Matrix and Vector class Manual

MATRIX AND VECTOR CLASS MANUAL 1

1 INTRODUCTION 9

2 VERSION HISTORY 9

2.1 VERSION 1.0.0.0 9
2.2 VERSION 1.0.*.* 9
2.3 MANUAL HISTORY 11

3 LICENSE CONDITIONS 14

4 INSTALLATION INSTRUCTIONS 14

5 VECTOR CLASS 16

5.1 CONSTRUCTORS 16
5.1.1 Vector() *16*
5.1.2 Vector(int n) *16*
5.1.3 Vector(Vector v) *16*
5.2 PROPERTIES 16
5.2.1 Vector Abs *16*
5.2.2 ComplexVector ViewAsComplex *17*
5.2.3 Vector Cos *17*
5.2.4 int Dim *17*
5.2.5 Vector Exp *17*
5.2.6 [i] *17*
5.2.7 Vector Log *17*
5.2.8 double Max *17*
5.2.9 double Min *18*
5.2.10 double Mean *18*
5.2.11 Modulus *18*
5.2.12 Vector Sin *18*
5.2.13 double Sd *18*
5.2.14 UnitVector *18*
5.3 OVERLOADED OPERATORS 19
5.3.1 + operator *19*
5.3.2 Add(Vector v) *19*
5.3.3 – operator *19*
5.3.4 Subtract(Vector v) *19*
*5.3.5 * operator* *19*
5.3.6 Multiply(double d) *20*
5.3.7 operator / *20*
5.3.8 Divide(double d) *20*
5.3.9 operator == *20*
5.3.10 operator != *20*
5.4 METHODS 21
5.4.1 Assign *21*
5.4.2 Vector ConvertVectorFromLocalSystem(Matrix M) *21*
5.4.3 Vector ConvertVectorToLocalSystem(Matrix M) *21*
5.4.4 Vector Convolution(Vector v) *21*
5.4.5 Vector Crosscorrelation(Vector $\boldsymbol{v}$, Boolean $\boldsymbol{normalise}$) *21*
5.4.6 double Dot *22*
5.4.7 void Edit() *22*
5.4.8 Vector GaussVector() *22*
5.4.9 Vector HouseholderVector() *23*
5.4.10 Boolean IsMonotonic() *23*

5.4.11 *Boolean IsZero()* 23
5.4.12 *Matrix ViewAsMatrix()* 23
5.4.13 *double MaxL(ref int loc)* 24
5.4.14 *double MinL(ref int loc)* 24
5.4.15 *Vector NelderMeanSolve(NelderMeanFit $\boldsymbol{f}$, Vector $\boldsymbol{h}$, Vector $\boldsymbol{x}$, Vector $\boldsymbol{y}$, int ref $\boldsymbol{conv}$)* 24
5.4.16 *Vector NonlinearSolve(NonlinearFit f Vector x ,Vector y , int ref $conv$)* 25
5.4.17 *Vector NonlinearSolve2(NonlinearFit2 f Vector[ ] x ,Vector y , int ref $conv$)* 27
5.4.18 *Polynomial(double d)* 28
5.4.19 *SetSubVector(int i_1 ,int i_2 ,Vector v)* 28
5.4.20 *Vector Sort(int [ ] $location$)* 28
5.4.21 *void Swap(int i ,int j)* 29
5.4.22 *Vector SubVector(int i_1 ,int i_2)* 29
5.4.23 *Matrix Transpose()* 29

6 COMPLEX CLASS 30

6.1 CONSTRUCTORS 30
6.1.1 *Complex()* 30
6.1.2 *Complex(Complex c)* 30
6.1.3 *Complex(double $real$,double $imag$)* 30
6.2 PROPERTIES 30
6.2.1 *Complex Cos* 30
6.2.2 *Complex Exp* 30
6.2.3 *double Real* 30
6.2.4 *double Imaginary* 30
6.2.5 *double Magnitude* 31
6.2.6 *double Phase* 31
6.2.7 *Complex Sin* 31
6.2.8 *Complex Sqrt* 31
6.3 OVERLOADED OPERATORS 32
6.3.1 *+ operator* 32
6.3.2 *Add(Complex c)* 32
6.3.3 *- operator* 32
6.3.4 *Subtract(Complex c)* 32
6.3.5 ** operator* 32
6.3.6 *Multiply(Complex c)* 32
6.3.7 *operator /* 32
6.3.8 *Divide(Complex c)* 33
6.3.9 *operator ==* 33
6.3.10 *operator !=* 33
6.4 METHODS 33
6.4.1 *Complex Conjugate()* 33
6.4.2 *void Assign(Complex c)* 33

7 COMPLEXVECTOR CLASS 34

7.1 CONSTRUCTORS 34
7.1.1 *ComplexVector()* 34
7.1.2 *ComplexVector(int dim)* 34
7.1.3 *ComplexVector(ComplexVector v)* 34
7.2 PROPERTIES 34
7.2.1 *int Dim* 34
7.2.2 *[i]* 34
7.2.3 *Vector Abs* 35
7.2.4 *ComplexVector Cos* 35
7.2.5 *ComplexVector Exp* 35
7.2.6 *Vector Imaginary* 35
7.2.7 *double Modulus* 36

7.2.8 Vector Phase *36*
7.2.9 Vector Real *36*
7.2.10 ComplexVector Sin *36*
7.2.11 ComplexVector Sqrt *36*
7.3 OVERLOADED OPERATORS 36
7.3.1 + operator *36*
7.3.2 Add(ComplexVector v) *37*
7.3.3 - operator *37*
7.3.4 Subtract(ComplexVector v) *37*
*7.3.5 * operator* *37*
7.3.6 Multiply(double d) *37*
7.3.7 operator / *37*
7.3.8 Divide(double d) *38*
7.3.9 operator == *38*
7.3.10 Operator != *38*
7.4 METHODS 38
7.4.1 void Assign(ComplexVector v) *38*
7.4.2 ComplexVector Conjugate() *38*
7.4.3 ComplexMatrix ConjugateTranspose() *38*
7.4.4 ComplexVector Convolution(ComplexVector v) *38*
7.4.5 Complex Dot(ComplexVector v) *38*
7.4.6 void Edit() *39*
7.4.7 ComplexVector Fft() *39*
7.4.8 ComplexVector Ifft() *39*
7.4.9 Boolean IsZero() *40*
7.4.10 ComplexMatrix ViewAsMatrix() *40*
7.4.11 ComplexVector NonlinearSolve(ComplexNonlinearFit f, ComplexVector x,ComplexVector y, int ref $conv$) *40*
7.4.12 Complex NonlinearSolve2(ComplexNonlinearFit2 f, ComplexVector [ ] x,ComplexVector y, int ref $conv$) *42*
7.4.13 Void SetSubVector(int $i1$,int $i2$,ComplexVector v) *42*
7.4.14 ComplexVector SubVector(int $i1$,int $i2$) *42*
7.4.15 ComplexMatrix Transpose() *42*

