Proven Techniques for Efficiently Generating and Testing Software

by Keith R. Wegner
Northrop Grumman Corp.

Generating reliable, error-free software on time and within budget is becoming ever more important as competition increases and procurement budgets shrink. In response, software engineers are continuously striving to develop and implement processes that more efficiently bridge the gap from system analysis and design to embedded systems. This paper presents a proven process that uses advanced, commercially available, MathCAD® and MATLAB® tools to design, develop and test optimal, error-free embedded software.

The MATLAB programming language is quickly and unobtrusively becoming a primary prototype and analysis tool at Northrop Grumman Electronic Sensors & Systems Sector (ESSS) as well as at other engineering entities the world over. Its value to both systems and software engineers has been well demonstrated on many programs at ESSS, including Comanche, and it is especially well-suited to effectively support the corporate Integrated Product Team (IPT) concept. Engineers desiring to be technologically competent and efficient may soon find a solid foundation in MATLAB to be almost indispensable. Its application is becoming pervasive within systems engineering groups, and it is rapidly spreading to other disciplines, such as software, rapid prototyping, and financial analysis. On a similar note, MathCAD is routinely used in the systems engineering disciplines to perform detailed algorithm analysis and development. Although not as well-suited for large, elaborate simulations as MATLAB, this tool excels in performing symbolic manipulations and in developing and documenting detailed mathematical derivations. It is especially renowned for its representation of equations using succinct, precise mathematical notation. Systems engineers are routinely using both MathCAD and MATLAB to develop and capture their design documentation in Mode (MDD) and Function (FDD) Description Documents. Thus, software engineers with a working knowledge of MathCAD and MATLAB will be well positioned to much more efficiently develop software products by bridging the gap from systems MDDs and FDDs to embedded software. These two tools are especially powerful when applied to processes that are inherently mathematical, such as stochastic processes, image and signal processing, inertial navigation, etc.

This paper describes informal techniques developed and successfully applied by the author, within the governing software process, on the Comanche program at ESSS to produce robust, optimal software directly from a systems FDD captured in a MATLAB simulation. In this process, MathCAD was first used to quickly generate the algebraic equivalent of involved Kalman filtering matrix equations expressed in MATLAB. These expanded representations were then coded and tested in MATLAB before being translated into the target Ada programming language using available text-editing tools. MATLAB was extensively used to rigorously unit test the deliverable Ada software product. This deceptively simple process allowed the author to design and develop large, efficient amounts of code in a very short time. Furthermore, the final software was found to be virtually error free, making the successful unit testing of code so developed almost a foregone conclusion.

Background

The Comanche program's Target Acquisition System Software (TASS) contains a number of Computer Software Configuration Items (CSCIs), one of which is the Target Threat Manager (TTM) CSCI. This CSCI maintains tracks of various fidelities on numerous targets detected by assorted sensors from different sources. As such, it contains an association component that attempts to correlate new detections with established track files maintained in a target threat database (TTDB). A number of Kalman filtering algorithms implemented in the Target State Estimator (TSE) component initialize and update entries in the TTDB. These algorithms provide for 3, 6 or 9 states, with independent or partially correlated measurements, depending on the operational mode and the target type. Video TV and FLIR sensors, for example, can be scanned or operated in a 30Hz stare mode, resulting in different measurement dependencies. After performing detailed timing analyses on the target processor hardware, it was determined that the Kalman algorithms should, for example, be optimized by eliminating loops to minimize floating point operations (FLOPs). To this end, an efficient mechanism for optimizing the many mathematical operations inherent in Kalman filtering and coordinate transformation operations was developed.

Initially, the TTM systems engineering design was captured using traditional word processing and drawing tools, such as Microsoft PowerPoint and Microsoft Word. The supporting system simulations were developed in the Ada and C programming languages. However, due to the author's influence, the advantages of MATLAB over Ada and C for developing and maintaining the systems simulations were quickly realized, and a TTM systems simulation was subsequently developed entirely in the MATLAB programming language. It was soon quite obvious that modules of the well-documented MATLAB simulation code, supplemented by other descriptive documents as necessary, were, in effect, equivalent to an FDD. At this point, the author developed the techniques described in this paper to generate optimal, virtually error-free, Ada software directly from the MATLAB simulation/FDD.

