Part I

Rudiments

Part II

Archetypes

Chapter 1

Modal Example

The linear system for this problem relates a series of temperature measurements, T, to positions, x:

$$\begin{array}{ccc} \mathbf{A} & a & = & T \\ \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_9 \end{bmatrix} & \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} & = & \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_9 \end{bmatrix}. \end{array}$$

Different solution methods are explored in the other volume. The solution of minimum error norm is the pseudoinverse solution

$$a = \mathbf{A}^{\dagger} T.$$

The normal equations are another path. They are

$$\mathbf{A}^{*}\mathbf{A}a = \mathbf{A}^{*}T$$

which has the solution

$$a = \left(\mathbf{A}^{*}\mathbf{A}\right)^{-1}\mathbf{A}^{*}T. \tag{1.1}$$

The errors in the fit parameters are an important part of the solution and they are given by

$$\epsilon_k = \sqrt{\frac{r^{*}r}{m-n}\left(\mathbf{A}^{*}\mathbf{A}\right)^{-1}_{kk}}, \quad k = 1\colon n$$

1.1 Mathematica

Figure 1.1 shows the input data and computation in Mathematica. Notice that the data are expressed as rational numbers to preclude numeric errors.

data

```
In[1]:= n = 2; (* number of free parameters *)
        T = 1/10 {156, 175, 366, 438, 582, 616, 642, 704, 988};
        m = Length[T]; (* number of measurements *)
        X = Range[m];
        one = Table[1, {m}]; (* vector of 1's *)
        A = {one, X}^T; (* system matrix A *)
```

solution method I

```
In[7]:= {intcpt, slope} = LeastSquares[A, T]
        % // N
Out[7]= {1733/360, 1129/120}
Out[8]= {4.81389, 9.40833}
```

solution method II

```
In[9]:= {intcpt, slope} = Inverse[A^H.A].A^H.T
Out[9]= {1733/360, 1129/120}
```

solution method III

```
In[10]:= {intcpt, slope} = PseudoInverse[A].T
Out[10]= {1733/360, 1129/120}
```

error

```
In[11]:= r = A.{intcpt, slope} - T; (* residual errors *)
In[12]:= ϵ = Sqrt[r.r/(m - n) Diagonal[Inverse[A^H.A]]] // N
Out[12]= {4.88621, 0.868302}
```

Figure 1.1. *Bevington's example solved in Mathematica using different methods.*

Three solution paths are shown. The first path (`In[7]`) uses the intrinsic command `LeastSquares`, and, as in the other cases, exact results are returned. The next method is based upon the normal equations (`In[9]`), and the final solution uses the pseudoinverse method (`In[10]`).

1.2 Fortran 2015

The main program, `bevington.f08`, is rather brief and acts as control routine. The module files are loaded, and then routines are called to load the data, compute intermediate quantities and results.

```
! respect load order
```

```
include 'sharedModules/mod precision definitions.f08'
include 'sharedModules/mod parameters.f08'
include 'sharedModules/mod measurements.f08'
include 'sharedModules/mod intermediates definitions.f08'
include 'sharedModules/mod build matrices.f08'
include 'sharedModules/mod compute results.f08'

program bevington

    use mPrecisionDefinitions, only : ip
    use mParameters
    use mMeasurements
    use mIntermediatesDefinitions
    use mBuildMatrices
    use mResults

    implicit none

    integer ( ip ) :: k = 0

    type ( measurements )  :: myMeasurements
    type ( intermediates ) :: myIntermediates
    type ( matrices )      :: myMatrices
    type ( results )       :: myResults

        call myMeasurements  % load_data             ( )
        call myIntermediates % compute_intermediates ( myMeasurements )
        call myMatrices      % build_matrices        ( myIntermediates )
        call myResults       % compute_results       ( myMatrices, myMeasurements )

        do k = 1, n
            print *, myResults % descriptor ( k ), " = ", &
                     myResults % a ( k ), "+/-", &
                     myResults % epsilon ( k )
        end do

end program bevington
```