8 MATRIX CLASS 43

8.1 CONSTRUCTORS 43
8.1.1 Matrix() *43*
8.1.2 Matrix(int m,int n) *43*
8.1.3 Matrix(Matrix A) *43*
8.2 PROPERTIES 43
8.2.1 ComplexMatrix ViewAsComplex *43*
8.2.2 Diagonal *43*
8.2.3 Max *43*
8.2.4 Min *43*
8.2.5 Modulos *44*
8.2.6 int Nrows *44*
8.2.7 Ncols *44*
8.2.8 $[i, j]$ *44*
8.2.9 MaxNumberOfLoops *44*
8.2.10 ZeroTolerance *44*
8.3 STATIC METHODS 44
8.3.1 Matrix CsvRead(String s, int $offset$) *44*
*8.3.2 Matrix CsvRead2(String **s**, int offset, char options)* *45*
8.3.3 Matrix excel_read(String s) - no longer available *45*
8.3.4 Matrix Eye(Int m,Int n) *45*
8.3.5 CalculateGivensParameters(double a,double b,double ref c,double ref s) *45*

8.3.6 Matrix Identity(int m) 46
8.4 OVERLOADED OPERATORS 46
8.4.1 + operator 46
8.4.2 Add(Matrix A) 46
8.4.3 – operator 46
8.4.4 Subtract(Matrix A) 46
8.4.5 * operator 46
8.4.6 Multiply(double d) 47
8.4.7 Multiply(Matrix A) 47
8.4.8 Multiply(Vector v) 47
8.4.9 operator / 47
8.4.10 Divide(double d) 47
8.4.11 operator == 47
8.4.12 operator != 47
8.5 METHODS 47
8.5.1 Assign 47
8.5.2 Vector BezierCurve(double t) 48
8.5.3 Vector BezierCurve(Vector w, double t) 48
8.5.4 Vector BezierCurve(double t, Matrix L, Matrix R) 49
8.5.5 Vector BezierCurve(double t, Vector w, Vector wL, Matrix L, Vector wR, Matrix R) 49
8.5.6 Bidiagonalization(Matrix U, Matrix V) 50
8.5.7 BidiagonalizationLight(Matrix U, Matrix V) 50
8.5.8 ColumnHouseholder(Vector v) 51
8.5.9 double Determinant(ref int $converged$) 51
8.5.10 double Dot(Matrix ***M***) 51
8.5.11 void Edit() 51
8.5.12 int Eigenvalues(ComplexVector E) 51
8.5.13 int Eigenvectors(ComplexVector $Eval$, ComplexMatrix $Evec$) 52
8.5.14 int OrthogonalEigenvectors(ComplexVector $Eval$, ComplexMatrix $Evec$) 52
8.5.15 double Eigenvector(Vector $Eval$, Vector $initial$, ref int $conv$) 52
8.5.16 void ElementaryRowOperation(int i, int j) 52
8.5.17 void GaussPreMultiplication(int k, Vector g) 53
8.5.18 int GaussSeidel(Vector b, Vector xo, Vector x) 53
8.5.19 Vector GetColumnVector(int i) 53
8.5.20 Vector GetRowVector(int i) 53
8.5.21 GivensPostMultiplication(int i, int k, double c, double s) 54
8.5.22 Givens_pre_multiplication(int i, int k, double c, double s) 54
8.5.23 Void HessenbergReduction() 54
8.5.24 Void HessenbergReduction(Matrix Q) 54
8.5.25 HouseholderBidiagonalization(Matrix U, Matrix V) 54
8.5.26 HouseholderQRDecmposition(Matrix Q, Matrix R) 55
8.5.27 void Image() 55
8.5.28 int Inverse(Matrix inv) 56
8.5.29 Boolean IsBidiagonal() 57
8.5.30 Boolean IsHessenberg() 57
8.5.31 Boolean IsOrthogonal() 57
8.5.32 Boolean IsZero() 57
8.5.33 int Jacobi(Vector b, Vector xo, Vector x) 57
8.5.34 void LUDecomposition(Matrix L, Matrix U, Matrix P) 58
8.5.35 Matrix Minor(int i, int j) 59
8.5.36 Matrix Pdot(Matrix ***M***) 59
8.5.37 QRDecomposition(Matrix Q, matrix R) 59

8.5.38 *Int Rank()* 59
8.5.39 *int RealSchurDecomposition(Matrix Q ,Matrix T)* 60
8.5.40 *RowHouseholder(Vector v)* 60
8.5.41 *SetColumnVector(int i ,Vector v)* 60
8.5.42 *SetDiagonal(Vector v)* 60
8.5.43 *SetRowVector(int i ,Vector v)* 60
8.5.44 *SetSubMatrix(int i_1 ,int i_2 ,int j_1 ,int j_2 ,Matrix A)* 61
8.5.45 *int SingularValues(Vector $sValues$)* 61
8.5.46 *Matrix Softmax_normalisation()* 61
8.5.47 *int SymmetricOverRelaxation(Vector b ,Vector xo, double w, Vector x)* 61
8.5.48 *Matrix SubMatrix(int i_1 ,int i_2 ,int j_1 ,int j_2)* 61
8.5.49 *int SVD(Matrix U ,Matrix S ,Matrix V)* 61
8.5.50 *int JacobiSVD(Matrix U ,Matrix S ,Matrix V)* 61
8.5.51 *int SVDLight(Matrix U ,Matrix S ,Matrix V)* 62
8.5.52 *SVDSolve(Vector b , ref int $conv$)* 63
8.5.53 *Matrix Transpose()* 63

