APPENDIX A

A.1 Observer Based Approaches

The basic idea of the observer based approach is to reconstruct the outputs of the system from the measurements or subsets of the measurements with the aid of observers using the estimation error as a residual for the detection and isolation of the faults.

A 1.1 Luenberger Observer for Linear Systems

The fundamental configuration of a linear full order state estimator is shown in Figure A 1.1.

![Luenberger Observer](image)

Figure A.1.1: Structure of Luenberger observer
The full order observer simply consists of a parallel model of the process with a feedback of the estimation error, \( e = y - \hat{y} \). Though in principle the open-loop model would do, the feedback is important for several reasons.

1) To compensate for differences in the initial conditions.

2) To stabilize the parallel model in case of an unstable system.

3) To provide freedom for the design of the filter, for example, to decouple the effects of faults from the effects of other faults or unknown inputs.

Consider the linear system defined by the state equations

\[
\begin{align*}
  x(t) &= Ax(t) + Bu(t) + Ed(t) + Kf(t) \quad \text{(A1)} \\
  y(t) &= Cx(t) + Fd(t) + Gf(t) \quad \text{(A2)}
\end{align*}
\]

Where \( x \) is the \( nx1 \) state vector, \( u \) is the \( px1 \) known input vector, \( y \) is the \( qx1 \) vector of measured outputs and \( A, B \) and \( C \) known matrices of appropriate dimensions. The term \( Ed \) models the unknown inputs to the actuators and to the dynamic process. \( Kf \) represents actuator and component faults. \( Fd \) the unknown inputs to the sensors and \( Gf \) sensor faults. The state \( \hat{x} \) and output \( \hat{y} \) of a full order observer are governed by the equations.

\[
\begin{align*}
  \dot{x} &= (A - HC)\hat{x} + Bu + Hy \quad \text{(A3)} \\
  \dot{y} &= C\hat{x} \quad \text{(A4)}
\end{align*}
\]

Where \( H \) denotes the feedback gain matrix that has to be chosen properly to achieve a desired performance of the observer. The relations for the state estimation error, \( \xi = x - \hat{x} \) and the output estimation error, \( e = y - \hat{y} \), become

\[
\begin{align*}
  \dot{\xi} &= (A - HC)\xi + Ed + Kf - HFd - HGf \quad \text{(A5)} \\
  e &= C\xi + Fd + Gf \quad \text{(A6)}
\end{align*}
\]
It is seen from equation A5, and A6 that the output estimation error \( e \) is a function of \( f \) and \( d \) (but not of \( u \)). Hence, \( e \) can be used as the residual \( r \), for the purpose of detection and isolation of the fault. When no fault occurs, i.e. \( f=0 \), then \( r \) only will be influenced by the unknown input, \( d \). When however \( f \neq 0 \), \( r \) will be increased. Thus a fault can be detected by checking the increment of \( r \) caused by \( f \). In the simplest case this can be done by a threshold logic. To avoid false alarms, the threshold must be chosen larger than zero, though this reduces the sensitivity to faults.

### A1.2 Single Observer Scheme

The most simple configuration used for instrument fault detection is a single estimator (observer or Kalman filter), where a single full or reduced order estimator is driven by only one (the most reliable) sensor output, and the full output is reconstructed Ref [40]. The comparison of the actual output \( y \) with the estimated output \( \hat{y} \) using threshold logic, allows in principle a unique detection and isolation of a single faulty instrument.

![Figure A1.2: Instrument fault detection (IFD) using a single estimator](image)
Mehra and Peshon, Ref [60], used a single Kalman filter driven by the full output vector as shown in Fig. A.1.2 and make use of the fact that the innovation is white noise with zero mean and known covariance when no fault occurs. The occurrence of a fault is monitored by statistical innovation tests of whiteness, mean and covariance. The multiple hypothesis testing for example can be carried out using Bayesian decision theory.

