ)-ALOG(QM
     $-DELQ))
      AE=F1*(QM-DELQ)**BE
      EP=1.-BE
      EXPP(I)=EP+1.
      KP(I)=AE*ARL(I)*QM**EP
      GO TO 51
   54 KP(I)=F*ARL(I)*QM**2
      EXPP(I)=2.
   51 CONTINUE
   17 NCT=NCT+1
C     THE NEXT FIVE CARDS CAN BE REMOVED
      WRITE(6,157) NCT,SUM,(Q(I),I=1,NP)
  157 FORMAT('NCT=',I3,' SUM=',E10.3,/,(1H ,13F10.3))
      WRITE(6,344) (EXPP(I),I=1,NP)
  344 FORMAT(1H ,13F10.3)
      WRITE(6,344) (KP(I),I=1,NP)
      IF(SUM .GT. ERR .AND. NCT .LT. MAX) GO TO 20
      IF(NCT .EQ. MAX) WRITE(6,108) NCT,SUM
  108 FORMAT('DID NOT CONVERGE IN',I5,' ITERATIONS -
     $SUM OF DIFFERENCES=',E10.4)
```

Figure 6: Original source code for linear method solver (6 of 7)

```
      IF(NUNIT .LT. 2) GO TO 63
      WRITE(6,127) (Q(I),I=1,NP)
  127 FORMAT('0 FLOWRATES IN PIPES IN CMS',/,(1H ,
     $13F10.4))
      DO 64 I=1,NP
   64 KP(I)=KP(I)*ABS(Q(I))
      WRITE(6,139) (KP(I),I=1,NP)
      GO TO 30
   63 WRITE(6,107) (Q(I),I=1,NP)
  107 FORMAT('O FLOW RATES IN PIPES IN CFS',/,(1H ,
     $13F10.3))
      DO 21 I=1,NP
      KP(I)=KP(I)*ABS(Q(I))
   21 Q(I)=449.*Q(I)
      WRITE(6,138) (KP(I),I=1,NP)
  138 FORMAT(' HEAD LOSSES IN PIPES',/,(1H ,13F10.3))
      WRITE(6,105) (Q(I),I=1,NP)
  105 FORMAT(' FLOW RATES (GPM)',/,(1H ,13F10.1))
      GO TO 30
   98 WRITE(6,106) JC(1),V
  106 FORMAT(' OVERFLOW OCCURRED -- CHECK
     $SPECIFICATIONS FOR REDUCDANT EQ. RESULTING
     $IN SINGULAR MATRIX',I5,2F8.2)
      GO TO 30
   99 STOP
      END
```

Figure 7: Original source code for linear method solver (7 of 7)

```
 40 WRITE(6,102) (D(I),I=1,NP)
102 FORMAT('OPIPE DIAMETERS (INCHES)',/,(1H  ,16F8.1))
    DO 43 I=1,   NP
 43 D(I)=D(I)/12.
    GO TO 44
 41 WRITE(6,112) (D(I),I=1,NP)
112 FORMAT( 'OPIPE DIAMETERS (FEET)',/,(1H  ,16F8.3))
 44 WRITE (6,103) (L(I),I=1,NP)
103 FORMAT('OLENGTHS OF PIPE (FEET)',/,(1H ,16F8.0))
    G2=64.4
    GO TO 50
 42 IF (NUNIT .EQ. 2) GO TO 45
    DO 46 I=1,NP
 46 D(I)=.01*D(I)
 45 WRITE(6,113) (D(I),I=1,NP)
113 FORMAT('O PIPE DIAMETERS (METERS)',/,(1H ,16F8.4))
    WRITE(6,114) (L(I),I=1,NP)
114 FORMAT('O LENGTH OF PIPES (METERS)',/,(1H ,16F8.1))
    G2=19.62
    WRITE(6,115) (E(I),I=1,NP)
115 FORMAT('O RELATIVE ROUGHNESS OF PIPES',/,
   $(1H ,16F8.6))
```

Figure 8: Legibility problems in linear method solver; PDF source document

Other problems were not related to the scanning process. One issue known from the outset of the project was that some of Jeppson's software relied on proprietary UNIVAC math libraries. Figure 9 shows a call to `GJR()`, a multipurpose matrix manipulation and solver routine from the Sperry Rand MATH-PACK library (Sperry Rand Corporation, 1970a), (Sperry Rand Corporation, 1970b). This is the subject of the original Stack Overflow post which led to this code recovery project.

```
C  SYSTEM SUBROUTINE FROM UNIVAC MATH-PACK TO
C  SOLVE LINEAR SYSTEM OF EQ.
     CALL GJR(A,51,50,NP,NPP,$98,JC,V)
```

Figure 9: Call to missing proprietary (UNIVAC) library in linear method solver

A related issue is the use of *alternate return points*, the ability of a subroutine to continue execution at a specified labeled line upon return, rather than continuing execution at the statement immediately following the subroutine call. This was often used for error handling, where one or more return points corresponding to error handling or diagnostic routines would be provided to a subroutine in case of specific errors. This could easily be accomplished with a return error code and error handling logic following the subroutine call, but the convenience, reduced code size, and marginal increase in speed occasionally made this rarely-used feature attractive. Alternate return points were never popular (thankfully) but one appears here as the `$98` term in the call to `GJR()`. This is confirmed by inspecting the code; the line labeled `98` begins a code block which writes an error message and sends control to the beginning of the code to read input for another analysis.

Other problems result from the source code being retyped for publication rather than being reproduced directly from machine-readable source code. Several typographical errors were found which could be attributed to the transcription process (Figures 10 and 11). In some cases the error was easy to detect and correct; 10 shows the variable `JB` mis-typed as `BJ`. In others, such as the missing closing (right) parenthesis in the line beginning with `Q=` in Figure 11, the resulting code was ambiguous and required further analysis and testing to determining the intent of the code in order to correct. This example is from corrective flow solver given in the latter half of Chapter 6, named `JEPPSON_CH6B` in the recovered code distribution.

The first approach to resolving the ambiguity was to compare the source code published in both (Jeppson, 1974) and (Jeppson, 1976) to determine if the error

```
      JB(II)=I
      GO TO 4
    3 READ(5,101) QJ(II)
      BJ(II)=I
```

Figure 10: Typographical error in variable JB in linear method solver