9 COMPLEXMATRIX CLASS 64

9.1 CONSTRUCTORS 64
9.1.1 *ComplexMatrix()* 64
9.1.2 *ComplexMatrix(int m ,int n)* 64
9.1.3 *ComplexMatrix(ComplexMatrix A)* 64
9.2 PROPERTIES 64
9.2.1 *Diagonal* 64
9.2.2 *int Nrows* 64
9.2.3 *Ncols* 64
9.2.4 *[i , j]* 65
9.2.5 *Matrix Real* 65
9.2.6 *Matrix Imaginary* 65
9.3 STATIC METHODS 65
9.3.1 *CalculateGivensParameters(Complex a ,Complex b ,Complex ref c ,Complex ref s)* 65
9.4 OVERLOADED OPERATORS 66
9.4.1 *+ operator* 66
9.4.2 *Add(ComplexMatrix A)* 66
9.4.3 *– operator* 66
9.4.4 *Subtract(ComplexMatrix A)* 66
9.4.5 ** operator* 66
9.4.6 *Multiply(double d)* 67
9.4.7 *Multiply(ComplexMatrix A)* 67
9.4.8 *Multiply(ComplexVector v)* 67
9.4.9 *operator /* 67
9.4.10 *Divide(double d)* 67
9.4.11 *operator ==* 67
9.4.12 *operator !=* 67
9.5 METHODS 67
9.5.1 *Assign* 67
9.5.2 *Bidiagonalization(ComplexMatrix U ,Complexmatrix V)* 68
9.5.3 *ColumnHouseholder(Vector v)* 68
9.5.4 *ComplexMatrix ConjugateTranspose()* 68
9.5.5 *double Determinant(ref int $converged$)* 68
9.5.6 *void Edit()* 68
9.5.7 *int Eigenvalues(ComplexVector E)* 69
9.5.8 *int Eigenvectors(ComplexVector $Eval$, ComplexMatrix $Evec$)* 69
9.5.9 *ComplexMatrix Eigenvectors2(ComplexMatrix Q ,ComplexMatrix T)* 70

9.5.10 Complex Eigenvector(ComplexVector $Eval$,ComplexVector $initial$, ref int $conv$) 70
9.5.11 ComplexVector GetColumnVector(int i) 70
9.5.12 ComplexVector GetRowVector(int i) 70
9.5.13 GivensPostMultiplication(int i ,int k ,Complex c ,Complex s) 70
9.5.14 GivensPreMultiplication(int i ,int k ,Complex c ,Complex s) 70
9.5.15 void HessenbergReduction() 70
9.5.16 void HessenbergReduction(ComplexMatrix Q) 71
9.5.17 HouseholderBidiagonalization(ComplexMatrix U ,ComplexMatrix V) 71
9.5.18 HouseholderQRDecmposition(ComplexMatrix Q ,ComplexMatrix R) 71
9.5.19 int Inverse(ComplexMatrix inv) 72
9.5.20 Boolean IsBidiagonal() 72
9.5.21 Boolean IsHessenberg() 72
9.5.22 Boolean IsUnitary() 72
9.5.23 Boolean IsZero() 72
9.5.24 ComplexMatrix Minor(int i ,int j) 72
9.5.25 RealBidiagonalization(ComplexMatrix U , ComplexMatrix V) 72
9.5.26 Int Rank() 73
9.5.27 RowHouseholder(Vector v) 73
9.5.28 int SchurDecomposition(ComplexMatrix Q, ComplexMatrix T) 73
9.5.29 SetColumnVector(int i ,Vector v) 74
9.5.30 SetDiagonal(ComplexVector v) 74
9.5.31 SetRowVector(int i ,ComplexVector v) 74
9.5.32 SetSubMatrix(int i_1 ,int i_2 ,int j_1 ,int j_2 ,ComplexMatrix A) 74
9.5.33 int SingularValues(Vector $sValues$) 74
9.5.34 ComplexMatrix SubMatrix(int i_1 ,int i_2 ,int j_1 ,int j_2) 74
9.5.35 int SVD(ComplexMatrix U ,ComplexMatrix S ,ComplexMatrix V) 74
9.5.36 SVDSolve(ComplexVector b , ref int $conv$) 75
9.5.37 ComplexMatrix Transpose() 75

10 NEURAL NETWORK CLASS 76

10.1 PROPERTIES 77
10.1.1 Boolean addBias 77
10.1.2 Boolean addBias1, addBias2 78
10.1.3 Boolean accelerateConvergence 78
10.1.4 int activationFunction 78
10.1.5 double beta 78
10.1.6 int betaType 78
10.1.7 Vector bias 78
10.1.8 Vector bias1, bias2 79
10.1.9 conjugateGradientRestart 79
10.1.10 Boolean fixedStepsize 79
10.1.11 double gamma 79
10.1.12 int gammaType 79
10.1.13 double gradientTolerance 80
10.1.14 int initialGuess 80
10.1.15 Matrix inputs 80
10.1.16 double maxmu 80
10.1.17 int maxNumberOfExperiments 80
10.1.18 Boolean normaliseInputs 80
10.1.19 int numberOfFeatures 81
10.1.20 int minmu 82
10.1.21 double mu 82
10.1.22 int numberOfFeatures 82
10.1.23 Int numberOfTargets 82
10.1.24 Vector inputScale, Vector inputGain, double inputYmin, double input Ymax 82

10.1.25 double performance ... *82*
10.1.26 double performanceTolerance ... *82*
10.1.27 Boolean showGraphic ... *82*
10.1.28 int sizeOfHiddenLayer ... *82*
10.1.29 Boolean steepestDescentOnly ... *82*
10.1.30 Matrix targets ... *82*
10.1.31 Matrix weights ... *82*
10.1.32 Matrix weights1 ... *83*
10.1.33 Matrix weights2 ... *83*
10.2 METHODS ... 83
10.2.1 Matrix Augment_input(Matrix $\boldsymbol{I}$) ... *83*
10.2.2 void Calculate_normalisation_coefficients(Matrix $\boldsymbol{M}$) ... *83*
10.2.3 double Calculate_performance(Matrix $\boldsymbol{T}$, Matrix $\boldsymbol{T}$) ... *83*
10.2.4 void Calculate_softmax_gradient_function(Matrix $\boldsymbol{W}$, Matrix I,Matrix T) ... *83*
10.2.5 void Calculate_softmax_gradients_function(Matrix $\boldsymbol{W1}$, Matrix $\boldsymbol{W2}$, Matrix $\boldsymbol{I}$, Matrix $\boldsymbol{T}$) ... *83*
10.2.6 void Perform_normalisation(Matrix $\boldsymbol{M}$) ... *83*
10.2.7 Matrix Softmax_transfer_function(Matrix $\boldsymbol{M}$) ... *83*
10.2.8 double tansig(double $\boldsymbol{z}$) ... *83*
10.2.9 Matrix Tansig_transfer_function(Matrix $\boldsymbol{M}$) ... *83*
10.2.10 void train() ... *84*
10.2.11 void trainHL() ... *84*