Process Description

The overall process, as depicted in Figure 1, is described as follows: Given what in MATLAB is a concise, mathematical expression, use the symbolic capabilities of MathCAD to algebraically expand or "unfold" the operation. This is typically done, for example, to eliminate inefficient looping operations in multiple matrix multiplications or inversions. The expanded MathCAD results are coded in MATLAB. MATLAB is used instead of other higher-order languages because it is interpretative

MathCad is a registered trademark of MathSoft Inc.
MATLAB is a registered trademark of The MathWorks Inc.

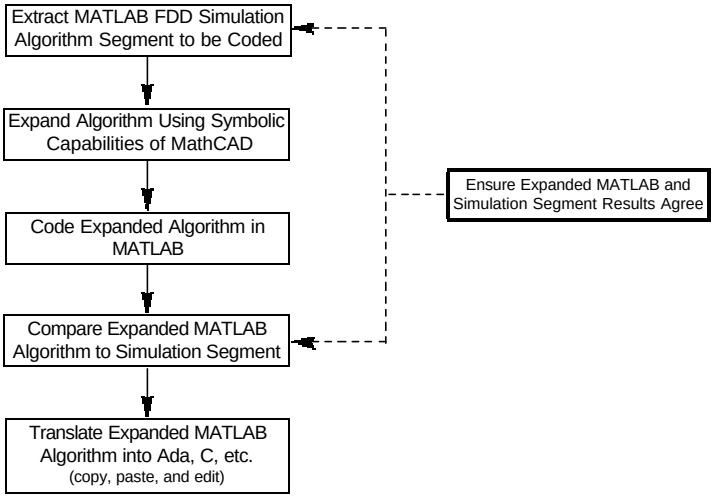


Figure 1. *An Efficient Process Generates Optimal Code*

and results can be quickly confirmed using well-established, embedded routines, and diagnostic techniques are easily applied. Once the expanded algorithm is available in MATLAB, its results are verified by comparison with those of the original, concise code segment from the FDD. Test data, vectors, matrices, etc. are easily generated in MATLAB to expedite this process. When both versions of the algorithm generate equivalent results, thereby validating the expanded algorithm, traditional code editing tools are used to copy the expanded algorithm and paste it into an appropriate file for the target programming language. For the Comanche TTM application, Ada was the target language, but any other higher-order language could have been used. That file is edited to conform to conventions of the target language as well as applicable program coding standards. This final step is the key. Very little new code is generated and perturbations to and modifications of the expanded and already tested MATLAB code are minimized. The bulk of the expanded algorithm is simply tailored to conform to the semantic requirements of the target language. Generally, this involves little more than global find-and-replace operations. For example, if the target language is Ada, the MATLAB "=" is replaced with Ada's "=". Since MATLAB is interpretative, any implicit objects that it uses must be explicitly defined in the target language. Although this overall process may at first seem to be fairly elaborate and perhaps even unnecessary, an attempt to manually optimize even mildly complex 6-by-6, let alone 9-by-9, matrix and vector operations will quickly demonstrate the limitations of pencil and paper. The described process has been found to generate very reliable code, equivalent to the original algorithm segment as specified in the MATLAB FDD.