```
      DO 33 IK=1,NPUMP
      IL=IABS(LLP(IK))
      DO 33 KK=1,NNP
      IF(IL .NE. IABS(LP(II,KK))) GO TO 33
      Q=ABS(FLOAT(LLP(IK)/IL)*QI(IL)+DQ(II)
      HP=(A(IK)*Q+B(IK))*Q+HO(IK)
      IF(LLP(II) .LT. 0) GO TO 35
      DR(II,NLP)=DR(II,NLP)-HP
      DR(II,IL)=DR(II,IL)+2.*A(IK)*Q+B(IK)
      GO TO 33
   35 DR(II,NLP)=DR(II,NLP)+HP
      DR(II,IL)=DR(II,IL)-2.*A(IK)*Q-B(IK)
   33 CONTINUE
```

Figure 11: Ambiguity in corrective flow Newton-Raphson solver source code

had been corrected in the later text. Unfortunately, the source code in both documents appears to be identical; errors found in one text have been confirmed to occur in both.

The second approach was to apply subject-matter-expert knowledge (fluid mechanics, piping system analysis, computational modeling) and source code analysis to determine the intent of the code and suggest corrections. The assumption was that adding a single closing parenthesis would resolve the error. Examining the code, the corrected line has four legal permutations:

1. `Q = ABS(FLOAT(LLP(IK))/IL) * QI(IL) + DQ(II)`
2. `Q = ABS(FLOAT(LLP(IK)/IL)) * QI(IL) + DQ(II)`
3. `Q = ABS(FLOAT(LLP(IK)/IL) * QI(IL)) + DQ(II)`
4. `Q = ABS(FLOAT(LLP(IK)/IL) * QI(IL) + DQ(II))`

Reviewing the code input and source code, the variables `QI` (the initial flow) is always positive and `DQ` (the corrective flow) may be positive or negative. `IL` is always positive since it is the absolute value of `LLP`, the signed pipe index indicating the presence of pump `IK`. Thus `QI(IL)` is the magnitude of flow through pipe `IL` and the sign of `LLP(IK)` gives the direction of the flow from a pump in pipe `IL` (positive for clockwise in the flow loop, negative for counter-clockwise). By inspection we can eliminate interpretations 1 and 2 since both `ABS(FLOAT(LLP(IK))/IL)` and `ABS(FLOAT(LLP(IK)/IL))` will always evaluate to +1.0.

As an aside, the construct `FLOAT(I/ABS(I))` is equivalent to `sign(1.0, real(I))`, however the `sign()` intrinsic function is part of the FORTRAN 77 standard and was unlikely to be available in Sperry Rand's 'FORTRAN 5' which was based on the FORTRAN 66 standard. In later refactoring, it may be advisable to replace `FLOAT(I/ABS(I))` with `sign(1.0, real(I))`; even though they are functionally identical and the former is shorter, the intent of the latter is is slightly clearer - to extract the sign of the argument. It is possible that both constructs will be converted to identical assembly code by the compiler[5] so the `sign()` intrinsic should be used to make the intent of the code more obvious.

The next line `HP = (A(IK) * Q + B(IK)) * Q + HO(IK)` is the pressure-flow relationship through a pump, colloquially known as a 'pump curve'; see Figure 12. By convention in these relations, `Q` is always positive; backflow through the pump

[5]This can be confirmed with a disassembler or a tool such as `objdump`. The `-S` flag to `gfortran` will create assembly language files for the target processor.

is not permitted. This implies that the corrective `DQ` flow cannot be greater than the pump flow `QI`; physically this would correspond to 'deadheading' the pump which would result in zero flow[6]. This eliminates interpretation 3, thus permutation 4 should be used as the corrected source code line. This can be tested on sample cases; test the code using both permutation 3 and 4 to determine which yields correct results. While it is best to understand the subject matter and modeling techniques when trying to derive the intent of ambiguous code, experimentation is often a cheap and easy method of confirming or refuting one's assumptions. Both theory and experiment are powerful tools in source code reconstruction and recovery.

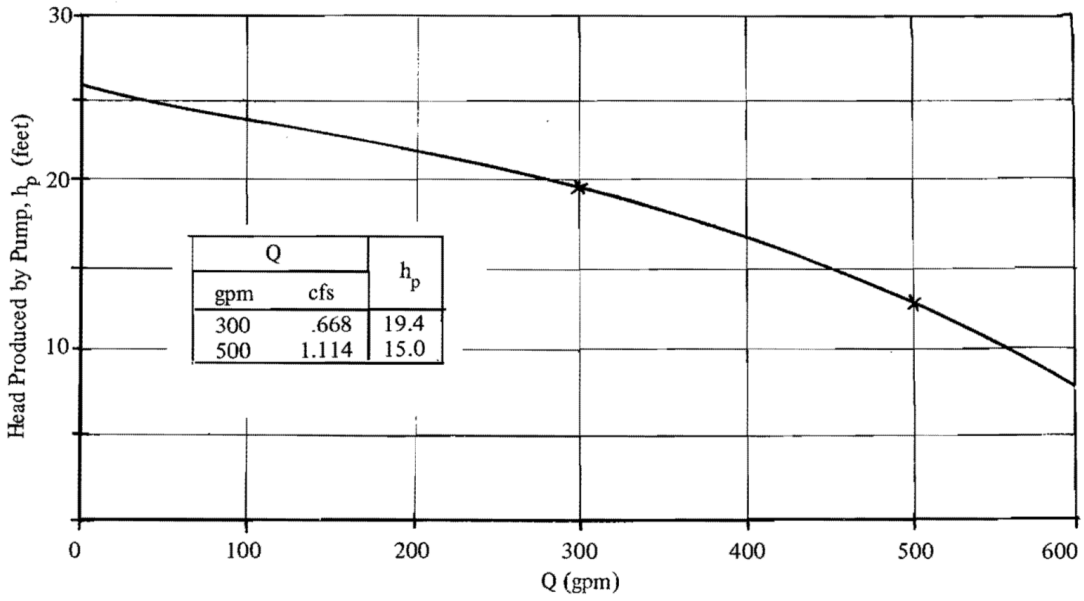


Figure 12: Example pump curve

Note that this explanation is somewhat of a *post hoc* rationalization; experimentation with permutations 3 and 4 was conducted initially and only later was the theoretical basis for selecting permutation 4 derived after becoming more familiar with the code and subject matter. However it was clear that permutations

[6]This is a simple instructional code; a more comprehensive piping network analysis code should account for both *deadheading* (zero flow when pump discharge pressure is lower than the downstream pressure) and *runout* (maximum flow).

1 and 2 were equivalent and of no effect very early in the analysis. The salient point is that the combined use of subject matter expertise and experimentation is powerful in scientific source code reconstruction, regardless of the order in which they are applied.

2.4 Products and Deliverables

2.4.1 Organization

Six programs were identified from the review of the Jeppson texts. Application names were not given in the source documents so the applications were named based on their location in the source text. The application names and roles are:

- `JEPPSON_CH2` calculates the Darcy-Weisbach friction factor and head loss as described on pages 20 of (Jeppson, 1974) and pages 39 of (Jeppson, 1976).
- `JEPPSON_CH4` is an incompressible flow pipe network solver as described on page 34 of (Jeppson, 1974) and pages 64-65 of (Jeppson, 1976).
- `JEPPSON_CH5` is a piping network flow solver based on the linear method solver code described on pages 41-42 of (Jeppson, 1974) and pages 75-58 of (Jeppson, 1976).
- `JEPPSON_CH6A` is a Newton-Raphson method piping network flow solver based on the Newton-Raphson solver code described on pages 63 of (Jeppson, 1974) and pages 119-121 of (Jeppson, 1976).
- `JEPPSON_CH6B` is a piping network flow solver based on the corrective flow Newton-Raphson solver code described on pages 66-67 of (Jeppson, 1974) and pages 126-128 of (Jeppson, 1976).
- `JEPPSON_CH7` solves pipe flow networks using the Hardy Cross method as described on page 75 of (Jeppson, 1974) and pages 148-150 of (Jeppson, 1976).

A single Jeppson code recovery project directory was created and each program was given its own directory under the root project directory. Within each program project directory, the `src` directory contained source code and build scripts, the `test` directory contained input for each test case, the `test/ref` directory contained reference input and output for each test case, and the `userdoc` directory contained any additional files needed to create Doxygen documentation of each program.

2.4.2 Documentation

The root project directory and each of the program project files contained the following documentation files:

1. `BUILD.md`, build and test instructions for the project or specific program
2. `LICENSE.md`, the software license (MIT 'expat')
3. `README.md`, an description of the project or program
4. `TODO.md`, a list of refactoring and enhancement tasks

All documentation files are plain text, formatted as Markdown; see `https://daringfireball.net/projects/markdown/syntax`

2.4.3 Build Scripts

The root project directory, program project directories, and program source directories each contain a file named `CMakeLists.txt` which configure how the CMake cross-platform build automation system builds each program; see `https://cmake.org/` for a description and use of CMake. Table 1 gives an example CMake build script used for building `JEPPSON_CH5`.

Additionally, GNU Makefiles are provided in each program project directory, allowing manual compilation of each program. An example of the Makefile created for the JEPPSON_CH5 program is shown in Table 2.

Table 1: Example CMakeLists.txt for JEPPSON_CH5

```
# States that CMake required version must be greater than 2.8.7
cmake_minimum_required(VERSION 2.8.7)
enable_language (Fortran)
project(JEPPSON_CH5 Fortran)

find_package(LAPACK)

# make sure that the default is a DEBUG
if (NOT CMAKE_BUILD_TYPE)
  set (CMAKE_BUILD_TYPE DEBUG CACHE STRING
       "Choose the type of build, options are: None Debug Release."
       FORCE)
endif (NOT CMAKE_BUILD_TYPE)

# FFLAGS depend on the compiler
get_filename_component (Fortran_COMPILER_NAME ${CMAKE_Fortran_COMPILER} NAME)

if (Fortran_COMPILER_NAME MATCHES "gfortran.*")
  # gfortran
  set (CMAKE_Fortran_FLAGS_RELEASE "-ffpe-trap=invalid,zero,overflow \
-fbacktrace -fno-automatic -finit-local-zero -O3 -g")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-ffpe-trap=invalid,zero,overflow \
-fbacktrace -fno-automatic -finit-local-zero -fbounds-check -Wall -pedantic \
-Og -pg -g")
elseif (Fortran_COMPILER_NAME MATCHES "ifort.*")
  # ifort
  set (CMAKE_Fortran_FLAGS_RELEASE "-O3 -save -zero")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-O0 -save -zero -pg")
elseif (Fortran_COMPILER_NAME MATCHES "g77")
  # g77
  set (CMAKE_Fortran_FLAGS_RELEASE "-funroll-all-loops -fno-f2c -O3 -m32")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-fno-f2c -O0 -g -m32")
else (Fortran_COMPILER_NAME MATCHES "gfortran.*")
  message ("CMAKE_Fortran_COMPILER full path: " ${CMAKE_Fortran_COMPILER})
  message ("Fortran compiler: " ${Fortran_COMPILER_NAME})
  message ("No optimized Fortran compiler flags are known, we just try -O2...")
  set (CMAKE_Fortran_FLAGS_RELEASE "-ffpe-trap=invalid,zero,overflow \
-fbacktrace -fno-automatic -finit-local-zero -O3 -g")
  set (CMAKE_Fortran_FLAGS_DEBUG   "-ffpe-trap=invalid,zero,overflow \
-fbacktrace -fno-automatic -finit-local-zero -fbounds-check -Wall -pedantic \
-Og -pg -g")
endif (Fortran_COMPILER_NAME MATCHES "gfortran.*")

file(GLOB SRC_FILES *.f90)

add_executable(JEPPSON_CH5 ${SRC_FILES})
target_link_libraries(JEPPSON_CH5 ${LAPACK_LIBRARIES} ${EXTRA_LIBS})
```

Table 2: Example Makefile for JEPPSON_CH5

```
objects = alfc_sperry_mathpack.o JEPPSON_CH5.o

# Set compiler and compiler options
FC = gfortran
# Debug - gfortran
FFLAGS = -Wall -pedantic -Og -g -fbounds-check -pg -finit-local-zero \
         -fno-automatic -fbacktrace -ffpe-trap=invalid,zero,overflow
# Release - gfortran
# FFLAGS = -Wall -O3 -g -finit-local-zero -fno-automatic -fbacktrace \
#          -ffpe-trap=invalid,zero,overflow
#
# FC = ifort
# Debug - ifort
# FFLAGS = -warn all -O0 -g -check bounds -pg -zero -save -traceback \
#          -fpe0
# Release - ifort
# FFLAGS = -warn all -O3 -g -zero -save -traceback -fpe0