11 GRAPH CLASS ... 86

11.1 EXAMPLES ... 86
11.1.1 Example 1 ... *86*
11.1.2 Example 2 ... *86*
11.2 CONSTRUCTOR ... 88
11.2.1 Graph() ... *88*
11.3 METHODS ... 88
11.3.1 Histogram(Vector $\mathbf{v}$) ... *88*
11.3.2 Histogram(Vector $\mathbf{v}$, int $\mathbf{N}$) ... *88*
11.3.3 Histogram(Vector $\mathbf{v}$, Vector $\mathbf{bins}$) ... *88*
11.3.4 Plot(Vector v) ... *89*
11.3.5 Plot(Vector v ,String $colour$,String $style$,int $weight$) ... *89*
11.3.6 Plot(Vector v ,Vector w) ... *90*
11.3.7 Plot(Vector v ,Vector w , String $colour$,String $style$,int $weight$) ... *90*
11.3.8 Plot(Matrix m) ... *90*
11.3.9 Plot(Matrix m ,Vector v) ... *90*
11.3.10 Plot(Matrix m , int Nc) ... *90*
11.3.11 Plot(Matrix m ,Vector X ,Vector Y) ... *90*
11.3.12 Plot (Matrix m ,Vector X ,Vector Y ,Vector v) ... *90*
11.3.13 Plot (Matrix m ,Vector X ,Vector Y ,int Nc) ... *91*
11.3.14 Plot (Matrix m, Vector X, Vector Y, int Nc, Color[] $colours$) ... *91*
11.3.15 AddImage(String $name$, Bitmap $image$, int $blend$) ... *91*
11.3.16 Size (int $width$,int $height$) ... *91*
11.3.17 AddAnnotation(double x ,double y ,String s) ... *91*
11.3.18 AddTitle(String s) ... *91*
11.3.19 AddXLabel(String s) ... *91*
11.3.20 AddYLabel(String s) ... *91*
11.3.21 AddLegend(String s) ... *91*
11.3.22 AddXLlimits(double $left$,double $right$) ... *91*
11.3.23 AddYlimits(double $bottom$,double top) ... *91*
11.3.24 AddXInterval(double itr) ... *91*
11.3.25 AddYInterval(double itr) ... *91*
11.3.26 Fill (Boolean $fill$) ... *91*

11.3.27 Fill (Boolean fill, Boolean contours, int fillSensitivity) ... 92
11.3.28 Cls() ... 92

12 GRAPHEXPORT CLASS ... 93

12.1 CONSTRUCTOR ... 93
12.1.1 GraphExport(Matrix Z, Vector X, Vector Y, int Nc) ... 93
12.2 METHODS ... 93
12.2.1 Plot() ... 93
12.2.2 Fill(Boolean $fill$, Boolean $contours$, int $fillSensitivity$) ... 93
12.2.3 AddImage(string $name$, Bitmap image, int $blend$) ... 93

13 GRAPH3D CLASS ... 95

13.1 CONSTRUCTOR ... 96
13.1.1 Graph3D() ... 96
13.2 PROPERTIES ... 96
13.2.1 Boolean drawAxes ... 96
13.2.2 Boolean drawGrid ... 96
13.2.3 int numberOfTicks ... 96
13.3 METHODS ... 96
13.3.1 void AddXAxisLimits(double XMin, double XMax) ... 96
13.3.2 void AddYAxisLimits(double YMin, double XMin) ... 97
13.3.3 void AddZAxisLimits(double ZMin, double YMin) ... 97
13.3.4 void BackgroundColour(Color backgroundColour) ... 97
13.3.5 Cls() ... 97
13.3.6 Plot(Vector x, Vector y, Vector z, Vector s) ... 97
13.3.7 Plot(Vector x, Vector y, Vector z, Color c) ... 97

14 VIDEO CLASS ... 98

14.1 CONSTRUCTOR ... 98
14.1.1 Video() ... 98
14.2 METHODS ... 98
14.2.1 void AddSeries(Vector $\boldsymbol{v}$) ... 98
14.2.2 Void PlayVideo(Vector t) ... 98
14.2.3 void AddMatrix(Matrix m) ... 99
14.2.4 void PlayMatrixVideo(Vector t) ... 99
14.2.5 void AddRealTimeMatrix(Matrix m) ... 99
14.2.6 void AddRealTimePoint(int index, double pt, double t) ... 100
14.2.7 void AddRealTimeSeries(int index, Vector x,Vector y) ... 100
14.2.8 void AddRealTimeSeries(int index, Vector x, Vector y, String color, String style, int weight). 101
14.2.9 void AddRealTimeMatrix(Matrix m) ... 102
14.2.10 void AddTitle(String title) ... 103
14.2.11 void AddXLabel(String xlabel) ... 103
14.2.12 void AddXLimits(double left, double right) ... 103
14.2.13 void AddYLimits(double top, double bottom) ... 103
14.2.14 void AddXInterval(double interval) ... 103
14.2.15 void AddYInterval(double interval) ... 103
14.2.16 void AddLegend(String s) ... 103
14.2.17 void AddAnnotation(double x, double y , String ann) ... 103
14.2.18 Void AddAnnotation(double x, double y, String ann, String colour) ... 103
14.2.19 Void Cls() ... 104
14.2.20 Void DisplayLegends(Boolean display) ... 104
14.2.21 Void SetBackgroundColour(String colour) ... 104
14.2.22 void SetContrast(byte contrast) ... 104

15 DELEGATE TYPES ... 104

15.1.1 delegate Vector NelderMeanFit(Vector $\boldsymbol{w}$, Vector $\boldsymbol{beta}$) ... 104
15.1.2 delegate Vector NonlinearFit(Vector w, Vector $beta$) ... 104
15.1.3 delegate double NonlinearFit2(Vector w, Vector $beta$) ... 104
15.1.4 delegate ComplexVector ComplexNonlinearFit(ComplexVector w, ComplexVector $beta$) 104

15.1.5 *delegate Complex ComplexNonlinearFit2(ComplexVector w, ComplexVector $beta$)* 104
17 REFERENCES 105
18 CONTACT 105

1 Introduction

This document describes the constructors methods and properties which are included in ‘Matrix.dll’ and which may be accessed in any Microsoft .NET environment. There are six classes provided: Vector, Matrix, Graph, Complex, ComplexVector and ComplexMatrix. These perform many common matrix and vector manipulations using both real and complex arithmetic; and allow graphing and videos of the results.