The modules are listed according to the load order established by the `include` commands. The first of these modules defines working precision for the computation. Therefore, the precision is controlled in two places (lines 8 and 9).

```
1  module mPrecisionDefinitions
2  
3      use iso_fortran_env
4  
5      implicit NONE
6  
7      ! Define working precision
8      integer, parameter         :: rp = REAL64
9      integer, parameter         :: ip = INT64
10 
11     ! real constants
12     real ( rp ), parameter     :: zero = 0.0_rp
13     real ( rp ), parameter     :: one  = 1.0_rp
```

```
end module mPrecisionDefinitions
```

The number of fit parameters is an important number which determines the number of columns in the matrix **A**. While this parameter definition can be tucked into the main routine, it is a seed for other parameters as simulation complexity grows.

```
module mParameters

    use mPrecisionDefinitions, only : ip

    implicit none

    integer ( ip ), parameter :: n = 2

end module mParameters
```

These data are taken from [?, Table 6-1] and represent the position in cm, `x`, and the temperature in centigrade, `y`.

```
module mMeasurements

    use mPrecisionDefinitions, only : ip, rp, one, zero

    implicit none

    integer ( ip ), parameter :: m = 9

    type :: measurements
        real ( rp ), dimension ( 1 : m ) :: x = zero, y = zero, &
                                            ones = one, residuals = zero
    contains
        private
        procedure, public :: load_data
    end type measurements

contains

    subroutine load_data ( me )

        class ( measurements ), target :: me

!           load data
            me % x ( 1 ) = 1.0_rp
            me % x ( 2 ) = 2.0_rp
            me % x ( 3 ) = 3.0_rp
            me % x ( 4 ) = 4.0_rp
            me % x ( 5 ) = 5.0_rp
            me % x ( 6 ) = 6.0_rp
            me % x ( 7 ) = 7.0_rp
            me % x ( 8 ) = 8.0_rp
            me % x ( 9 ) = 9.0_rp

            me % y ( 1 ) = 15.6_rp
            me % y ( 2 ) = 17.5_rp
```

```
            me % y ( 3 ) = 36.6_rp
            me % y ( 4 ) = 43.8_rp
            me % y ( 5 ) = 58.2_rp
            me % y ( 6 ) = 61.6_rp
            me % y ( 7 ) = 64.2_rp
            me % y ( 8 ) = 70.4_rp
            me % y ( 9 ) = 98.8_rp

    end subroutine load_data

end module mMeasurements
```

The first processing step is to perform the vector operations. Another option is to use the intrinsic command `sum` instead of using the dot product. But please, don't use `do` loops; the vector tools in Fortran are much faster.

```
module mIntermediatesDefinitions

    use mMeasurements

    implicit none

    type :: intermediates
        real ( rp ) :: em = zero, sx = zero, sx2 = zero, sy = zero, sxy = zero
    contains
        private
        procedure, public :: compute_intermediates
    end type intermediates

contains

    subroutine compute_intermediates ( me, myMeasurements )

        class ( intermediates ), target      :: me

        type ( measurements ), intent ( in ) :: myMeasurements

            me % em  = dot_product ( myMeasurements % ones, myMeasurements % ones )
            me % sx  = dot_product ( myMeasurements % ones, myMeasurements % x )
            me % sy  = dot_product ( myMeasurements % ones, myMeasurements % y )
            me % sx2 = dot_product ( myMeasurements % x,    myMeasurements % x )
            me % sxy = dot_product ( myMeasurements % x,    myMeasurements % y )

    end subroutine compute_intermediates

end module mIntermediatesDefinitions
```

The matrices are built using array constructors for the column vectors (lines 27, 28, and 30). Notice that the product matrix $\mathbf{A}^*\mathbf{A}$ is never constructed; instead there is a direct construction of the inverse matrix, $(\mathbf{A}^*\mathbf{A})^{-1}$.

```
1 module mBuildMatrices
2 
3     use mIntermediatesDefinitions
4     use mParameters
```

```

    implicit none

    type :: matrices
        real ( rp ), dimension ( 1 : n, 1 : n ) :: ASAinv = zero
        real ( rp ), dimension ( 1 : n )        :: B      = zero
        real ( rp )                             :: det    = zero
    contains
        private
        procedure, public :: build_matrices
    end type matrices

contains

    subroutine build_matrices ( me, myIntermediates )

        class ( matrices ),     target        :: me

        type ( intermediates ), intent ( in ) :: myIntermediates

            me % det = myIntermediates % em * myIntermediates % sx2 &
                                            - myIntermediates % sx ** 2
            me % ASAinv ( : , 1 ) = [ myIntermediates % sx2, -myIntermediates % sx ]
            me % ASAinv ( : , 2 ) = [-myIntermediates % sx,   myIntermediates % em ]
            me % ASAinv = me % ASAinv / me % det
            me % B = [ myIntermediates % sy, myIntermediates % sxy ]

    end subroutine build_matrices

end module mBuildMatrices
```