# Set link options
LDFLAGS = -L/usr/lib/lapack
LIBS = -llapack

# Compile
%.o : %.f90
        $(FC) ${FFLAGS} -c $<

# Link
JEPPSON_CH5 : $(objects)
        $(FC) -o JEPPSON_CH5 $(objects) $(LDFLAGS) $(LIBS)

# Clean
clean :
        /bin/rm -f JEPPSON_CH5 *.o *.mod

# __END__
```

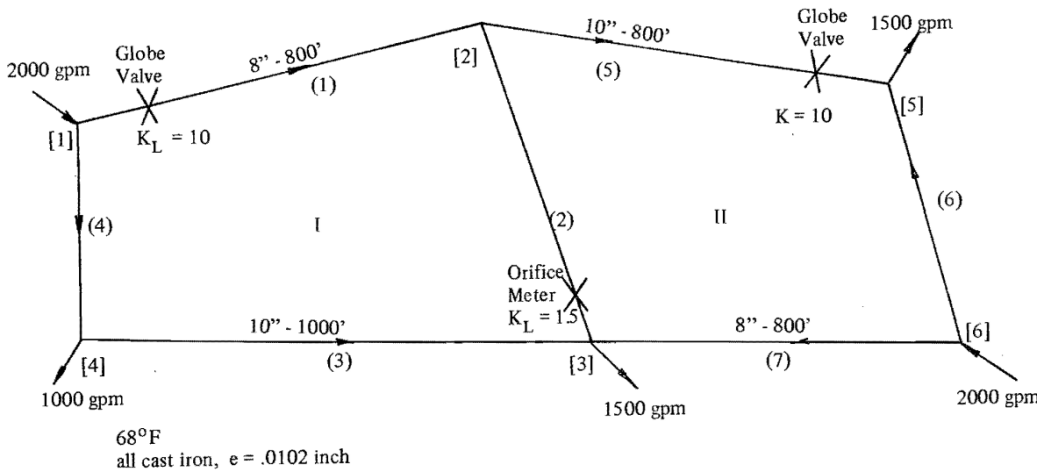
2.4.4 Test Cases

Since the code was provided as part of an instructional text, each code has at least one verification test associated with it, typically in the form of a derivation or example manual calculation ('hand calc') in the text just preceding the source code. Figure 13 shows the diagram and initial conditions for the first test case for the JEPPSON_CH5 program; Figure 14 and Table 3 show the published and reconstructed text input for the code.

Note that in Figure 13, no length or diameter is given for pipes (2), (4), or (6). Figure 14 give the dimensions as 751 feet of 12-inch pipe for (2), 500 feet of 12-inch pipe for (4), and 600 feet of 6-inch pipe for (6); diameters in inches are given on line 2 of the input and lengths in feet are given on line 3 of the input. The length given in the input is an *effective length*; flow obstructions such as the globe valves on pipes (1) and (5) and the orifice meter on pipe (2) add a 'virtual length' to the pipe which allows both frictional and 'form' losses from obstructions to be treated as purely frictional losses. The equivalent length (ΔL) for each flow obstruction loss factor (K_L) is shown in Figure 15.

The loss factor K_L for the orifice meter on pipe (2) is given as 1.5 in Figure 13 but is 1.2 in Figure 15. If the test input file is assumed to be correct, the test case diagram should be revised as in Figure 16. However, if the calculation is assumed to supersede the diagram, the 51 foot 'virtual length' added to pipe (2) would give a real pipe length of $751 - 51 = 700$ feet. As confirmation, the actual length of pipe (1) is given as 800 feet which should be equal to $1106 - 306 = 800$ feet. Similarly, the given actual length plus virtual length of pipe (5) is 800+400 = 1200 feet which is the value given in the input file shown in Figure 14. If, however, the diagram is to be believed, the effective length of pipe (2) should be $700 + \frac{1.5(1)}{0.0238} =$ 763 feet.

This inconsistency should not have a great effect on the results but it does illustrate that a number of errors appear in the text, errors that occur in both Jeppson's paper (Jeppson, 1974) and his (presumably edited) book (Jeppson, 1976). It is important then to check both the source code for errors introduced in publication as well as sample input. In some cases, example calculations are incorrect making these examples worse than useless for code verification. Substantial effort may be wasted attempting to find errors in the code when the errors are in the original manual calculations. An important insight is to understand the sample calculations and confirm their results are correct before using the sample cases as verification tests for the code.



Pipe No.	1	2	3	4	5	6	7
V_1 (fps)	3.35	0.425	0.922	2.76	2.76	5.10	6.70
V_2 (fps)	6.70	0.851	1.84	5.53	5.51	10.21	13.40
$Re_1 \times 10^{-5}$	1.83	3.50	6.29	2.27	1.89	2.10	3.67
$Re_2 \times 10^{-5}$	3.67	6.99	1.26	4.54	3.77	4.19	7.34
f_1	0.0221	0.0250	0.0234	0.0203	0.0212	0.0234	0.0215
f_2	0.0215	0.0226	0.0218	0.0196	0.0205	0.0229	0.0212
b	0.0408	0.146	0.101	0.0472	0.0475	0.0287	0.0215
a	0.0223	0.0213	0.0218	0.0210	0.0216	0.0234	0.0219
n	1.96	1.85	1.90	1.95	1.95	1.97	1.98
K	4.71	.402	1.37	.264	1.14	11.30	3.35

Figure 13: Linear method pipe network test case 1 diagram as published

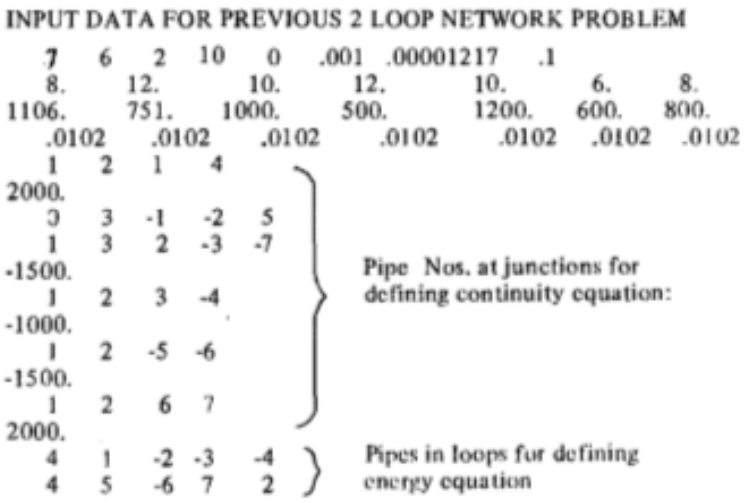


Figure 14: Linear method pipe network test case 1 input as published

Table 3: Linear method pipe network test case 1 input as reconstructed