In particular there is the inclusion of:

(i) The powerful SVD calculation which permits the solution of (in a least squares sense) any linear system of the form $Ax = b$.
(ii) The Fast Fourier transform which permits a spectral analysis of signals.
(iii) A powerful non-linear system solver.
(iv) Solution of Eigenvector and Eigenvalues of a square matrix.

2 Version history

2.1 Version 1.0.0.0

1.0.0.0 (9th April 2013)	Initial Version
1.0.0.2 (11th April 2013)	Correction to the bidiagonalisation routine
1.0.0.3 (12th April 2013)	Correction to the + Vector operator
1.0.0.4 (13th April 2013)	Correction to the SVD solve routine
1.0.0.5 (24th April 2013)	Addition of sort function, addition of singular_values function, addition of Determinant function, addition of Minor function. Improvement of the speed of convergence of routines involving the SVD routine.
1.0.0.6 (24th April 2013)	Correction of a bug in the is_bidiagonal() routine
1.0.0.7 (25th April 2013)	Correction of the prototype of the singular_values routine.
1.0.0.8 (15th May 2013)	Addition of SVD_light and the related bidiagonalisation routine. Addition of coordinate transformation functions. Addition of is_orthogonal() function.

2.2 Version 1.0.*.*

1.0.1.0 (30th December 2013)	Addition of Graph class. Addition of excel read function to matrix class.
1.0.1.1 2nd January 2014	Addition of Complex and ComplexVector classes permitting the manipulation of complex numbers. Addition of various componentwise functions.
1.0.1.2 9th January 2014	Addition of further properties and operators.
1.0.1.2 13th January 2014	Addition of non linear solve method to the Vector class.

1.0.1.3 28th January 2014	Addition of ComplexMatrix class including a complex SVD and solve routines. Addition of non linear solve method to the ComplexVector class.
1.0.1.4 3rd February 2014	Addition of function to read a comma separated variable file.
1.0.1.5 20th February 2014	Correction of bug on csv_read and excel_read functions that arose when the input data was non-numeric.
1.0.1.6 2nd April 2014	Addition of namespace 'MathematicalServices' to all classes. Removal of excel read function to make the code CLS compliant. Change of name of non_linear_solve routines to nonlinear_solve. Nonlinfit and complexnonlinfit are now no longer subclasses. 'Calculate_Givens_parameters' declared as a static function within the Matrix and ComplexMatrix classes. Signing of the dll so it can be put in the GAC.
1.0.1.7 10th April 2014	Major refactoring to enable the code and exported properties and methods to meet Microsoft naming, style and design guidelines for managed code. Provision of functions for those languages which do not support overloaded operators.
1.0.1.8	Addition of JacobiSVD routine
1.0.1.9	Addition of matrix (contour) plotting together with cosmetic improvements to all plotting routines.
1.0.1.10	Addition of Bezier curves to the matrix class. Improvements to the SVDsolve routine.
1.0.1.11 13th May 2014	Speed improvements to those routines using an SVD decomposition.
1.0.1.12 16th May 2014	Further improvements to the SVD related routines.
1.0.1.13 23rd May 2014	Further speed improvements to the real SVD routines
1.0.1.15 3rd June 2014	Speed improvement to SVDLight routine.
1.0.1.16 10th June 2014	Addition of colour, style and weight properties to plotted vectors. Addition of a SaveAs function to save a graph in a prescribed format.
1.0.1.19 11th August 2014	Addition of Edit method to matrix class to permit interactive editing of the elements of a matrix.
1.0.2.0 28th August 2014	Addition of eigenvectors and eigenvalues routines.
1.0.2.2 8th September 2014	Improvements to the convergence of the complex SVD routine.
14th September 2014	Addition of Bidiagonalization routine to ComplexMatrix class.
1.0.2.5 15th September 2014	Addition of Orthogonal Eigenvectors calculation method. Improvements to the convergence of the complex SVD method
1.0.2.6 30th September 2014	Addition of vector Edit method.
1.0.2.7 7th October	Addition of Image method
1.0.2.8 21st October 2014	Improvements to the Image method.

1.0.2.9 26th October 2014	Change of labels on the Image method
1.0.2.10 29th October 204	Addition of fill method for contour plotting.
1.0.2.11 29th October 2014	Determinant method is now calculated via the SVD and is present in the paid for version only.
1.0.2.12 31st October 2014	Change of the signature of the SingularValues method so that it does not invoke call by reference.
1.0.2.13 3rd November 2014	Addition of annotations to the graph class.
1.0.2.14 10th November 2014	Addition of singular vectors and values plotting option to the Matrix image method.
1.0.2.15 17th November 2014	Change of prototype of Eigenvalues function.
1.0.2.16 4th April 2015	Addition of LU decomposition.
1.0.2.17 17th April 2015	Addition of zoom and scroll properties to the graph class. Addition of NonlinearFit2 function. Addition of Eigenvector routine for finding a single eigen pair.
1.0.3.0 22nd April 2015	Separation of versions locked to one computer and one that is not.
1.0.3.1 27th April 2015	Bug fix to the image function (eigenvectors when the matrix is not square).
1.0.3.1	Addition of 3D plot, movie classes, various bug fixes.