A. 1.3 : Dedicated Observer Scheme

More flexibility in the isolation of actuator, component or sensor faults can be achieved by using an estimator scheme, i.e. a bank of estimators driven by the actual output vector, \( \mathbf{y} \), or subsets of \( \mathbf{y} \) as shown in Fig A1.3. A popular approach is based on multiple hypothesis testing, Ref 1. In this case, each of the estimators is designed for a different fault hypothesis, for example: \( H_0 \) : no fault, \( H_1 \) : bias in sensor 1, \( H_2 \) : bias in sensor 2; \( H_3 \) : zero output in sensor 1, etc. The hypothesis is then tested in terms of likelihood functions using, for example, Bayesian decision theory.

![Figure A1.3: IFD using a bank of estimators](image)

A well known approach for instrument fault detection is to assign a dedicated estimator to each of the sensors. In the dedicated observer scheme proposed for IFD...
by Clark in Ref [40], each sensor is driven by a different sensor output and the complete output vector \( y \) or, if this is not possible, as many components of \( y \) as possible, are estimated. With this scheme multiple simultaneous faults can be principally detected and isolated by checking properly structured sets of estimation errors, for example with the aid of a threshold logic. If, for example, a certain sensor fails, then the related output estimate reconstructed by the corresponding estimator will be erroneous which can be identified by the logic.

A.1.4 : Generalized Observer Scheme

The generalized observer scheme provides that an estimator dedicated to a certain sensor is driven by all outputs except that of the respective sensor. This IFD scheme allows one to detect and isolate only a single fault in any of the sensors, however with increased robustness with respect to unknown inputs.

A.2 The Kalman Filter for Linear Systems

When uncertainty in the form of noise is considered in the modeling, i.e., in the stochastic case, the states of a system are estimated using a Kalman filter. The Kalman filter is a set of mathematical equations that provides an efficient (recursive) solution of the least-square method. Consider a linear discrete time system described by the set of equations

\[
\begin{align*}
x(k+1) &= A(k)x(k) + b(k)u(k) + w(k) + f_i(k, \theta)v \\
y(k) &= C(k)u(k) + D(k)u(k) + v(k) + g_i(k, \theta)v
\end{align*}
\]

Where \( f_i(k, \theta)v \) and \( g_i(k, \theta)v \) represent the presence of the i-th type of abrupt change, i=1,\ldots,N. \( \theta \) is the unknown time the failure occurs and \( f_i \) and \( g_i \) are the dynamic profiles of the i-th change type. The scalar \( v \) denotes the magnitude of the
change. On the other hand $w(k)$ and $v(k)$ are zero mean white Gaussian noise sequences with covariances defined by

$$E[w(k)w(j)^T] = Q \delta_{ij} \quad A(9)$$
$$E[v(k)v(j)^T] = R \delta_{ij} \quad A(10)$$

Here $E[]$ is the expectation operator. Assume that we design a Kalman filter based on normal operation, i.e., neglecting the terms $f_i$ and $g_i$. Then the filter is given by

$$\hat{x}(k+1/k) = A(k)\hat{x}(k/k) + B(k)u(k) \quad A(11)$$
$$\hat{x}(k+1/k+1) = \hat{x}(k+1/k) + K(k+1)\gamma(k+1) \quad A(12)$$
$$\gamma(k+1) = y(k+1) - C(k)\hat{x}(k+1/k) \quad A(13)$$

The term $\gamma(k+1)$ is known as the innovations. It is the difference between the actual measurement and the expected measurement. The innovations drive the estimator. When the estimator is doing well, this driving term is small. The Kalman gain $K(k+1)$ is calculated from the following equations.

$$P(k+1/k) = A(k)P(k/k)A^T(k) + Q \quad A(14)$$
$$V(k) = C(k)P(k/k-1)C^T(k) + R \quad A(15)$$
$$K(k) = P(k/k-1)C^T(k)V^{-1}(k) \quad A(16)$$
$$P(k/k) = P(k/k-1) - K(k)C(k)P(k/k-1) \quad A(17)$$

Here $P(k+1/k)$ denotes the estimation error covariance in the estimate $\hat{x}(k+1/k)$ and $P(k+1/k+1)$ is the covariance of the error $x(k+1) - \hat{x}(k+1)$.

The innovation sequence is a zero mean white Gaussian noise with covariance $V(k)$ under the fault free condition. Consequently, deviations from this behaviour are indicative of failure. Using this fact, a variety of statistical tests have been proposed. A whiteness test on the innovations sequence produced by a Kalman filter was
proposed by Mehra and Peshon [60]; and the use of a bank of Kalman filters together with specific statistical tests, were proposed by Willsky [9].