```
7    6    2    10   0    0.001     0.00001217 0.1
8.0       12.0      10.0      12.0      10.0      6.0       8.0
1106.0    801.0     1000.0    500.0     1200.0    600.0     800.0
0.0102    0.0102    0.0102    0.0102    0.0102    0.0102    0.0102
1    2     1    4
2000.0
0    3    -1   -2    5
1    3     2   -3   -7
-1500.0
1    2     3   -4
-1000.0
1    2    -5   -6
-1500.0
1    2     6    7
2000.0
4    1    -2   -3   -4
4    5    -6    7    2
```

The second step is to form equivalent pipes for those pipes containing the globe values and the orifice meter by use of Eq. 4-8,

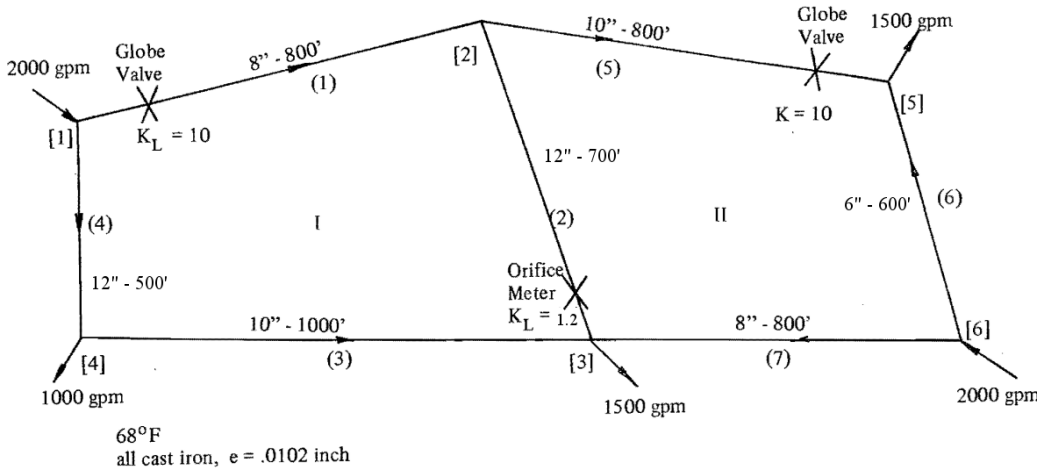
$$\Delta L = \frac{K_L D}{f} = \frac{10(.667)}{.0218} = 306 \text{ ft} \quad \text{for globe valve in pipe (1)}$$

$$\Delta L = \frac{10(.833)}{.02085} = 400 \text{ ft} \quad \text{for globe valve in pipe (5)}$$

$$\Delta L = \frac{1.2(1)}{.0238} = 51 \text{ ft} \quad \text{for the orifice meter in pipe (2)}$$

These lengths are added to the pipe lengths to form equivalent pipes. The K's in the table below the network are for the equivalent pipes.

Figure 15: Length equivalent of form loss factors calculated for linear method pipe network test case 1



Pipe No.	1	2	3	4	5	6	7
V_1 (fps)	3.35	0.425	0.922	2.76	2.76	5.10	6.70
V_2 (fps)	6.70	0.851	1.84	5.53	5.51	10.21	13.40
$Re_1 \times 10^{-5}$	1.83	3.50	6.29	2.27	1.89	2.10	3.67
$Re_2 \times 10^{-5}$	3.67	6.99	1.26	4.54	3.77	4.19	7.34
f_1	0.0221	0.0250	0.0234	0.0203	0.0212	0.0234	0.0215
f_2	0.0215	0.0226	0.0218	0.0196	0.0205	0.0229	0.0212
b	0.0408	0.146	0.101	0.0472	0.0475	0.0287	0.0215
a	0.0223	0.0213	0.0218	0.0210	0.0216	0.0234	0.0219
n	1.96	1.85	1.90	1.95	1.95	1.97	1.98
K	4.71	.402	1.37	.264	1.14	11.30	3.35

Figure 16: Revised linear method pipe network test case 1 diagram

2.4.5 MATH-PACK Compatibility Module

As part of this project, a compatibility library was created to simplify recovery of legacy codes which depend on proprietary libraries such as the Sperry Rand MATH-PACK (Sperry Rand Corporation, 1970a), (Sperry Rand Corporation, 1970b) and STAT-PACK (Sperry Rand Corporation, 1969), (Sperry Rand Corporation, 1970c) libraries and those from the *Numerical Recipes* texts (Press et al., 1986). The scope of this project only requires compatibility with the MATH-PACK routines, specifically the Gauss-Jordan matrix solver `GJR`.

A web search revealed the contents of the MATH-PACK and STAT-PACK libraries (Albert and Whitehead, 1986). This reference only provided routine names and brief descriptions, not the full calling interface (argument list) of the routines. Documentation of the interface of `CGJR` was found in (Stodt, 1978) and the source code for `DGJR` and `CGJR`, the double precision and complex versions of the single precision `GJR` were found in (Ding and Kennedy, 1982).

The `GJR` routine performs one or more matrix manipulation functions depending on the value of the argument `V(1)`: matrix inversion, calculation of determinant, or solving $A\vec{x} = \vec{b}$ for $\vec{x}$ using the Gauss-Jordan method. The Jeppson codes only use the latter matrix solution mode of `GJR` so the immediate focus was on implementing the `GJR` interface and finding the appropriate implementation functions in LAPACK.

There are three reasons the source code from (Ding and Kennedy, 1982) was not used in the development of the compatibility library. The first is that (Ding and Kennedy, 1982) was not found until a month after the compatibility library was written. The second is that the intent of providing a compatibility library was to replace proprietary code with open source cod; the original Sperry code is likely still under copyright and would not be legal to redistribute without the copyright holder's consent. Finally, it was felt that the LAPACK libraries are better tested than the original MATH-PACK routines.

The `GJR` compatibility routine consists of a wrapper around the LAPACK routines `SGETRF` (*LU* factorization), `SGETRI` (inversion of *LU* factorized matrix), and `SGESV` general matrix solution. The `V(1)` variable is interpreted to determine the function(s) `GJR` is expected to perform, the original arguments are copied to allocatable arrays compatible with the LAPACK routines, the LAPACK routines are called, and the results are copied back to the original arrays if no errors occurred. Otherwise error codes are set consistent with the original behavior of `GJR`. One major difference is that the alternate return point functionality of the original `GJR` routine is not supported so error handling code is added after the call to `GJR` to

replicate the original application behavior.

At present, only integration tests with the Jeppson applications are performed; there are no unit tests or integration tests for the `GJR` routine beyond those supplied for the Jeppson applications. This also implies that the matrix inversion and determinant calculation functions have not been tested since neither function is used by the Jeppson applications. More comprehensive testing is needed for the compatibility libraries however the matrix solver functionality appears to work correctly in the Jeppson applications. This is better than no testing and is sufficient for the initial use case of the compatibility library in the recovery phase of this project.

3 Phase 2: Modernization

3.1 Initial Goals

The primary goal of the modernization phase is to convert as much of the FORTRAN 66 code idiom to well-formed Fortran 2008 in order to clarify its logic and reduce the number and severity of warnings issued by the compiler. Refactorings include:

- Convert the source code to free-format
- Properly indent the code
- Add Doxygen-compatible documentation to routines and variables
- Replace `GOTO` logic with modern control structures such as `IF-THEN-ELSE` ‘`DO-WHILE`, `SELECT-CASE`, `CYCLE`, and `EXIT`
- Replace `DO-CONTINUE` with `DO-END DO` to remove line numbers
- Replace numeric line numbers with text labels
- Replace common numeric literals with named constants (π, e, *etc.*)
- Declare all variables and disable implicit variable typing and declaration with ‘IMPLICIT NONE‘
- Convert textual conditional operators to modern intuitive equivalents, *e.g.* change `.EQ.` to `==`, `.GT.` to `>`, `.LE.` to `<=`, `.NE.` to `/=`, and so on

- Convert type-specific intrinsic functions to generic equivalents, *e.g.* change `IABS` to `ABS`, `ALOG10` to `LOG10`
- Replace simple loops with array operations
- Convert Fortran keywords to lower case

Several references which may prove useful in understanding modern Fortran constructs and the rationale for these changes include (Metcalf et al., 2011), (Markus, 2012), and (Clerman and Spector, 2011).

3.1.1 Process

Much of this process could be performed mechanically. The `findent` tool (`https://sourceforge.net/projects/findent/`) rapidly converted the source code from fixed to free format. Much of the rote work was performed manually in a text editor but is more reliably performed by a dedicated refactoring tool such as the Photran plugin for Eclipse (`https://www.eclipse.org/photran/`) or the commercial PlusFORT toolkit (`http://www.adeptscience.co.uk/products/fortran-tools/plusfort-with-spag/plusfort-version-6.html`)

Untangling FORTRAN 66 style `IF-GOTO` logic required manual effort as loops in several applications were found to be open-code equivalents of what should have been function or subroutine calls. As the intent of the code was derived, plans were developed to refactor code segments into standalone routines in the modularization phase of the project described in section 4.

Adding variable declarations was accomplished by adding `IMPLICIT NONE` to each application and parsing variable names from the compiler error output. Following Fortran name-type convention, variables whose names began with `[I-N]` were declared `INTEGER` and the remainder were declared `REAL`. This was accomplished with a short shell script using the common UNIX text manipulation tools `grep`, `sed`, and `sort`. This script would not be sufficient for refactoring a more complex program but in this instance it was effective in quickly producing declaration statements for the Jeppson applications. This could have been accomplished just as easily with any scripting language capable of regular expression search and and robust text manipulation features.

Table 4: gfortran error parsing script to generate variable declarations