2.3 Manual History

1st April 2013	Initial document.
10th April 2013	Correction to the expression for A in section 8.5.49. Correction to section 8.5.6
11th April 2013	Removal of the manual from the distributed zip file. The up-to-date manual is now available online to download. Removal of the restriction, $m \geq n$, in section 8.5.6.
12th April 2013	Addition of Eye matrix.
24th April 2013	Addition of description of new functionality of 1.0.0.5. Addition of a table of contents.
24th April 2013	Addition of Version 1.0.0.6
27th April 2013	Corrections to the SVD solve routine description.
15th May 2013	Addition of Version 1.0.0.8 information.
21st June 2013	Addition of scalar division operator for the matrix class (manual omission only).
23rd December	Addition of Graph class. Addition of function to read an Excel sheet into

2013	a matrix.
1st January 2014	Corrections to introduction and version history.
2nd January 2014	Addition of Complex and ComplexVector classes.
9th January 2014	Addition of further properties and operators.
13th January 2014	Addition of non linear solve function to the Vector class
20th January 2014	Addition of Householder bidiagonalisation function to the manual.
28th January 2014	Addition of ComplexMatrix class including a complex SVD and solve routines. Addition of non linear solve method to the ComplexVector class.
3rd Febraury 2014	Addition of function to read a comma separated variable file.
26th March	Renaming 'max_no_SVD_loops' to 'max_no_loops'
2nd April 2014	Addition of namespace 'MathematicalServices' to all classes. Removal of excel read function to make the code CLS compliant. Change of name of non_linear_solve routines to nonlinear_solve. Nonlinfit and complexnonlinfit are now no longer subclasses. 'Calculate_Givens_parameters' declared as a static function within the Matrix and ComplexMatrix classes. Signing of the dll so it can be put in the GAC.
10th April 2014	Major refactoring to enable the code and exported properties and methods to meet Microsoft naming, style and design guidelines for managed code. Provision of functions for those languages which do not support overloaded operators.
18th April 2014	Addition of JacobiSVD routine.
2nd May 2014	Addition of matrix (contour) plotting together with cosmetic improvements to all plotting routines.
9th May 2014	Addition of Bezier curves to the matrix class.
16th May 2014	Enhancements to the BidiagonalizationLight and SVDLight routines.
23rd May 2014	Speed improvements to real SVD routines.
10th June 2014	Addition of colour, style and weight properties to plotted vectors. Addition of a SaveAs function to save a graph in a prescribed format.
11th August 2014	Addition of Edit method to matrix class to permit interactive editing of the elements of a matrix.
28th August 2014	Addition of Eigenvectors and Eigenvalues routines.
14th September 2014	Addition of Bidiagonalization routine to ComplexMatrix class.
15th September 2014	Addition of Orthogonal Eigenvectors calculation method. Improvements to the convergence of the complex SVD method
30th September 2014	Addition of vector Edit method.

7th October 2014	Addition of image method to matrix class.
21st October 2014	Improvements to the Image method description.
26th October 2014	Change of labels on the Image method.
29th October 2014	Addition of fill method for contours. Formatting improvements to the contour plotting routines.
29th October 2014	Determinant method is now calculated via the SVD and is present in the paid for version only
3rd November 2014	Addition of annotations to the Graph class.
4th April 2015	Addition of LU decomposition.
17th April 2015	Addition of zoom and scroll properties to the graph class. Addition of NonlinearFit2 function. Addition of Eigenvector routine for finding a single eigen pair.
22nd April 2015	Separation of versions locked to one computer and one that is not.
14th June 2017	Addition of Video class, Addition of Graph3D class.
7th April 2017	Addition of Nelder Mean solve function.
17th April 2020	Addition of Neural Network class.
29th December 2024	Addition of GraphExport class. Various additions to filling in contour maps.

3 License Conditions

The vendor of this software makes no warranty that the software is free from defects or that it is suitable for any particular purpose – although efforts have been made to ensure that it is so. It is the user’s responsibility to verify the accuracy of the software and the calculations performed, for any particular task.

The use of this dll is governed by English law.

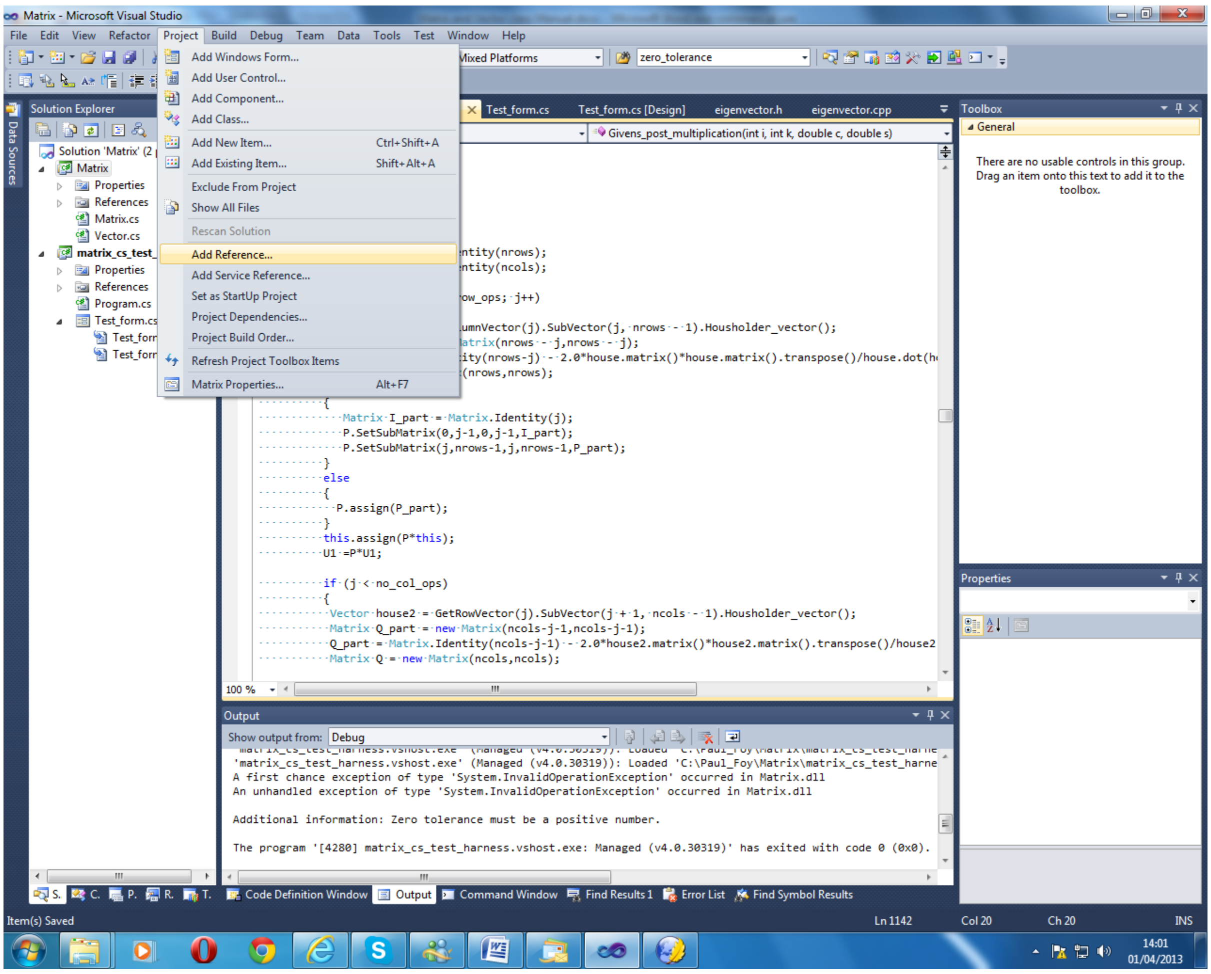
4 Installation Instructions

Unzip the downloaded file to reveal:

- The Matrix.dll file

Use of this software implies acceptance of the license condition contained in the license file and above.