Willsky calls his methods the Multiple Model (MM) method and the Generalized Likelihood Ratio (GLR). The difference between these methods is in the statistical test applied. The MM method proposes a scheme based on a bank of Kalman filters where each filter is designed based on a particular hypothesis. For example, Filter 1, assume that no change has occurred, Filter 2, assume a change based on the magnitude of a sensor, Filter 3 assume that a particular known fault has occurred, etc. The output of each filter is expected to be small if the corresponding hypothesis is correct. The decision concerned with the kind of failure affecting the system, was made by the observation of the filter with smaller innovation values. Therefore the problem became a multiple hypothesis-testing problem.

The GLR method model a change in the state and output of the dynamic system as \( f_i(k, \theta) \nu \) and \( g_i(k, \theta) \nu \). In these equations, \( \theta \) represented the unknown time at which the failure occurred, so \( f_i(k, \theta) = g_i(k, \theta) = 0 \) for \( k < \theta \). The scalar \( \nu \) denoted the magnitude of the failure. Then the dynamic behavior of the system was modeled

\[
\begin{align*}
x(k+1) &= A(k)x(k) + b(k)u(k) + w(k) + f_i(k, \theta) \nu \\
y(k) &= C(k)u(k) + D(k)u(k) + \nu(k) + g_i(k, \theta) \nu
\end{align*}
\]

Where the normal model consisted of these equations without the \( f_i(k, \theta) \nu \) and \( g_i(k, \theta) \nu \) terms. The GLR method is based on the idea that different types of faults produced different kinds of effects on the filter innovations and the method estimated each possible event by correlating the innovations with the corresponding signature.
A.3 Fault diagnosis through parameter estimation

Fault detection via parameter estimation relies on the principle that possible faults in the monitored system can be associated with specific parameters and states of the mathematical model of the system given in the form of an input-output relation:

\[ y(t) = f(u(t), e(t), \theta, x) \]  \hspace{1cm} (A21)

where \( y(t) \) represents the output vector of the system, \( u(t) \) the input vector, \( x(t) \) the state variables which are partially measurable, \( \theta \) the non measurable parameters which are likely to change on the occurrence of a fault, and \( e(t) \) the modeling errors and/or noise terms affecting the process.

The general procedure to detect faults follows the steps below:

1) Establishment of the mathematical model of the system’s normal behavior,

\[ y(t) = f(u(t), \theta) \]

At this stage, allowable tolerances for the system’s parameter values are also defined.

2) Determination of the relationship between the model parameters \( \theta_i \) and the physical system parameters \( p_j \).

3) Estimation of the model parameters \( \theta_i \) from measurements of \( y(t) \), \( u(t) \) by a suitable estimation procedure.

4) Calculation of the physical system parameters, via the inverse relationship:

\[ p = f^{-1}(\theta) \]

5) Decision on whether a fault has occurred, based either on the changes \( \Delta p_j \) or on the changes \( \Delta \theta_i \) and tolerance limits.
If the decision is made based on the ‘Δθ’ , the affected ‘p,’ s can be easily determined from step 2. This may be achieved with the aid of a fault catalogue in which the relationship between process faults and changes in the coefficients ‘Δp’ has been established. Decision can be made either by simply checking against the predetermined threshold levels, or by using more sophisticated methods from the field of statistical decision theory. The basis of this class of methods is the combination of theoretical modeling and parameter estimation of continuous time models. The procedure is illustrated in Fig. A3.1.

Fig. A3.1 Parameter identification approach for fault detection
A.4 Fault diagnosis using fuzzy logic

Fault diagnosis requires a classification system that can distinguish between different faults based on observed symptoms of the process under investigation. Since the fault symptom relationships are not always known beforehand, a system is needed which can be learned from experimental or simulated data. A fuzzy logic based diagnosis is advantageous. It allows an easy incorporation of a-priori known rules and also enables the user to understand the inference of the system.