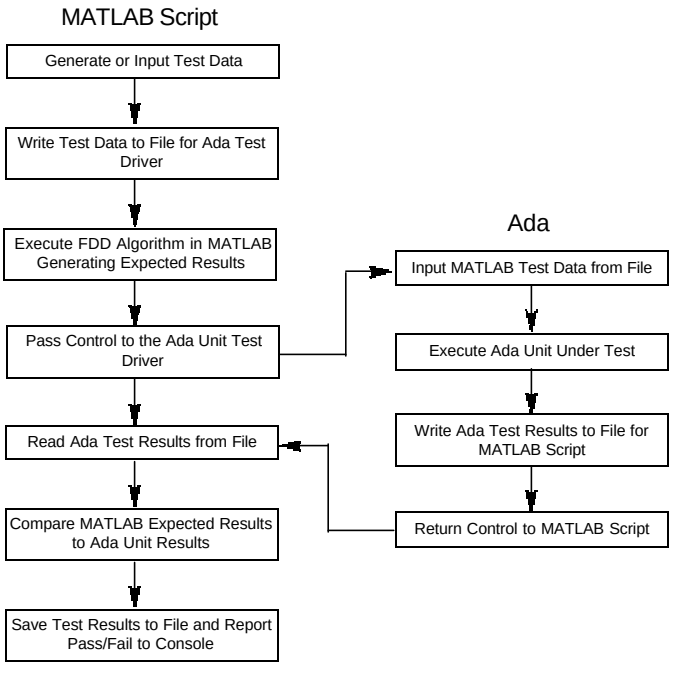
The advantages do not stop there. When it comes to the unit testing of elements generated by following the above process, MATLAB again excels. The author has developed and successfully applied an automated MATLAB-based process, as outlined in Figure 2, to generate repeatable, self-documenting unit tests. To summarize, the unit test process employs a MATLAB script file that executes the succinct code segment copied from the MATLAB FDD to generate expected results. The required unit test data is either generated internally by the MATLAB script or read from a previously prepared input file. That script writes the equivalent unit test data to a file to be input by the Ada unit test driver software. The MATLAB script then calls the executable Ada test driver that reads in the test data and exercises the element under test. When finished, the Ada test driver writes the results generated by the Ada element under test to a file for input by the MATLAB script. The MATLAB script resumes execution, reads in the data from the Ada test driver, and compares that data to its previously generated expected results. Finally, it saves test results to a unit test data file for permanent documentation and reports its findings to the console. The entire sequence is repeatable and can easily be tailored to perform multiple iterations over different random or predefined sets of test data.

Example

To demonstrate the techniques without overwhelming the reader or exceeding the physical capacity of this paper, an example of a portion of a simple 6-by-6 covariance matrix extrapolation extracted from a typical Kalman filtering application is demonstrated. For readability, the addition of the state noise covariance matrix is omitted. The equation to be developed, in any dimension, is $\mathbf{P} = \Phi\mathbf{P}\Phi^T$ where Φ is the state transition matrix, Φ^T is its transpose, **dt** is the sampling interval and **P** the symmetric covariance matrix. This is almost exactly how it appears in the MATLAB simulation. As shown below, the symbolic capabilities of MathCAD are used to generate the algebraic equivalent of this equation. The inherent symmetry of P is exploited in both the MathCAD derivation and the subsequent translation into MATLAB.

This last matrix result, which is symmetric, is now expressed in a MATLAB m-file that compares the expanded solution with the simple $\Phi\mathbf{P}\Phi^T$ matrix product. Although at first this may appear to be a somewhat daunting task, close examination reveals significant symmetry and repetition, except for indices, that greatly simplifies the MathCAD-to-MATLAB translation effort. Following is an example of a MATLAB m-file that implements and verifies the above MathCAD $\Phi\mathbf{P}\Phi^T$ result.

Figure 2. *MATLAB Script Efficiently Unit Tests Ada Element*



$$\Phi P \Phi^T = \begin{bmatrix} 1 & 0 & 0 & dt & 0 & 0 \\ 0 & 1 & 0 & 0 & dt & 0 \\ 0 & 0 & 1 & 0 & 0 & dt \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} p11 & p12 & p13 & p14 & p15 & p16 \\ p12 & p22 & p23 & p24 & p25 & p26 \\ p13 & p23 & p33 & p34 & p35 & p36 \\ p14 & p24 & p34 & p44 & p45 & p46 \\ p15 & p25 & p35 & p45 & p55 & p56 \\ p16 & p26 & p36 & p46 & p56 & p66 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & dt & 0 & 0 \\ 0 & 1 & 0 & 0 & dt & 0 \\ 0 & 0 & 1 & 0 & 0 & dt \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^T =$$