To access the method and properties in the dll add it as a reference to you Microsoft C# .NET, VB.NET or C++/CLI .NET project:



```
ntity(nrows);
ntity(ncols);

ow_ops; j++)

umnVector(j).SubVector(j, nrows - 1).Housholder_vector();
atrix(nrows - j,nrows - j);
ity(nrows-j) - 2.0*house.matrix()*house.matrix().transpose()/house.dot(h
(nrows,nrows);
          {
            Matrix I_part = Matrix.Identity(j);
            P.SetSubMatrix(0,j-1,0,j-1,I_part);
            P.SetSubMatrix(j,nrows-1,j,nrows-1,P_part);
          }
          else
          {
           P.assign(P_part);
          }
          this.assign(P*this);
          U1 =P*U1;

          if (j < no_col_ops)
          {
           Vector house2 = GetRowVector(j).SubVector(j + 1, ncols - 1).Housholder_vector();
           Matrix Q_part = new Matrix(ncols-j-1,ncols-j-1);
           Q_part = Matrix.Identity(ncols-j-1) - 2.0*house2.matrix()*house2.matrix().transpose()/house2
           Matrix Q = new Matrix(ncols,ncols);
```

```
'matrix_cs_test_harness.vshost.exe' (Managed (v4.0.30319)): Loaded 'C:\Paul_Foy\Matrix\matrix_cs_test_harne
A first chance exception of type 'System.InvalidOperationException' occurred in Matrix.dll
An unhandled exception of type 'System.InvalidOperationException' occurred in Matrix.dll

Additional information: Zero tolerance must be a positive number.

The program '[4280] matrix_cs_test_harness.vshost.exe: Managed (v4.0.30319)' has exited with code 0 (0x0).
```

It is recommended that each file of your C# code contains the statement:

```
using MathematicalServices;
```

Otherwise each class in this document will have to be prefixed (such) as:

```
MathematicalServices.Matrix m = new
MathematicalServices.Matrix(3,4);
```

5 Vector class

This class permits manipulation of arbitrary length vectors, whose elements are real numbers.

5.1 Constructors

5.1.1 **Vector()**

This default constructor initialises a 3-dimensional vector and sets all of its elements to zero. The index of the array of elements is zero based.

C# example:

```
Vector v = new Vector();
```

5.1.2 **Vector(int n)**

This constructor initialises a vector of dimension n and sets all of its elements to zero. The index of the array of elements is zero based.

C# example:

```
Vector v = new Vector(4);
```

5.1.3 **Vector(Vector v)**

This constructor initialises a vector having the same dimensions as v, and also having the same elements as v.

C# example:

```
Vector v1 = new Vector(3);
v1[1]=1.5;
Vector v2 = new Vector(v1);
```

5.2 Properties

5.2.1 **Vector Abs**

Returns a vector whose elements are the absolute value of the corresponding elements of the parent vector.

C# example:

```
Vector r = v.Abs;
Vector s = v.Exp;
Vector t = v.Cos;
Vector u = v.Sin
```

5.2.2 **ComplexVector ViewAsComplex**

Allows the parent vector to be regarded as a vector whose elements are complex numbers.

C# example:

```
Matrix mat = Matrix.CsvRead(str);

Vector r = mat.GetColumnVector(1);

ComplexVector R = new ComplexVector(r.Dim);

ComplexVector rc = r.ViewAsComplex;

R = rc.Fft();
```

5.2.3 **Vector Cos**

Returns a vector whose elements are the cosine of the corresponding elements in the parent vector. See 5.2.1.

5.2.4 **int Dim**

Returns the dimension of the vector – read only.

C# example:

```
int dimension;

Vector v = new Vector(3);

dimension = v.Dim
```

5.2.5 **Vector Exp**

Returns a vector whose elements are the exponential of the corresponding elements of the parent vector. See 5.2.1

5.2.6 **[i]**

Gets and sets the i th element of the vector. Here i is greater than or equal to zero and less than the dimension of the vector. If access outside of this range is requested an error is thrown.

C# example:

```
double element1, element2;

element1=1.5;

v[0]=element1;

element2=v[1];
```

5.2.7 **Vector Log**

Returns a vector whose elements are the log to the base 10 of the corresponding elements of the parent vector.

5.2.8 **double Max**

Returns the maximum of all the elements of the vector.

5.2.9 double Min

Returns the minimum of all the elements of the vector.

5.2.10 double Mean

Returns the mean of all the elements of the vector.

C# example:

```
double mn;

mn = v.Mean;
```

5.2.11 Modulus

Read only property to return the modulus or 2-norm of the vector. If $v = (v_0, v_1, ..., v_{n-1})$ then the modulus is: $\|v\| = \sqrt{v_0 * v_0 + ... + v_{n-1} * v_{n-1}}$.

C# example:

```
If (v.Modulus < 1.0e-7)

{

   break

}
```

5.2.12 Vector Sin

Returns a vector whose elements are the sine of the corresponding elements of the parent vector. See5.2.1.

5.2.13 double Sd

Returns the standard deviation of the elements of the vector. This is the quantity:

$$\sigma = \sqrt{\frac{\sum_{i-1}^{N} (v_i - \mu)^2}{N}}$$

where µ is the mean of the data.

C# example:

```
double s;

s = v.Sd;
```

5.2.14 UnitVector

This returns $\frac{v}{\|v\|}$. If v is the zero vector an error is thrown.

C# example:

```
If ( Math.Acos(v1.UnitVector[2]) > 0.999 )

{
  // vector is perpendicular
}
```

5.3 Overloaded operators

5.3.1 + operator

If v_1 and v_2 are vectors this permits the expression $v_1 + v_2$ to return a vector. If v_1 and v_2 are not of the same dimension an error is thrown.