```
#!/bin/sh

make 2>&1 \
    | grep 'Error: Symbol .* has no IMPLICIT type' \
    | sed 's/^Error: Symbol .\([a-z][a-z0-9_]*\). at (1) has no IMPLICIT type.*/\1/i' \
    | sort -u \
    | sed 's/^[a-ho-z].*/    real :: \U&/;s/^[i-n].*/    integer :: \U&/'
```

3.2 Reconnaissance

The structure of the code is made apparent by the indentation performed by `findent` and refactoring is simplified by ensuring line labels are uniquely associated with only `FORMAT` and `CONTINUE` statements (performed during the recovery phase). This clearly delineated loops for analysis and refactoring. Comparing applications to each other showed commonalities which could be be marked for later refactoring or, in the case of common numeric literals, extracted into parameters[7].

In several applications, a complex loop was analyzed and found to act as a pair of independent and identical sequences of operations with differing initial conditions. Since the goal of this phase of the project was to convert archaic FORTRAN to a modern idiom, refactoring the body of this loop was deferred to the modularization phase of refactoring as described in section 4. The line between modernization and modularization is not well defined but it was felt that deferring this work would avoid expanding the scope of the current refactoring phase. There is a temptation to fix problems as soon as they are found and it can be difficult to simply mark a problem as discovered and put off refactoring and resolution until later. However, separating discovery from resolution allows refactoring work to be planned and keeps the project scope constrained and manageable. Since this is a project with no fixed schedule or budget[8] it was felt that deferring work illustrates the discipline of breaking a software recovery project into distinct phases to ensure deliverables are produced on regular schedule without losing track of deferred work. The `TODO.md` for each application was updated to track refactoring of the open-loop code.

[7] In Fortran parlance, *parameters* are named constants defined at compile time

[8] This project is a technical demonstration with neither schedule nor budget

3.3 Products and Deliverables

No new products or deliverables were specified beyond updated source code files and documentation. The existing project architecture, content, and infrastructure were expected to be sufficient for this refactoring phase.

4 Phase 3: Modularization

4.1 Initial Goals

The primary goal of this phase of refactoring is identifying and isolating common sets of operations into *program units*; code fragments would be converted to *subroutines* or *functions* which would then be grouped as *modules* by role. Following good software design practice, subroutine and function interfaces would be clearly documented with the *intent* of each argument explicitly specified. Intent refers to whether an argument passed through the routine's call interface is meant only to be read, only to be written to, or both read and written to. 'Read-only' arguments are marked as `intent(in)`, 'write-only' (return) variables are marked as `intent(out)`, and those arguments which are both read and altered within a routine are marked as `intent(in out)`. By default, all arguments are considered `intent(in out)`, consistent with FORTRAN 77 behavior.

The `intent` system in modern Fortran provides a mechanism for controlling access to variables and leading to safer code. The Fortran compiler will flag `intent(out)` variables which are returned without having a value set and will flag attempts to change the value of `intent(in)` variables. This also provides hints to the compiler as to which routines may be run in parallel in contrast to those which must run in sequentially when attempting to auto-parallelize code. Further, when intent is set on all variables to be either `in` or `out` (not `in out`) and a routine has no *side effects* such as external I/O or use of intrinsics such as `cpu_time` or time and date functions, a routine may be marked as `pure` or `elemental` which allows the routine to safely be called in parallel code or to operate on array data without explicitly declaring arguments as arrays.

Aside: `intent`, `pure`, and `elemental` provide modern Fortran with limited *design-by-contract* support. These features make commitments to the compiler about how routines will use variables and whether they affect resources outside of those passed in via the routines' call interface. In exchange, the compiler is better able to optimize the executable and offer features such as array operations on routines which support it. While there may be no need for parallel or vector

operations at this stage of refactoring, it is good practice to lay the groundwork for high-performance computing support early in a project, especially if functionality is being moved into a library.

A secondary ongoing goal of this phase is to clarify the code. Much can be accomplished simply by documenting variables and sections of code. In some instances it was found that multiple pieces of information were encoded in a single variable, specifically flow direction and pipe index. When specifying the pipes connecting to a junction or composing a flow loop, a pipe index is specified by an unsigned (positive) integer and flow 'sense' or direction is indicated by sign. The sign convention used for junctions is that flows into the junction are considered positive and flows out of the junction are considered negative. The sign convention for flow loops is similar: 'clockwise' flows are considered positive in a flow loop, 'counter-clockwise' flows are considered negative. The sign indicating flow direction is combined with the unsigned pipe index to simplify user input; this can be seen in Figure 14 and Table 3. While this encoding simplifies user input, storing both pieces of information (pipe index and flow direction) as a single encoded variable requires that the variable be decoded every time it is used.

4.2 Process

4.2.1 Clarify Code Logic

Five main techniques were used to clarify the intent of the code:

- Document variables
- Document significant sections of code
- Replace conditional expressions with meaningful logical variables
- Replace numeric literals with named constants
- Disambiguate encoded variables

The first two techniques required code and documentation analysis to determine the meaning of each variable and the function of code blocks.

The next technique was used to clarify the termination criteria of iterative calculation loops. An iterative calculation loop terminates if either 1) convergence criteria are met (*e.g.* the magnitude of deviation or correction is below a numerical threshold), or 2) the maximum allowed number of iterations is reached. The

first condition was stored in the logical variable `CONVERGED` and the logical-or of both conditions was stored in the variable `DONE`. This allowed the code to be written with much clearer intent; the iterative calculation loop was rewritten as `do while (.not. DONE)` and convergence was checked after loop termination with `if (.not. CONVERGED)`. The distinction between loop termination with and without convergence is made clear with the addition of only two variables.

Replacing numeric literals with named constants serves two purposes; it imparts meaning to what is otherwise an undocumented value and it ensures consistent usage by setting the value in a single point in the code. In large scientific programs built by multiple developers, it is common to see multiple inconsistent uses of the same constant; π may be variously defined as `3.14`, `3.14159`, or `3.14159265`. Subtle errors may be introduced if the value is mistyped as `3.14195265`. Worse yet is when $\frac{\pi}{4}$ is written as `0.78539816`. In symbolic form, the meaning is clear; in decimal form, meaning is likely lost. To simplify review and code maintainability, it is preferable to migrate numeric literals to named constants any time a constant is used more than once or its meaning is not clear from context.

The final clarification technique is decoding variables which store multiple pieces of information in a single variable. In this case, encoded pipe index/flow convention values were decoded as they were read, storing the pipe index as an unsigned integer and storing the flow direction in a new real variable; `+1.0` indicating entering junction flow or clockwise loop flow, and `+1.0` indicating exiting junction flow or counter-clockwise loop flow. This required code changes to make use of the flow direction variable and to remove decoding of the pipe index. This required careful checking but the resulting code is much simpler; the detection and treatment of flow direction is much more apparent and the pipe index variable can be used directly without needing sign correction.

Should it be desired to use object-oriented techniques in this code suite, the strong coupling between pipe index and flow direction suggest the creation of a Pipe object which contains attributes such as length, diameter, roughness, and flow direction. That is beyond the scope of this refactoring phase but is noted as future consideration.

4.2.2 Identify Common Elements

This software suite is designed to solve problems in steady-state incompressible flow of Newtonian fluids with constant properties so the individual programs are expected to share a number of common elements. Some of the common elements

will be related to the physical systems being modeled, friction factor correlations under different flow regimes being a prime example. Other common elements will include:

- mathematical constants such as π and e,
- physical constants such as unit conversion factors and acceleration due to gravity,
- geometric relations such as circular area and perimeter as a function of diameter, hydraulic radius, and hydraulic diameter, and
- numerical and mathematical functions such as root finders and matrix solvers.

The programs in the suite are meant as educational examples and as a result, programs presented early in the text illustrate simple cases which are extended to create progressively more complex programs. As the code is decomposed into common elements and the underlying structure of the code is made clear, we should expect to find common structures of higher levels of abstraction.

Since all the programs follow a common computational model (a linear, procedural *read-calculate-report* scheme), we should also expect to find common utility elements such as

- read and write formats
- text processing functions such as scanners, tokenizers, and parsers,
- reporting, logging, and error handling elements

Unlike languages like C or C++[9], Fortran has never had a standard library so the most basic utility functions may be duplicated among the programs in this suite.

Finally, as common elements are extracted into separate program units, we will need to verify that newly-created routines are complete and correct. This implies a common testing framework and potentially other common infrastructure. Common infrastructure is an *emergent property* of the overall system; it does not exist in the code at present but will be generated as part of this refactoring process.

[9]Or perhaps any language created since 1970.

4.2.3 Extract Common Elements into Cohesive Program Units

As common elements are identified, they should be extracted into shared program units and organized in a sensible and *cohesive* manner. Cohesion as described in (Moses, 1988) and (Yourdon and Constantine, 1979) is the internal functional relatedness of program units (*i.e.* functions and subroutines). From least to most cohesive, seven levels of relatedness or association are identified:

1. Coincidental association
2. Logical association
3. Temporal association
4. Procedural association
5. Communicational association
6. Sequential association
7. Functional association

Per Yourdon *et al.*, cohesion levels 1-3 are very low cohesion and are discouraged. cohesion levels 4-7 are generally considered acceptable, with functional association being the strongest level of cohesion. Reviewing the forms of cohesion may inform our design decisions when grouping common elements identified in the previous phase of this analysis.

Coincidental association essentially groups program elements for convenience; there is no other relatedness among the program elements.

Logical association groups elements which are in the same logical class. For example, grouping routines which print error messages is a form of logical association.

Temporal association groups elements which occur at the same time in program execution, for example grouping routines which occur during the initialization or shutdown phase of an application would be considered temporal association.

Procedural association groups elements because the are used in a given program unit. For example, it is common to break a long program unit into a sequence of shorter elements. If those elements are grouped together only because they are used by that longer routine, that grouping would be considered procedural association.

Communicational association groups program elements by the data they operate on; that is, the routines are associated by having the same interface.

Sequential association groups elements when each operates on the output of an element and provides input to another element in a sequence of operations. Procedural and sequential association are both related to 'flowchart thinking', the decomposition of a process into a series of sequential steps. The distinction between the two is whether elements are grouped by the containing program element or by the sequence of operations.

Moses defines *functional association* negatively – program elements which are not related by any of the other six forms of association are considered functionally cohesive. I would argue that grouping common program elements which (for example) calculate friction factors under various flow regimes would be functionally cohesive. The routines are logically related but they have a clear focus; they will not be called in all phases of a program's execution but they aren't required to be called in any given phase or order or routine. They may or may not share an interface.

Note that Moses' work was published in 1988, shortly before the release of Fortran 90 which introduced the module, a significant advance in Fortran's ability to group program units and data. Moses refers to functional cohesion as resulting in perfect 'black box' functionality. Program elements are note grouped by any of the other weaker forms of cohesion and there's implications of low coupling and data encapsulation. Further, Moses' discussion of cohesion is meant to address the grouping of program *statements* not program units (functions, subroutines) and there are limits to applying principles of statement cohesion to larger program elements. But that said, the main point is to let the concept of cohesion inform our design choices when extracting common code into subroutines and procedures and secondarily when organizing routines into modules shared among the programs in this suite of applications.

At this point we can only speculate on the which elements will be found in common and how they should be grouped. Final determinations will be made during the upcoming reconnaissance phase but we can identify at least five functionally cohesive groups:

- Compatibility routines for legacy proprietary code
- Numerical routines
- Friction factor routines

- Physical and numerical constants
- Unit testing routines

Beyond that, only code analysis will determine other functional groupings which apply to the common elements found.

4.2.4 Extend the Build Infrastructure to Incorporate Common Dependencies

To avoid maintaining multiple identical program units, the build infrastructure should allow common elements to be compiled a single time and linked to each of the programs which depend upon them, either as compiled object code, static libraries, or dynamic shared objects (DSOs or DLLs). The specifics of the objects generated and the method of linking is not as important as ensuring dependencies are properly built and managed by the current build infrastructure (CMake). Here we are very clear on what outcome we want but are ambivalent on how it is to be achieved. This provides a great deal of flexibility to deal with the uncertainty in configuring CMake to manage dependencies

4.2.5 Verify Shared Dependencies Perform as Expected

Two methods will be used to verify correct operation of shared dependencies. First, we can test the applications as in previous refactoring phases to ensure the results are acceptably close to the original reference results. This implicitly tests dependencies through *integral testing*. Further, we can produce standalone test applications which verify the results of individual library routines in isolation by the method of *unit testing*. Both integral and unit testing will be used to verify that both the parts and the whole are performing as desired.

Testing is part of a larger verification and validation (V & V) effort. *Verification* ensures the code's functional requirements satisfy its design goals. *Validation* ensures the code meeds its functional requirements. Or more colloquially, validation asks "*is this the right code?*", verification asks *"is the code is right?"*.

In an ideal greenfield project, software requirements would be solicited and divided into 'functional' and 'non-functional' categories. Both sets of requirements would be translated into specifications to define what to software to build, and the functional requirements/specifications would define the testing necessary to show the program correctly and completely implements its specifications. This is a simplified idealization of the development process. In this case, there are no formal

requirements or specifications, so those must be inferred from the existing software. Without a functional specification, we cannot construct a test plan which verifies the code. The best we can do at this stage is to perform *regression testing* to verify the code's behavior is consistent with the original behavior. We do not expect results to be numerically identical to previous versions but the results should be reasonably close and deviations should be explicable.

We can begin to reconstruct a functional specification by defining requirements and specifications for the common elements isolated into new program units and modules. Those allow us to create a formal test plan which will define the unit and integral testing to verify each new module. We may wish to start reconstructing the software's *design basis* (underlying goals, requirements, and specifications) during this refactoring phase and complete it a later refactoring effort.

4.3 Reconnaissance

Review of introductory material in chapters 1 - 3 of (Jeppson, 1974) and (Jeppson, 1976) provided a number of pipe flow friction factor correlations which we see are used throughout the flow analysis applications. Both the Darcy-Weisbach and Hazen-Williams friction factor correlations are used in the pipe flow calculation applications and scheduled for extracted into the `jfriction` module. Note that the Darcy-Weisbach friction factor is determined by different correlations depending on the flow regime which may be determined by Reynolds number or a separate figure of merit used to distinguish transition turbulent flow from wholly-rough-pipe flow. Calculations for both the Reynolds number and this figure of merit were also added to the `jfriction` module. Defining the logic to determine the flow regime allowed the Darcy-Weisbach friction factor to be calculated at a higher level of abstraction which greatly shortened and clarified `JEPPSON_CH2`, reducing it to a simple 5 step sequence; see Table 5.

Most applications are concerned with pipe flow and the flow area is simply the circular area of the inner diameter of the pipe. The area function was extracted as a utility function to `jutil` and π was extracted to the `jconstants` module.

4.4 Products and Deliverables

4.4.1 Modules

New modules were added to the code to isolate common elements. These include:

Table 5: `JEPPSON_CH2.f90` after refactoring common elements into modules