MathCAD's Symbolic Evaluation

$$\begin{bmatrix} p11 + 2 \cdot dt \cdot p14 + dt^2 \cdot p44 & p12 + dt \cdot p24 + dt \cdot p15 + dt^2 \cdot p45 & p13 + dt \cdot p34 + dt \cdot p16 + dt^2 \cdot p46 & p14 + dt \cdot p44 & p15 + dt \cdot p45 & p16 + dt \cdot p46 \\ p12 + dt \cdot p24 + dt \cdot p15 + dt^2 \cdot p45 & p22 + 2 \cdot dt \cdot p25 + dt^2 \cdot p55 & p23 + dt \cdot p35 + dt \cdot p26 + dt^2 \cdot p56 & p24 + dt \cdot p45 & p25 + dt \cdot p55 & p26 + dt \cdot p56 \\ p13 + dt \cdot p34 + dt \cdot p16 + dt^2 \cdot p46 & p23 + dt \cdot p35 + dt \cdot p26 + dt^2 \cdot p56 & p33 + 2 \cdot dt \cdot p36 + dt^2 \cdot p66 & p34 + dt \cdot p46 & p35 + dt \cdot p56 & p36 + dt \cdot p66 \\ p14 + dt \cdot p44 & p24 + dt \cdot p45 & p34 + dt \cdot p46 & p44 & p45 & p46 \\ p15 + dt \cdot p45 & p25 + dt \cdot p55 & p35 + dt \cdot p56 & p45 & p55 & p56 \\ p16 + dt \cdot p46 & p26 + dt \cdot p56 & p36 + dt \cdot p66 & p46 & p56 & p66 \end{bmatrix}$$

```
% Define the number of test iterations and
% an acceptable tolerance for this test
Num_Iterations = 1000 ;
Tolerance      = 1e-014 ;

% Define 3x3 matrices and the sampling interval
I3 = eye(3) ;    % identity
Z3 = zeros(3) ;  % zeros
dt = 1/30 ;      % 30Hz

% Reset generator to initial state for repeatability
rand('state', 0) ;

% Define Phi, the 6x6 state transition matrix
Phi = [I3  I3*dt
       Z3  I3   ] ;

for Iteration = 1:Num_Iterations

  % Define a new random symmetrical 6x6 covariance matrix
  P = rand(6) ;                  % random 6x6
  U = triu(P,1) ;                % upper triangular above
                                   main diagonal
  P = U + U' + diag(diag(P)) ;   % symmetric 6x6

  % Execute the simple form
  flops(0) ;              % reset flops counter
  P_Simple = Phi*P*Phi' ;
  Simple_flops = flops ; % accumulate flops

  % Execute the expanded form
  flops(0) ; % reset flops counter

  dt2 = dt*dt ;
  P_Expanded(1,1) = P(1,1) + 2.0*P(1,4)*dt + P(4,4)*dt2 ;
  P_Expanded(1,2) = P(1,2) + ( P(1,5) + P(2,4) )*dt +
                    P(4,5)*dt2;
  P_Expanded(1,3) = P(1,3) + ( P(1,6) + P(3,4) )*dt +
                    P(4,6)*dt2;
  P_Expanded(1,4) = P(1,4) + P(4,4)*dt ;
  P_Expanded(1,5) = P(1,5) + P(4,5)*dt ;
  P_Expanded(1,6) = P(1,6) + P(4,6)*dt ;

  P_Expanded(2,1) = P_Expanded(1,2) ;
  P_Expanded(2,2) = P(2,2) + 2.0*P(2,5)*dt + P(5,5)*dt2 ;
  P_Expanded(2,3) = P(2,3) + ( P(2,6) + P(3,5) )*dt +
                    P(5,6)*dt2;
  P_Expanded(2,4) = P(2,4) + P(4,5)*dt ;
  P_Expanded(2,5) = P(2,5) + P(5,5)*dt ;
  P_Expanded(2,6) = P(2,6) + P(5,6)*dt ;

  P_Expanded(3,1) = P_Expanded(1,3) ;
  P_Expanded(3,2) = P_Expanded(2,3) ;
  P_Expanded(3,3) = P(3,3) + 2.0*P(3,6)*dt + P(6,6)*dt2 ;
  P_Expanded(3,4) = P(3,4) + P(4,6)*dt ;
  P_Expanded(3,5) = P(3,5) + P(5,6)*dt ;
  P_Expanded(3,6) = P(3,6) + P(6,6)*dt ;

  P_Expanded(4,1) = P_Expanded(1,4) ;
  P_Expanded(4,2) = P_Expanded(2,4) ;
  P_Expanded(4,3) = P_Expanded(3,4) ;
  P_Expanded(4,4) = P(4,4) ;
  P_Expanded(4,5) = P(4,5) ;
  P_Expanded(4,6) = P(4,6) ;

  P_Expanded(5,1) = P_Expanded(1,5) ;
  P_Expanded(5,2) = P_Expanded(2,5) ;
  P_Expanded(5,3) = P_Expanded(3,5) ;
  P_Expanded(5,4) = P_Expanded(4,5) ;
  P_Expanded(5,5) = P(5,5) ;
  P_Expanded(5,6) = P(5,6) ;

  P_Expanded(6,1) = P_Expanded(1,6) ;
  P_Expanded(6,2) = P_Expanded(2,6) ;
  P_Expanded(6,3) = P_Expanded(3,6) ;
  P_Expanded(6,4) = P_Expanded(4,6) ;
  P_Expanded(6,5) = P_Expanded(5,6) ;
  P_Expanded(6,6) = P(6,6) ;

  Expanded_flops = flops ; % accumulate flops

  % Compare approaches
  Diff = P_Simple - P_Expanded ;

  % Ensure MATLAB and Ada implementations are equivalent
  if find(abs(Diff) > Tolerance)
    disp('Expanded implementation is not correct.')
    Diff     % display differences
    keyboard % debug mode on error
  end

  % Ensure expanded implementation enforces symmetry of P
  if find(abs(P_Expanded - P_Expanded') > Tolerance)
   disp('The expanded implementation violates symmetry of P')
    P_Expanded % display the covariance matrix
    keyboard   % debug mode on error
  end

  % Report flop results to console
  disp(['Iteration #', num2str(Iteration)])
  disp(['  Simple flops   = ' num2str(Simple_flops)])
  disp(['  Expanded flops = ' num2str(Expanded_flops)])
  disp(' ')

end % loop
```