This final module computes the solution vector a and the error vector ϵ using vector tools. The diagonal matrix elements are harvested in line 40 using an implied `do` loop.

```
1  module mResults

       use mBuildMatrices
       use mParameters
5      use mMeasurements

       implicit none

       integer ( ip ) :: row = 0
10
       type :: results
           real ( rp ), dimension ( 1 : n )          :: a          = zero
           real ( rp ), dimension ( 1 : n )          :: epsilon    = zero
           real ( rp )                               :: r2         = zero
15         character ( len = 9 ), dimension ( 1 : n ) :: descriptor = [ 'intercept', &
                                                                        'slope    ' ]
       contains
           private
           procedure, public :: compute_results
20     end type results
```

```
contains

    subroutine compute_results ( me, myMatrices, myMeasurements )

        class ( results ), target               :: me

        type ( matrices ),     intent ( in )    :: myMatrices
        type ( measurements ), intent ( inout ) :: myMeasurements

!           fit parameters
            me % a = matmul ( myMatrices % ASAinv, myMatrices % B )
            me % a = matmul ( myMatrices % ASAinv, myMatrices % B )

!           errors
            myMeasurements % residuals = myMeasurements % y - me % a ( 1 ) &
                                       - myMeasurements % x * me % a ( 2 )
            me % r2 = dot_product ( myMeasurements % residuals, &
                                    myMeasurements % residuals )
            me % epsilon = [ ( myMatrices % ASAinv ( row, row ), row = 1, n ) ]
            me % epsilon = sqrt ( me % epsilon * me % r2 / ( m - n ) )

    end subroutine compute_results

end module mResults
```

The compilation command is

```
$ gfortran  -Wall -Wextra -Wconversion -Og -pedantic -fcheck=bounds \\
    -fbacktrace -g  -fmax-errors=5 xbevington.f08
```

and execution results are

```
$ ./a.out
  intercept =     4.8138888888888971      +/-   4.8862063121833534
  slope     =     9.4083333333333314      +/-  0.86830164765636075
```

How good are these numerical values? With arbitrary precision computation we are able to measure the full precision of a computation:

Fortran	4.8138888888888971	9.4083333333333314
Arbitrary precision	4.8138888888888888888888889	9.4083333333333333333333333

1.3 Octave

The open-source program Octave offers a nice set of tools applicable to numerical linear algebra as seen by the following screen session. The first line creates a column vector of data, and the second line measures the length. This returns m, the number of measurements. Because the measurements are on a regular mesh, the vector x can be generated easily with an array command as seen in 3. There are a few ways to compute a vector of 1's and in line 4 we chose to exploit vector arithmetic.

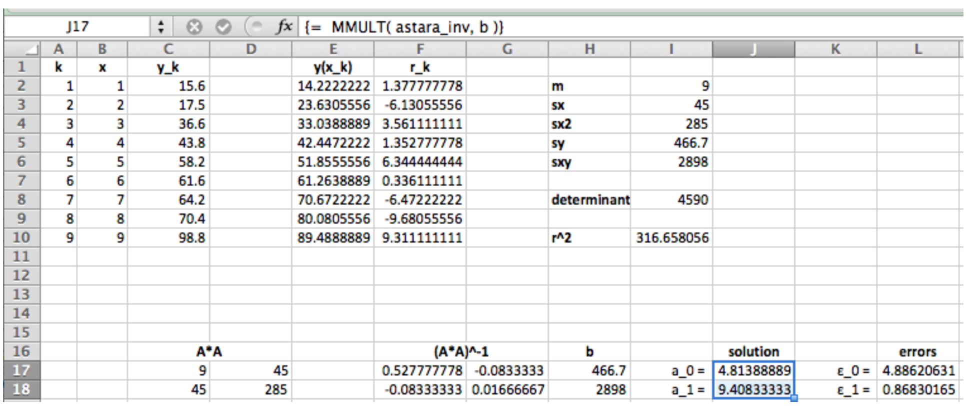
```
octave:1> T = [ 156; 175; 366; 438; 582; 616; 642; 704; 988 ] / 10;
octave:2> m = length( T );
octave:3> x = 1 : m;
octave:4> ones = x - x + 1;
octave:5> A = [ ones ; x ].';
```