```
!> Darcy-Weisback friction factor solver
program JEPPSON_CH2
    use, intrinsic :: iso_fortran_env, only: STDIN => input_unit,       &
        STDOUT => output_unit
    use jfriction, only: f_darcy_weisbach
    use jutil, only: circ_area

    implicit none

    ! Flow regime selection metric above which flow is turbulent-rough
    real, parameter :: PARLIM = 100.0
    ! Maximum number of iterations in friction factor root finder
    integer, parameter :: MAXITER = 15
    ! Minimum difference to continue iterating in flow solver
    ! (numerical tolerance)
    real, parameter :: MAXDIF = 1.0E-5

    ! Pipe diameter, ft
    real :: D
    ! Absolute roughness of pipe, ft
    real :: E
    ! Darcy-Weisback friction factor
    real :: F
    ! Length of pipe, ft
    real :: FL
    ! Acceleration of gravity, ft/s**2
    real :: G
    ! Head loss, ft
    real :: HL
    ! Volumetric flow rate, cfs
    real :: Q
    ! Bulk velocity, ft/s
    real :: V
    ! Kinematic viscosity of fluid (nu)
    real :: VIS

    ! Read format
100 format(6F10.5)
    ! Write format
101 format ('Q = ', F10.4, ' cfs, D = ', F10.4, ' ft, L = ', F10.2,     &
            ' ft, F = ',  F10.5, ', Head loss = ', F10.4, ' ft')

    continue

    do
        ! 1) Read flow conditions and pipe geometry
        ! D   - Pipe diameter, ft
        ! Q   - Flow rate, cfs
        ! FL  - Length of pipe, ft
        ! VIS - Kinematic viscosity of fluid (nu)
        ! E   - Absolute roughness of pipe, ft
        ! G   - Acceleration of gravity, ft/s**2
        read(STDIN, 100, end=99) D, Q, FL, VIS, E, G

        ! 2) Calculate friction factor
        F = f_darcy_weisbach(Q, D, E, VIS, PARLIM, MAXITER, MAXDIF)

        ! 3) Calculate bulk flow velocity
        V = Q / circ_area(D)

        ! 4) Calculate head loss
        HL = F * FL * V * V / (2.0 * G * D)

        ! 5) Display results
        ! Q  - Flow rate, cfs
        ! D  - Pipe diameter, ft
        ! FL - Length of pipe, ft
        ! F  - Darcy-Weisbach friction factor
        ! HL - Head loss along pipe, ft
        write(STDOUT, 101) Q, D, FL, F, HL
    end do

99  continue

    stop
end program JEPPSON_CH2
```

- `jfriction` - pipe flow friction factor correlations and related functions,
- `jconstants` - physical and numerical constants and unit conversion factors,
- `alfc_sperry_mathpack` - legacy Fortran compatibility libraries for proprietary UNIVAC MATH-PACK routines,
- `jutil` - common routines which aren't otherwise categorized (contains a single routine to calculate the area of a circle to determine flow area in a filled circular pipe), and
- `m_ftncheck` - Unit testing framework based on `ftncheck` from Arjen Markus' `FLIBS` Fortran utility routine library[10].

4.4.2 Programs

All programs from the previous modification phase were retained and the program `test_jfriction` was added as an illustration of a test driver application which runs unit tests on the `jfriction` module routines; see section 4.4.4. Prototype functions for the `jfriction` library were developed in the Jupyter Notebook under `JEPPSON_CH2/ipynb/Frictional Losses.ipynb`; see section 4.4.4.

4.4.3 Build Infrastructure

The CMake build infrastructure was heavily modified to track the dependencies of each application, compile each dependency once and independently, and ensure each application links with the appropriate dependencies. This breaks compatibility with the existing Makefiles and additional build documentation is provided to help users manually build the software if CMake isn't available. The static Makefile build infrastructure should be updated to handle dependencies or deprecated in favor of CMake.

The biggest change to the CMake configuration was the definition of a common location for storing object (`*.o`) and module files (`*.mod`) of dependencies and allowing for builds of the entire jeppson_pipeflow project as well as builds of individual projects within it.

[10] https://flibs.sourceforge.net/

4.4.4 Test Infrastructure

Example unit tests and a driver utility are provided for the `jfriction` module to illustrate on method of unit testing in a modern Fortran application. There are other methods which can be used but the one selected illustrates the essential functionality needed. As a practical convenience, the Test Anything Protocol (TAP; see `https://testanything.org/`) is implemented in a limited fashion, showing how unit testing results can be presented in a manner which allows *continuous integration* and the automated analysis of test results. Showing that programs pass verification tests at any point in the code's development simplifies release and development and catches errors early in the development cycle. This moves the role of testing closer to the developer, simplifying the role of the test engineer and freeing test engineers to focus on the more cognitively demanding act of software validation. This shortens the release cycle and improves overall system quality.

The Test Anything protocol was originally developed for testing Perl modoules and was found to be a straightforward way to communicate the results of automated testing. Table 6 shows the results of two tests: the number of individual tests run, the pass/fail (ok/not ok) state of each test along with it's serial ID, and some diagnostic comments.

At it's core, a TAP stream consists of a *plan*, a line showing the number of tests which are intended to be run, and one line per test showing the results of that test, beginning with `ok` if the test passed or `not ok` if the test failed. Comments (prefaced with #) can be placed almost anywhere. TAP is very flexible and has more capabilities but the essential notion is that it communicates the count and status of tests in a simple human- and machine readable form.

As noted previously, the `ftncheck` unit testing framework from the FLIBS library was modified and used as the core of the standalone test application for the `jfriction` module. Tests of individual routines in the `jfriction` module are defined in the `jtest` module; as an example, the unit test of the laminar friction factor correlation is shown in Table 7. This routine generates the output seen in lines 4 through 29 of Table 6.

Functions to add TAP plans and test results are have been added to the `ftncheck` framework along with type- and rank-dependent functions for integer and real comparisons which implement the generic function interfaces `assert_equal` and `assert_comparable`. Note that the `fortran-testanything` library (see `https://github.com/dennisdjensen/fortran-testanything`) already provides this functionality however it has difficulty compiling under the Intel Fortran compiler `ifort` so it was decided to add minimal TAP support to `ftncheck` rather than

Table 6: Unit test results in TAP format