MATLAB code to be translated into Target Language

In this sparse matrix example, for each iteration, MATLAB reports 864 FLOPs before the expansion and 49 FLOPs after, for a 17.6 : 1 savings in FLOPs!

Although the above example is quite simple and is almost as easily developed using pencil and paper, one quickly becomes severely entangled in tedious algebra when attempting, for example, to manually derive the Kalman gain matrix from $K = PH^T[HPH^T + R]^{-1}$ for even six states. (Here H is 3×n, P is n×n, R is 3×3 and K is n×3, where *n* is the number of states.) It is for examples similar to this that the power and value of the process are quickly substantiated. This is true even when the capabilities of MathCAD to display symbolic results are exceeded. In such cases, one simply subdivides the process into manageable

portions. In the above Kalman gain equation, for example, intermediate expressions for PH^T are computed first, followed by $[HPH^T + R]$. Then $[HPH^T + R]^{-1}$ is derived and that result premultiplied by the PH^T term that was already expanded. Although this requires somewhat more work due to the maintaining of intermediate results, overall savings and code generation metrics are still very impressive. Depending on the particular implementation and the correlation of the measurements, FLOP savings on the order of 4:1 to 6:1 and more have been tabulated. These savings were achieved with significantly less effort and higher reliability than the manually tedious and error-prone traditional approach.

A process similar to the above is then used to unit test the algorithm in the target language. A test driver is created that executes the unit under test. Its executable image is called in place of the expanded code bracketed in the example m-file above. The MATLAB script and the target language test driver must both provide for the reading and writing of data files to effect the transfers of test data and results. The author has elected to mechanize these transfers in IEEE 64-bit binary format to take full advantage of the numeric capabilities of MATLAB. An outline of the MATLAB script was shown in Figure 2.

Although applications vary enormously in complexity, the author has experienced estimated savings from 40 to almost 80 percent in development and testing time (including debugging effort). For example, it took approximately 20 hours to manually develop and test code to decrement a 6-state covariance matrix and about 16 hours for the 6-state Kalman gain matrix. Applying the techniques outlined in this paper reduced these efforts to approximately 12 and eight hours, respectively. Without manually deriving corresponding 9-state equations, a fairly daunting task, extrapolations from actual 6-state manual results and other similar experience were used to estimate the level of effort that would be required for the manual derivation of 9-state equations. The following table summarizes and compares both approaches.