Three different solution methods follow. The first, in line 6, matches (1.1). The next method, line 7, exploits the ease of the backslash command in solving linear systems. Finally, in line 8, there is a direct application of the pseudoinverse matrix using the command `pinv`.

```
octave:6> a = inv( A.' * A ) * ( A.' * T )
a =

   4.8139
   9.4083

octave:7> a = A.' * A \ A.' * T
a =

   4.8139
   9.4083

octave:8> a = pinv( A ) * T
a =

   4.8139
   9.4083
```

1.4 Excel 2011

Spreadsheet toolkits can solve the example problem. Figure 1.2 shows one method of solution.



J17 {= MMULT(astara_inv, b)}

	A	B	C	D	E	F	G	H	I	J	K	L
1	k	x	y_k		y(x_k)	r_k						
2	1	1	15.6		14.2222222	1.377777778		m	9			
3	2	2	17.5		23.6305556	-6.13055556		sx	45			
4	3	3	36.6		33.0388889	3.561111111		sx2	285			
5	4	4	43.8		42.4472222	1.352777778		sy	466.7			
6	5	5	58.2		51.8555556	6.344444444		sxy	2898			
7	6	6	61.6		61.2638889	0.336111111						
8	7	7	64.2		70.6722222	-6.47222222		determinant	4590			
9	8	8	70.4		80.0805556	-9.68055556						
10	9	9	98.8		89.4888889	9.311111111		r^2	316.658056			
11												
12												
13												
14												
15												
16			A*A			(A*A)^-1		b		solution		errors
17			9	45		0.527777778	-0.0833333	466.7	a_0 =	4.81388889	ε_0 =	4.88620631
18			45	285		-0.08333333	0.01666667	2898	a_1 =	9.40833333	ε_1 =	0.86830165

Figure 1.2. *Bevington's example solved in Excel using the normal equations.*

Table 1.1 details the named ranges. Table 1.2 details the named variables. The product matrix $\mathbf{A}^*\mathbf{A}$ has the expression

$$\mathtt{astara} = \begin{bmatrix} \mathtt{m} & \mathtt{sx} \\ \mathtt{sx} & \mathtt{sx\_} \end{bmatrix}.$$

The inverse matrix, $(\mathbf{A}^*\mathbf{A})^{-1}$, is defined using an intrinsic command

$$\mathtt{astara\_inv} = \mathtt{MINVERSE(\ astara\ )}.$$

Table 1.1. *Named ranges.*

name	data range	formula
`x`	`B2:B10`	
`y`	`C2:C10`	
`prediction`	`E2:E10`	`= intercept + x * slope`
`residuals`	`F2:F10`	`= y - prediction`
`astara`	`D17:E18`	
`astara_inv`	`F17:G18`	
`b`	`H17:H18`	

Table 1.2. *Named variables.*

name	formula
`m`	`= COUNT( x )`
`sx`	`= SUM( x )`
`sx2_`	`= SUMPRODUCT( x, x )`
`sy`	`= SUM( y )`
`sxy`	`= SUMPRODUCT( x, y )`
`determinant`	`= m * sx2_ + sx^2`
`r^2`	`= SUMSQ( residuals )`

The data vector is

$$\mathtt{b} = \begin{bmatrix} \mathtt{sy} \\ \mathtt{sxy} \end{bmatrix}.$$

The solution is a matrix–vector multiplication `MMULT( astara_inv, b )`, and the individual entries are named `intercept` and `slope`. The error term, ϵ, is the only instance where cell addressing is used. For instance, the term ϵ_0 is computed using `= SQRT( r_2 / ( m - 2 ) * F17 )`, where `F17` is the first term on the diagonal of $(\mathbf{A}^*\mathbf{A})^{-1}$.

Part III

Applications: Finding Patterns

Part IV

Applications: Stitching

Part V

Applications: Inverting the Gradient

Part VI

Applications: Nonlinear Problems