```
#
# Running all unit tests.
#
# 1. # Testing f_laminar()
# Test: jfriction:f_laminar()
1..22
ok - Test 1: Laminar friction factor for Re =   1.0000E+00 =   6.4000E+01
ok - Test 2: Laminar friction factor for Re =   1.4659E+00 =   4.3659E+01
ok - Test 3: Laminar friction factor for Re =   2.1489E+00 =   2.9782E+01
ok - Test 4: Laminar friction factor for Re =   3.1502E+00 =   2.0316E+01
ok - Test 5: Laminar friction factor for Re =   4.6179E+00 =   1.3859E+01
ok - Test 6: Laminar friction factor for Re =   6.7695E+00 =   9.4542E+00
ok - Test 7: Laminar friction factor for Re =   9.9235E+00 =   6.4493E+00
ok - Test 8: Laminar friction factor for Re =   1.4547E+01 =   4.3995E+00
ok - Test 9: Laminar friction factor for Re =   2.1325E+01 =   3.0012E+00
ok - Test 10: Laminar friction factor for Re =   3.1261E+01 =   2.0473E+00
ok - Test 11: Laminar friction factor for Re =   4.5826E+01 =   1.3966E+00
ok - Test 12: Laminar friction factor for Re =   6.7177E+01 =   9.5271E-01
ok - Test 13: Laminar friction factor for Re =   9.8476E+01 =   6.4990E-01
ok - Test 14: Laminar friction factor for Re =   1.4436E+02 =   4.4334E-01
ok - Test 15: Laminar friction factor for Re =   2.1162E+02 =   3.0243E-01
ok - Test 16: Laminar friction factor for Re =   3.1022E+02 =   2.0631E-01
ok - Test 17: Laminar friction factor for Re =   4.5475E+02 =   1.4074E-01
ok - Test 18: Laminar friction factor for Re =   6.6663E+02 =   9.6005E-02
ok - Test 19: Laminar friction factor for Re =   9.7723E+02 =   6.5491E-02
ok - Test 20: Laminar friction factor for Re =   1.4325E+03 =   4.4676E-02
ok - Test 21: Laminar friction factor for Re =   2.1000E+03 =   3.0476E-02
ok - Test 22: Array calculation of laminar friction factors
# Test plan conpleted with 22 tests.
# 2. # Testing f_blasius()
# Test: jfriction:f_blasius()
1..22
ok - Test 1: Blasius friction factor for Re =   2.1000E+03 =   4.6680E-02
ok - Test 2: Blasius friction factor for Re =   2.5475E+03 =   4.4480E-02
ok - Test 3: Blasius friction factor for Re =   3.0903E+03 =   4.2383E-02
ok - Test 4: Blasius friction factor for Re =   3.7487E+03 =   4.0385E-02
ok - Test 5: Blasius friction factor for Re =   4.5475E+03 =   3.8481E-02
ok - Test 6: Blasius friction factor for Re =   5.5165E+03 =   3.6667E-02
ok - Test 7: Blasius friction factor for Re =   6.6920E+03 =   3.4938E-02
ok - Test 8: Blasius friction factor for Re =   8.1179E+03 =   3.3291E-02
ok - Test 9: Blasius friction factor for Re =   9.8476E+03 =   3.1722E-02
ok - Test 10: Blasius friction factor for Re =   1.1946E+04 =   3.0226E-02
ok - Test 11: Blasius friction factor for Re =   1.4491E+04 =   2.8801E-02
ok - Test 12: Blasius friction factor for Re =   1.7579E+04 =   2.7443E-02
ok - Test 13: Blasius friction factor for Re =   2.1325E+04 =   2.6150E-02
ok - Test 14: Blasius friction factor for Re =   2.5869E+04 =   2.4917E-02
ok - Test 15: Blasius friction factor for Re =   3.1381E+04 =   2.3742E-02
ok - Test 16: Blasius friction factor for Re =   3.8068E+04 =   2.2623E-02
ok - Test 17: Blasius friction factor for Re =   4.6179E+04 =   2.1556E-02
ok - Test 18: Blasius friction factor for Re =   5.6019E+04 =   2.0540E-02
ok - Test 19: Blasius friction factor for Re =   6.7955E+04 =   1.9572E-02
ok - Test 20: Blasius friction factor for Re =   8.2435E+04 =   1.8649E-02
ok - Test 21: Blasius friction factor for Re =   1.0000E+05 =   1.7770E-02
ok - Test 22: Array calculation of Blasius friction factors
# Test plan conpleted with 22 tests.
#
# Completed all unit tests.
#
```

modifying `fortran-testanything` to build under `ifort`.

The unit testing frameworks available for Fortran are not nearly as robust or mature as those available for comparable languages. Local modification is often necessary to support one's specific toolchain (platform, compiler, build framework, *etc.*).

Note that `test_f_laminar` calls `f_laminar` with both scalar and array arguments. This verifies that `f_laminar` behaves correctly as an `elemental` function and shows the power of implementing `assert_comparable` as a generic interface and type- and rank-specific implementation functions. The underlying complexity of the function's implementation is not apparent from the caller's perspective.

Part of developing the test cases was generation of test data. Note that lines 6-13 of Table 7 give logarithmically-spaced values for Reynolds number `re` over the interval [1.0 .. 2100.0] and corresponding reference values in lines 15-22 for `f_expected`. These data were generated using a Jupyter Notebook using Python 2 and the NumPy numerical analysis library. Many of the correlations in the `jfriction` library were first prototyped in Python using the Jupyter Notebook environment and the resulting code was translated to Fortran. Test data was rapidly and independently[11] generated in Python and used to verify the Fortran.

5 Insights

Initial recovery of the code took approximately 1 week. The initial challenge was in removing errors introduced by the optical character recognition process and secondarily, correcting errors in the original typeset code listings. The first program in Chapter 6 presented a substantial challenge due to the presence of variables `II` and `I1` which made proofreading difficult. Manually checking the test cases for correctness was also an unexpected difficulty but confirmed the proper functioning of the software.

Once the CMake build system was configured properly, the software build process became trivial. Despite having copious documentation, understanding how CMake operates and how to properly configure it was difficult, often leading to blind trial and error to resolve issues. The underlying logic or model of CMake is not apparent. It was selected as the most popular of the cross-platform build systems; see also `waf`, `scons`. The modular nature of the source tree complicates the build process under CMake; while the current build configuration works, it

[11] Independence is limited since a single author developed the both the prototype and production code.

Table 7: Unit test of `f_laminar` function

```
!> Laminar friction factor test, Re in [1.0 .. 2100.0]
subroutine test_f_laminar()
    use jfriction, only: f_laminar
    real, parameter :: EPS = 1.0E-3
    integer, parameter :: NTESTPTS = 21
    real, dimension(NTESTPTS), parameter :: re = [                      &
        1.00000000e+00,   1.46592234e+00,   2.14892832e+00,             &
        3.15016204e+00,   4.61789293e+00,   6.76947243e+00,             &
        9.92352089e+00,   1.45471110e+01,   2.13249351e+01,             &
        3.12606988e+01,   4.58257569e+01,   6.71770011e+01,             &
        9.84762669e+01,   1.44358560e+02,   2.11618439e+02,             &
        3.10216198e+02,   4.54752857e+02,   6.66632374e+02,             &
        9.77231292e+02,   1.43254519e+03,   2.10000000e+03 ]

    real, dimension(NTESTPTS), parameter :: f_expected = [              &
        6.40000000e+01,   4.36585200e+01,   2.97822870e+01,             &
        2.03164152e+01,   1.38591347e+01,   9.45420794e+00,             &
        6.44932385e+00,   4.39949897e+00,   3.00118147e+00,             &
        2.04729908e+00,   1.39659450e+00,   9.52707012e-01,             &
        6.49902784e-01,   4.43340526e-01,   3.02431113e-01,             &
        2.06307731e-01,   1.40735784e-01,   9.60049384e-02,             &
        6.54911488e-02,   4.46757286e-02,   3.04761905e-02 ]

    real, dimension(NTESTPTS) :: f_calculated
    integer :: i
    integer :: nplan
    integer :: icurrtest
    character (len=80) :: tlabel

10  format('Test ', I0, ': Laminar friction factor for Re = ', ES12.4,  &
           ' = ', ES12.4)
20  format('Test ', I0, ': Array calculation of laminar friction ',     &
           'factors')
    continue

    nplan = NTESTPTS + 1

    call tap_start_plan(nplan)

    icurrtest = 0

! Single call
    f_calculated = -1.0
    do i = 1, NTESTPTS
        ! TODO: Refactor to generic test routine (next test ID)
        icurrtest = icurrtest + 1
        f_calculated(i) = f_laminar(re(i))
        write(tlabel, 10) icurrtest, re(i), f_expected(i)
        call assert_comparable(f_calculated(i), f_expected(i), EPS,     &
            tlabel)
    end do