	Pencil & Paper	MathCAD & MATLAB	Estimated Savings
Decrement Covariance $[I-KH]P[I-KH]^T + KRK^T$			
6-State	20	12	40.0%
9-State	56*	16	71.4%
Compute Kalman Gain $PH^T[HPH^T + R]^{-1}$			
6-State	16	8	50.0%
9-State	42*	10	76.2%

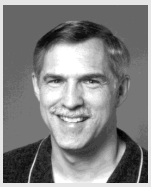
* Extrapolated from similar 6-State experience

Table 1. *Estimated Effort to Code and Test (hours)*

Summary

The use of commercial tools, such as MATLAB and MathCAD, can dramatically improve the efficiency of the software development process as well as the reliability of the final embedded product. Since systems engineering groups are increasingly using these tools to develop and document their products, it is becoming imperative that software engineers acquire the necessary proficiency to use and integrate these products into their processes. This is especially critical as contractors strive to remain competitive while meeting exigent schedules and maintaining budgetary constraints. Although not a panacea for all that is ailing in the software development process, the logical application of available commercial tools can have a significant, positive impact on that effort. This paper outlines such a process that the author has successfully applied on a current development program.

About the Author



Keith R. Wegner is a Fellow Engineer in the software engineering group at Northrup Grumman Corporation's Electronic Sensors and Systems Sector in Baltimore. He has a master's degree in electrical engineering from the Johns Hopkins University with emphasis in signal processing and control systems. He has used MATLAB for approximately 14 years.

Northrup Grumman Corp.
P.O. Box 746, MS-429
Baltimore, Md. 21203
Voice: 410-765-4664
Fax: 410-765-1492
E-mail: keith_r_wegner@mail.northgrum.com

▸▸Statistical Process Control Meets Earned Value, by Lipke/Vaughn, continued from page 20▸▸

References

1. Florac, William A., and Anita D. Carleton, *Measuring the Software Process*, Addison Wesley, Reading, Mass., 1999.
2. Pitt, Hy, *SPC for the Rest of Us*, Addison-Wesley, Reading, Mass., 1995.
3. Software Engineering Institute Course, Statistical Process Control for Software, July 1999.
4. Software Productivity Consortium Course, Statistical Process Control and Quality Management Techniques, August 1998.
5. Radice, Ron, Statistical Process Control for Software Projects, 3rd Annual Software Metrics Conference, December 1997.
6. Fleming, Quentin W., *Cost/Schedule Control Systems Criteria, The Management Guide to C/SCSC,* Probus, Chicago, 1988.
7. Lipke, Walter H., Applying Management Reserve to Software Project Management, CrossTalk, March 1999.
8. Crow, Edwin L., Davis, Francis A., Maxfield, Margaret W., *Statistics Manual*, Dover Publications, New York, 1960.

Notes

1. To remove any confusion, by monthly performance values, we mean the values include only the performance occurring during the month.
2. The application of Table 1 in [7] required CPI^{-1} and SPI^{-1} to be cumulative values. For this application, CPI^{-1} and SPI^{-1} are average values of the monthly data.
3. The assumption in overtime and staffing equations is that the plan is being executed; i.e., the overtime rate and the staffing employed is in agreement with the project plan. If the effective values for either differ from the plan, it is recommended to use those values in the equations.
4. See the Table 1 management action description for the condition: cost comparison green, schedule comparison red.

About the Authors

Walt Lipke is the deputy chief of the Software Division at the Oklahoma City Air Logistics Center, which employs approximately 600 people, most of whom are electronics engineers. He has 30 years of experience in the development, maintenance, and management of software for automated testing of avionics. In 1993, with his guidance, the Test Program Set and Industrial Automation (TPS and IA) functions of the division became the first Air Force activity to achieve Software Engineering Institute Capability Maturity Model (SEI CMM®) Level 2. In 1996, these functions became the first software