In general, it is not possible to obtain an exact model of a real process, only an approximation can be found. Therefore, one has to cope with modeling errors which degrade the performance of model-based fault detection methods. A rule based representation of human problem solving strategies is an appropriate technique for this type of problem. Therefore, today’s diagnosis systems make use of both mathematical descriptions as well as heuristic formulations. To incorporate both techniques a symbolic representation of numerical values is needed. Recently, fuzzy theory is used in many technical disciplines taking care of vague descriptions. The fuzzy approach is used to build an adaptive fuzzy threshold which takes care of modeling errors, so that no increased threshold is necessary and even small faults can quickly be detected.

Fuzzy logic is used to combine crisp numerical values from the technical domain with vague concepts from human decision making abilities. The numerical values are transformed into linguistic terms by membership functions. Based on linguistic terms, experienced knowledge is composed into production rules. The exploitation of these rules, also known as fuzzy inference, can be carried out with different methods. The most common one is that of Mamdani which takes the minimum for a logical AND and the maximum for a logical OR.
A.4.1 Fuzzy Inference System

Fuzzy inference is the process of formulating the mapping from a given input to an output using fuzzy logic. The mapping then provides a basis from which decisions can be made. i.e., fuzzy inference is a method that interprets the values in the input vector and, based on some set of rules, assigns values to the output vector. There are two types of fuzzy inference systems that can be implemented in the Fuzzy Logic Toolbox: Mamdani-type and Sugeno-type. These two types of inference systems vary somewhat in the way outputs are determined. In the Fuzzy Logic Toolbox, there are five parts of the fuzzy inference process: fuzzification of the input variables, application of the fuzzy operator (AND or OR) in the antecedent, implication from the antecedent to the consequent, aggregation of the consequents across the rules, and defuzzification.

A.4.2 Fuzzification

The first step is to take the inputs and determine the degree to which they belong to each of the appropriate fuzzy sets via membership functions. In the Fuzzy Logic Toolbox, the input is always a crisp numerical value limited to the universe of discourse of the input variable and the output is a fuzzy degree of membership in the qualifying linguistic set (always the interval between 0 and 1).

A.4.3 Fuzzy operator

Once the inputs have been fuzzified, the degree to which each part of the antecedent has been satisfied for each rule is known. If the antecedent of a given rule has more than one part, the fuzzy operator is applied to obtain one number that represents the result of the antecedent for that rule. This number will then be applied to the output function. The input to the fuzzy operator is two or more membership values from fuzzified input variables. The output is a single truth value. In the Fuzzy Logic Toolbox, two built-in AND methods are supported: min (minimum) and prod (product). Two built-in OR methods are also supported: max (maximum), and the probabilistic OR method probor.
A.4.4 Implication method

The input for the implication process is a single number given by the antecedent, and the output is a fuzzy set. Implication is implemented for each rule. Two built-in methods are supported, and they are the same functions that are used by the AND method: min (minimum), which truncates the output fuzzy set, and prod (product), which scales the output fuzzy set.

A.4.5 Aggregate All Outputs

Aggregation is the process by which the fuzzy sets that represent the outputs of each rule are combined into a single fuzzy set. Aggregation only occurs once for each output variable, just prior to the fifth and final step, defuzzification. The input of the aggregation process is the list of truncated output functions returned by the implication process for each rule. The output of the aggregation process is one fuzzy set for each output variable.

A.4.6 Defuzzification

The input for the defuzzification process is a fuzzy set (the aggregate output fuzzy set) and the output is a single number. As much as fuzziness helps the rule evaluation during the intermediate steps, the final desired output for each variable is generally a single number. However, the aggregate of a fuzzy set encompasses a range of output values, and so must be defuzzified in order to resolve a single output value from the set.

Fuzzy logic-based motor fault diagnosis methods have the advantages of embedded linguistic knowledge and approximate reasoning capability. However, the design of such a system heavily depends on the intuitive experience acquired from practicing operators. The fuzzy rules which form the knowledge base are developed from a combination of faults that is reflected from the combination of the estimated parameters in the plant. The inputs of the reasoning engine are changes in the parameters and its consequent is the type of fault on the system.
The DC motor parameters are identified using the EKF parameter estimation technique. The normal parameters of the system are first identified which are used as the reference. Then the abnormal or faulty parameters are also identified by changing the motor conditions. The fuzzification component is constructed using these identified parameters with minimum and maximum tolerance. The fuzzy rules are then constructed from the relation of the identified parameters with respect to the normal parameters.