! Array call - tests if 'elemental' works
    f_calculated = -1.0
    f_calculated = f_laminar(re)
    ! TODO: Refactor to generic test routine (next test ID)
    icurrtest = icurrtest + 1
    write(tlabel, 20) icurrtest
    call assert_comparable(f_calculated, f_expected, EPS, tlabel)

    call tap_finish_plan(icurrtest, nplan)

    return
end subroutine test_f_laminar
```

is not necessarily optimal and the difficulty in understanding CMake limits any effort to improve the build process for fear of inadvertently breaking the build configuration. The choice of build system should be reevaluated in the future to see if an alternate cross-platform build utility would be more maintainable.

Programs from chapters 2, 4, and 7 were rather straightforward to recover. The programs from chapters 5 and 6 required writing an argument-compatible replacement for the Sperry MATH-PACK routine `GJR`. This required experimentation with the BLAS and LAPACK libraries, specifically in data arrangement and the use of modern Fortran's `reshape` intrinsic function.

Removal of `GOTO`s and other archaic control structures simplified code analysis, allowing further changes which greatly clarified the code. By constraining the scope of a refactoring evolution into a well-defined 'phase', code changes are easier to test and validate. As noted in section 3.2, structure may appear after refactoring which was not apparent in the original code. By segmenting a refactoring effort into a number of small evolutions rather than a single large project, emergent work can be deferred to a later evolution without affecting the current schedule and effort commitments. This also gives developers the time necessary to analyze the code and devise a refactoring plan, rather than adding unplanned work to the current schedule.

A vital part of any refactoring effort is building a test suite and improving its effectiveness. This project shows the value of both integral and unit testing. Strongly coupled code is difficult to unit test; the test cases are larger and the source of an error is not always apparent. The effect of unit testing is improved if code can be refactored into loosely-coupled routines with results which only depend on arguments passed in. This does not reduce the need for integral testing but it allows routines to be tested in isolation which is especially important when building reusable libraries.

There are still substantial deficiencies in the scope and depth of testing for this software suite. However, each program has at least one test case with reference data and several functions within the `jfriction` module have unit tests, providing a solid basis for extending both unit and integral tests. A further code revitalization evolution could be devoted to improving the test suite and testing architecture. For example, the CMake build system has an automatic testing feature called CTest. This could be leveraged to allow automated testing of new builds to allow *continuous integration*, incorporating testing into development practice, rather than segregating testing from development. Regardless of the development methodology used – waterfall, agile, *etc.* – reducing the cost of testing increases the frequency of testing and reduces the time needed to detect and correct errors.

Few specialty or custom software tools were needed in refactoring the Jeppson software suite. A text editor, revision control system, modern Fortran compiler, `make`, CMake, and the Doxygen documentation generator were the primary tools used to revitalize the software. The `findent` source code reformatter was developed recently and has proven very capable as refactoring tool since it can operate on both fixed and free-format Fortran code. Custom software was developed to parse gfortran error output to create Fortran variable declarations using the unix utilities `grep`, `sed`, and `sort`. This could have easily been done with Python, Perl, or any language with regular expression and sorting libraries. Jupyter Notebook and Pythonwas used to verify the

The Eclipse development environment and the Photran plugin would have helped with a number of refactoring tasks but like many IDEs, Photran has a bias toward creating a single executable per project which did not match the multiple executable architecture of the Jeppson software suite. Further, integration between CMake and Eclipse is not straightforward so the decision was made to not use Eclipse. This is unfortunate since Eclipse is cross-platform and has good support for the Mercurial revision control system, the Doxygen code documentation system, and for Fortran via the Photran plugin.

Had this project involved a single executable, it would have been ideal for maintaining in Eclipse. Fortunately it was small enough to maintain manually via CMake. Automatic refactoring is almost always preferable to manual refactoring due to its speed and consistency, especially for tedious evolutions such as implementing `IMPLICIT NONE` (*i.e.* requiring explicit variable declaration). Since the software suite is composed of six monolithic programs with references to a single library routine, it was felt that the overhead of building multiple linked projects in Eclipse would have required too much ongoing effort to be spent on overhead compared to the one-time effort expenditure spent configuring CMake. This is sort of a hindsight tradeoff; without an understanding of the benefits and limitations of Eclipse and CMake, a different decision may have been appropriate.

This points out a theme of software revitalization - *a major problem is not knowing what we don't know!*. A phased approach of refactoring with clearly defined goals, outcomes, and deliverables and a willingness to constrain or defer goals to complete a refactoring iteration defends the project against scope creep and maintains project inertia. This prevents emergent issues from dominating the development plan and allows planned work to be completed. Measurable progress is made while allowing for adjustments to the project plan and goals as new information becomes available. Automated build and testing allow iterations to be completed in a timely manner, reducing the tendency toward fewer and larger re-

leases as issues emerge. This admits the reality that software project estimation is not an exact and predictable science and reduces schedule risk by setting clear and reasonable expectations at each phase of the project. By detecting emergent issues earlier in the project, disruption is limited to individual phases of the project rather than buried in the entire project. The overall project should become more manageable and development should become more predictable as a result.

Software, Documentation, and Online Resources

- `jeppson_pipeflow` project archive - `https://bitbucket.org/apthorpe/jeppson_pipeflow`
- Link to (Jeppson, 1974); source document for project and code. See also (Jeppson, 1976) - `https://digitalcommons.usu.edu/water_rep/300/`
- Acorvid Legacy Fortran Compatibility Library - archive: `https://bitbucket.org/apthorpe/alfc` ; introduction: `http://www.acorvid.com/2018/01/20/introducing-the-acorvid-legacy-fortran-compatibility-library/`
- LAPACK linear algebra package - `http://www.netlib.org/lapack/`
- Markdown documentation - `https://daringfireball.net/projects/markdown/syntax`
- CMake build automation system - `https://cmake.org/`
- waf build automation system - `https://waf.io/`
- scons build automation system - `http://scons.org/`
- Doxygen source code documentation system - `http://www.stack.nl/~dimitri/doxygen/`
- PlusFORT Fortran development and refactoring toolkit - `http://www.adeptscience.co.uk/products/fortran-tools/plusfort-with-spag/plusfort-version-6.html`
- `findent` Fortran source code reformatter - `https://sourceforge.net/projects/findent/`
- Jupyter Notebook interactive computing environment - `http://jupyter.org/`
- FLIBS Fortran utilities - `https://flibs.sourceforge.net/`
- Numdiff numerically-aware file difference checker - `http://www.nongnu.org/numdiff/`
- Test Anything Protocol (TAP) - `https://testanything.org/`

- TAP provider for Fortran - `https://github.com/dennisdjensen/fortran-testanything`
- Photran plug-in for Eclipse IDE - `https://www.eclipse.org/photran/`
- Link to (Albert and Whitehead, 1986) - `http://www.dtic.mil/dtic/tr/fulltext/u2/a170611.pdf`
- Link to (Ding and Kennedy, 1982) - `http://www.dtic.mil/dtic/tr/fulltext/u2/a110089.pdf`

References

Albert, W. G. and L. K. Whitehead
1986. Mathematical and statistical software index: Second edition. Technical Report AFHRL-TP-85-47, Manpower and Personnel Division, Brooks Air Force Base.

Clerman, N. and W. Spector
2011. *Modern Fortran: Style and Usage*. Cambridge University Press.

Ding, L. and R. M. Kennedy
1982. A numerical treatment of the dynamic motion of a zero bending rigidity cylinder in a viscous stream. Technical Report 6343A, Naval Underwater Systems Center.

Fowler, M. and K. Beck
1999. *Refactoring: Improving the Design of Existing Code*, Component software series. Addison-Wesley.

Hollnagel, P. E.
2014. *Safety-I and Safety–II: The Past and Future of Safety Management*. Ashgate Publishing Company.

Jeppson, R. W.
1974. Steady flow analysis of pipe networks: An instructional manual. Technical Report 300, Utah Water Research Laboratory.

Jeppson, R. W.
1976. *Analysis of Flow in Pipe Networks*. Ann Arbor, MI: Ann Arbor Scientific Publishers, Inc.

Markus, A.
2012. *Modern Fortran in Practice*. Cambridge University Press.

Metcalf, M., J. Reid, and M. Cohen
2011. *Modern Fortran Explained*. Oxford University Press.

Moses, G. A.
1988. *Engineering Applications Software Development Using FORTRAN 77*. Wiley.

Press, W., S. Teukolsky, W. Vetterling, and B. Flannery
1986. *Numerical Recipes*. Cambridge University Press.

Sperry Rand Corporation
1969. *UNIVAC 1106 system/1108 multiprocessor system STAT-PACK program abstracts*. Sperry Rand Corporation.

Sperry Rand Corporation
1970a. *UNIVAC large scale systems MATH-PACK program abstracts*. Sperry Rand Corporation.

Sperry Rand Corporation
1970b. *UNIVAC large scale systems MATH-PACK programmers reference*. Sperry Rand Corporation.

Sperry Rand Corporation
1970c. *UNIVAC large scale systems STAT-PACK programmers reference*. Sperry Rand Corporation.

Stodt, J. A.
1978. Documentation of a finite element program for solution of geophysical problems governed by the inhomogeneous 2-d scalar helmholtz equation. Technical Report GL04099, University of Utah, Department of Geology and Geophysics.

Yourdon, E. and L. L. Constantine
1979. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Yourdon Press computing series. Prentice Hall.

Revision History

Revision	Date	Description
A	March 7, 2018	Initial draft
0	March 8, 2018	Initial release