By representing the symptoms as fuzzy sets with membership functions $0 \leq \mu(\xi_i) \leq 1$ and the events (faults) with $0 \leq \mu(\eta_k) \leq 1$ the conditional part of the IF-THEN-rules can then be determined by fuzzy-logic operations. For the determination of the membership function $\mu(\eta_k)$ of the output based on the inputs $\mu(\xi_i)$ different t-norms and t-conorms can be applied. Relatively simple operations are obtained by max-min operation.

$$\text{Fuzzy-AND } \mu(\eta) = \min[\mu(\xi_1), \ldots, \mu(\xi_v)]$$

$$\text{Fuzzy-OR } \mu(\eta) = \max[\mu(\xi_1), \ldots, \mu(\xi_v)]$$

The set of nominal values detected by the parameter estimation process is used as the center of the universe of discourse for each fuzzy variable to be used in the fuzzy fault diagnosis scheme. The range of values around each center is determined based on an interval that takes into account practical aspects, being sufficiently large to guarantee the completeness of the set of rules. After determining the universe of discourse, membership functions are defined to specify the set of antecedents and consequents of each rule. The membership functions are taken to be triangular and trapezoidal functions. The fuzzy rules for fault detection are constructed based on the purpose of finding which physical parameter (or set of parameters) is responsible for the detected fault.
5. Benchmarking C Code on the ADSP 21161 DSP Processor

(i) **ADSP 21161 DSP Processor**

SHARC is the name of a family of high performance 32 bit floating point processors based on a Super Harvard Architecture. The Super Harvard architecture extends the original concepts of separate program and data memory buses by adding an I/O processor with its associated dedicated busses. The block diagram of ADSP-21161N SHARC DSP processor is shown in Figure A5.1.

![Figure A5.1 ADSP 21161 SHARC Block diagram (Courtesy Ref :102)](image)

Fabricated in a state of the art, high speed, low power CMOS process, the ADSP-21161N has a 10 ns instruction cycle time. With its Single Instruction Multiple Data (SIMD) computational hardware running at 100 MHz the ADSP 21161N can perform 600 million math operations per second.
Visual DSP++ is an easy to install and easy to use integrated software development and debugging environment (IDDE) that enables efficient management of projects from start to finish from within a single interface. Because project development and debugging is integrated, one can move quickly and easily between editing, building and debugging activities. Key features include the native C/C++ Compiler, advanced graphical plotting tools, statistical profiling and the Visual DSP++ Kernal (VDK). Other features include assembler, linker, libraries, loader, splitter, cycle-accurate and functional-accurate compiled simulators, emulator support, etc.

At the heart of VisualDSP++ is a robust and powerful C/C++ compiler. The compiler is designed to ensure that the most performance demanding applications can be written entirely in the C language, accelerating development time while maintaining a portable code base. The compiler is backed by a rich library of signal processing routines, allowing easy access to hand coded, optimized implementation of FFT’s, FIR filters, etc. The compiler is further backed by a powerful assembler and linker technology.

The compiler processes C/C++ programs into assembly code. The compiler generates a linkable object file by compiling one or more C/C++ source files. The assembler generates an object file by assembling a source, header and data files. C and C++ programs depend on library functions to perform operations that are basic to the C and C++ programming languages. These operations include memory allocations, character and string conversions, and math calculations. The C and C++ run-time libraries (RTLs) are collections of functions, macros, and class templates that can be called from source programs. The RTL simplifies software development by providing code for a variety of common needs.

The linker links separately assembled files (object files and library files) to produce executable files, shared memory files and overlay files which can be loaded onto the target. The splitter produces executable files to prepare non-bootable programmable read-only memory (PROM) image files. These files are executed from the processor’s external memory.
Debugging both C/C++ and assembly programs with the Visual DSP++ debugger, programmers can:

- View mixed C/C++ and assembly code (interleaved source and object information).
- Insert break points
- Set conditional breakpoints on registers, memory and stacks
- Trace instruction execution
- Perform linear or statistical profiling of program execution
- Fill, dump and graphically plot the contents of memory
- Source level debugging
- Create custom debugger windows

(iii) Benchmarking C code on the ADSP-21161 family of DSPs

Benchmarking is done to measure the performance of a C compiler, or to understand how many DSP clock cycles a specific section of code will take. Once the number of clock cycles is known, the amount of time that function will take to execute can be quickly calculated using the instruction rate of that processor.

The ADSP-21xxx family of DSPs has a set of registers called EMUCLK and EMUCLK2 which make up a 64 bit counter. This counter is unconditionally incremented during every instruction cycle on the DSP and is not affected by cache-misses, wait states, etc. Every time EMUCLK wraps back to zero, EMUCLK2 is incremented by one as though they are one 64 bit register. These registers are used for benchmarking purposes and can be accessed like any universal register. However, these variables are not directly accessible from C as they are memory mapped and hence Analog Devices have provided two macros to retrieve the EMUCLK values and compute cycle counts.

The Visual DSP++ libraries offer the ability to obtain accurate cycle counts and measure execution time by embedding code in an application to calculate and return cycle count information. This is facilitated by the above cycle count registers and macros provided in the libraries.
The two macros provided are CYCLE_COUNT_START and CYCLE_COUNT_STOP. The first macro, CYCLE_COUNT_START, copies the contents of EMUCLK into an integer variable. The second macro, CYCLE_COUNT_STOP, subtracts the stored value of EMUCLK from the current value. It then subtracts a value of 4 representing the overhead incurred in these two functions. The final value is then stored back into the integer variable.

These macros do not take into account the potential wrapping of the EMUCLK register back to zero and the subsequent increment of the EMUCLK2 register in order to save space and execution time. If benchmarking is to be done on a free-running system, the EMUCLK register will wrap once every 71 seconds on a 60 Mhz DSP. However, since most benchmarking is typically done in a simulation environment or in a controlled emulation environment, an EMUCLK wrap is typically never encountered as these registers are reset with the processor.

The two macros along with the structure of the program to be used for counting the number of cycles is taken for executing the application program is as follows.

```c
#include <stdio.h>
/*Cycle Count Example Code */
/*Cycle count macros */
#define CYCLE_COUNT_START (cntr) asm("r0 = emuclk; %0 =r0; "; 
   "=k" (cntr): "d" (cntr): \
   
   "r0")
#define CYCLE_COUNT_STOP (cntr) asm("r0 = emuclk; r1 =%1; r2=4; \
   r0 = r0 - r2; r0=r0 -r1; %0 = r0;" : \
   "=k" (cntr) : \
   "d" (cntr) : "r0", "r1")
int cnt0;  // does not have to be global
main ()
{
    int i;
    //read contents of EMUCLK and store in cnt0
    CYCLE_COUNT_START (cnt0);
}
```
Put application code to be benchmarked here

// calculate total number of cycles and store result in cnt0
CYCLE_COUNT_STOP (cnt0);
// print the result of the benchmark

Printf ("The cycle count for the application code was %d cycles. \n", cnt0);
}

The above macros were used for making assessment of the execution time of the linear Kalman filter, Extended Kalman filter and Unscented Kalman filters. The different FDI algorithms were written in C language and then compiled to ADSP 21161 executable code using the VDSP++ development environment. The above macros were used to assess the number of clock cycles taken by the DSP for executing the code. This is then divided by the operating frequency of 100 MHz of the DSP processor to obtain the execution time in seconds. The output display window showing the number of clock cycles taken is shown Figure A5.2, Figure A5.3 and Figure A5.4.
Figure A5.2  Output window showing number of execution cycles for linear Kalman filter
Figure A5.3: Output window showing number of execution cycles for Extended Kalman Filter
Figure A5.4: Output window showing number of execution cycles for Unscented Kalman Filter
(iv) Useful C routines for Matrix operations

The implementation of the Kalman Filter, the Extended Kalman filter and the Unscented Kalman filter requires a number of matrix operations like multiplication, addition, transpose and inverse operations. Some useful C routines that was used for this purpose is given below.

Matrix Multiplication Routine

```c
void MatrixMultiply(float* A, float* B, int m, int p, int n, float* C) {
  // A = input matrix (m x p)
  // B = input matrix (p x n)
  // m = number of rows in A
  // p = number of columns in A = number of rows in B
  // n = number of columns in B
  // C = output matrix = A*B (m x n)
  int i, j, k;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
      {
        C[n*i+j]=0;
        for (k=0; k<p; k++)
          C[n*i+j]= C[n*i+j]+A[p*i+k]*B[n*k+j];
      }
}
```

Matrix Addition Routine

```c
void MatrixAddition(float* A, float* B, int m, int n, float* C) {
  // A = input matrix (m x n)
  // B = input matrix (m x n)
  // m = number of rows in A = number of rows in B
  // n = number of columns in A = number of columns in B
  // C = output matrix = A+B (m x n)
  int i, j;
  for (i=0; i<m; i++)
    for (j=0; j<n; j++)
}
```
Matrix Subtraction Routine

void MatrixSubtraction(float* A, float* B, int m, int n, float* C)
{
  // A = input matrix (m x n)
  // B = input matrix (m x n)
  // m = number of rows in A = number of rows in B
  // n = number of columns in A = number of columns in B
  // C = output matrix = A-B (m x n)
  int i, j;
  for (i=0;i<m;i++)
    for(j=0;j<n;j++)
      C[n*i+j]=A[n*i+j]-B[n*i+j];
}

Matrix Transpose Routine

void MatrixTranspose(float* A, int m, int n, float* C)
{
  // A = input matrix (m x n)
  // m = number of rows in A
  // n = number of columns in A
  // C = output matrix = the transpose of A (n x m)
  int i, j;
  for (i=0;i<m;i++)
    for(j=0;j<n;j++)
      C[m*j+i]=A[n*i+j];
}

Matrix Inversion Routine

int MatrixInversion(float* A, int n, float* Alnverse)
{
  // A = input matrix (n x n)
  // n = dimension of A
  // Alnverse = inverted matrix (n x n)
  // This function inverts a matrix based on the Gauss Jordan method.
  // The function returns 1 on success, 0 on failure.
  int i, j, iPass, imx, icol, irowi;
  float det, temp, pivot, factor;
  float* ac = (float*)calloc(n*n, sizeof(float));
  det = 1;
  for (i = 0; i < n; i++)
\{
for (j = 0; j < n; j++)
{
    AInverse[n*i+j] = 0;
    ac[n*i+j] = A[n*i+j];
}
AI\text{Inverse}[n*i+i] = 1;
\}

// The current pivot row is iPass.
// For each pass, first find the maximum element in the pivot column.
for (iPass = 0; iPass < n; iPass++)
{
    imx = iPass;
    for (irow = iPass; irow < n; irow++)
    {
        if (fabs(A[n*irow+iPass]) > fabs(A[n*imx+iPass])) imx = irow;
    }
    // Interchange the elements of row iPass and row imx in both A and AInverse.
    if (imx != iPass)
    {
        for (icol = 0; icol < n; icol++)
        {
            temp = AInverse[n*iPass+icol];
            AInverse[n*iPass+icol] = AInverse[n*imx+icol];
            AInverse[n*imx+icol] = temp;
            if (icol >= iPass)
            {
                temp = A[n*iPass+icol];
                A[n*imx+icol] = temp;
            }
        }
    }
    // The current pivot is now A[iPass][iPass].
    // The determinant is the product of the pivot elements.
    pivot = A[n*iPass+iPass];
    det = det * pivot;
    if (det == 0)
    {
        free(ac);
        return 0;
    }
}

185
for (icol = 0; icol < n; icol++)
{
    // Normalize the pivot row by dividing each
    // element by the pivot element.
    Alnverse[n*iPass+icol] =
    A[n*iPass+icol] / pivot;
    if (icol == iPass) A[n*iPass+icol] =
    A[n*iPass+icol] / pivot;
}

for (irow = 0; irow < n; irow++)
    // Add a multiple of the pivot row to each
    // is chosen so that the element of A on the
row. The multiple factor
    // pivot column is 0.
    { factor = A[n*irow+iPass];
        for (icol = 0; icol < n; icol++)
        { if (irow != iPass)
            { A[n*irow+icol] -= factor *
                Alnverse[n*irow+icol];
                A[n*irow+icol] -= factor *
                A[n*irow+icol];
            }
        }
    }
free(ac);
return 1;