C# example:

```
Vector v = v1 + v2;
```

5.3.2 Add(Vector v)

Adds v to the parent. To be used in languages that can not overload +.

C# example:

```
Vector v = v1.Add(v2);
```

5.3.3 – operator

If v_1 and v_2 are vectors this permits the expression $v_1 - v_2$ to return a vector. . If v_1 and v_2 are not of the same dimension an error is thrown.

C# example:

```
Vector v = v1 - v2;
```

5.3.4 Subtract(Vector v)

Subtracts v from the parent. To be used in languages that can not overload -.

C# example:

```
Vector v = v1.Subtract(v2);
```

5.3.5 * operator

If v is a vector and λ a scalar this permits both the expression $\lambda * v$ and $v * \lambda$ to return a vector. The multiplication of the vector by the scalar is component-wise.

C# example:

```
double lambda1 = 1.4;

double lambda2 = 0.6;

Vector v = (lambda1 * v1) * lambda2;
```

5.3.6 Multiply(double d)

Multiplies the parent by d. To be used in languages that can not overload *. d may also be complex.

C# example:

```
Vector v = v1.Multiply(d);
```

5.3.7 operator /

If v is a vector and λ a scalar this permits the expression v/λ to return a vector. If λ is zero an error is thrown. The division of the vector by the scalar is component-wise.

C# example:

```
double lambda = 2.0;

Vector v = v1 / lambda;
```

5.3.8 Divide(double d)

Divides the parent by d. To be used in languages that can not overload /. d may also be complex.

C# example:

```
Vector v = v1.Divide(d);
```

5.3.9 operator ==

Returns true if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise false.

C# example:

```
Vector jumble = new Vector(5);
Vector sorted = new Vector(5);
int[] loc     = new int[5];

jumble[0] = 5; jumble[1] = 1; jumble[2] = 4; jumble[3] = 2;
jumble[4] = 8;
sorted[0] = 5; sorted[1] = 1; sorted[2] = 4; sorted[3] = 2;
sorted[4] = 8;

if (sorted == jumble)
{
   ;
}
else if (sorted != jumble)
{
   sorted.Assign(jumble.Sort(loc));
}
```

5.3.10 operator !=

Returns false if two classes are identical. Otherwise returns true.

C# example:

See 5.3.9.

5.4 Methods

5.4.1 **Assign**

If v_1 and v_2 are two vectors of the same dimension this method is the C# way of performing $v_1 = v_2$. If v_1 and v_2 are not of the same dimension an error is thrown.

C# example:

```
v1.Assign(v2);
```

5.4.2 **Vector ConvertVectorFromLocalSystem(Matrix M)**

This function converts a vector expressed in a local coordinate system to that of a new coordinate system. The vector returned is that of the vector expressed in the new coordinate system. M is an orthogonal transformation matrix whose columns represent the axes of the local system expressed in terms of the coordinate system that we are transferring to.

```
Matrix M = new Matrix(3, 3);
M.SetColumnVector(0, x_axis);
M.SetColumnVector(1, y_axis);
M.SetColumnVector(2, z_axis);

Vector CN_new  = new Vector(CN_local.ConvertVectorFromLocalSystem(M));
```

5.4.3 **Vector ConvertVectorToLocalSystem(Matrix M)**

This function converts a vector to that of a local coordinate system. M is an orthogonal transformation matrix whose columns represent the axes of the local system expressed in terms of the coordinate system that we are converting from.

5.4.4 **Vector Convolution(Vector v)**

If u is the parent vector of length m and v has length n then this method returns the vector w of length $m+n-1$ whose i th element is:

$$w[i] = \sum_j u[j]v[i-j+1]$$

Here the summation is over all values of j which give rise to legal subscripts for $u[j]$ and $v[i-j+1]$.

C# example:

```
h = f.Convolution(g);
```

5.4.5 **Vector Crosscorrelation(Vector v, Boolean $normalise$)**

This method returns the cross correlation of two vectors of the same length. If the vectors are of a different length an error is thrown.

If u is the parent vector of length N and v is also a vector of this length this method computes the vector w of length $2N-1$. The nth element $(-N+1 \le n \le N-1)$ is:

$$w[n] = \left\{ \frac{\sum_{i=0}^{N-n-1} u(i)v(n+i) if n \ge 0}{\sum_{i=-n}^{N+n-1} u(i)v(n+i) if n < 0} \right\}$$

The actual vector returned $f*g$ is such that $f*g[i] = w[-N+1+i] \;\; (0 \le i < 2N-1)$.

If $normalise$ is set to true $w[n]$ is divided by $N-|n|$.

5.4.6 double Dot

Returns the dot product with another vector v. If v is not of the same dimension as the parent an error is thrown.

C# example:

```
if ( Math.Acos( v1.Dot(v2) ) > 0.99999 )

{

   // vectors are parallel

}
```

5.4.7 void Edit()

Displays a dialogue in which it is possible to view the vector and edit its contents. To change a cell: select it, type in the new value and click elsewhere so the focus of the cell is lost. Press the update button to commit the changes to the vector. Press the cancel button to quit the form without committing any changes.

C# example:

```
 Vector v = new Vector(4);
 v.Edit();
```

5.4.8 Vector GaussVector()

Given an n dimensional vector x, this method returns an n dimensional vector v with the property that:

i. $v[0] = 1.0$

ii. $v[i] = -x[i]/x[0]$ for $i = 1$ to $n-1$

If $x[0]$ is zero an error is thrown.

C# example:

```
Vector x = new Vector(5);

For (int i = 0; i < x.Dim; i++)

{

  x[i] = i + 1.0;
}
Vector v = x.GaussVector();
```

5.4.9 Vector HouseholderVector()

Given an n dimensional vector x this method returns an n dimensional vector v with the property that:

i. $v[0] = 1.0$

ii. $\left(I - 2\frac{vv^T}{v^Tv}\right)x$ is zero in all but the first component (here I_n is the n by n identity matrix)

C# example:

```
Vector x = new Vector(5);
For (int i = 0; i < x.Dim; i++)
{
  x[i] = i + 1.0;
}
Vector v = x.HouseholderVector();
```

5.4.10 Boolean IsMonotonic()

Returns true if the elements of the vector are monotonic increasing. Otherwise it returns false.

5.4.11 Boolean IsZero()

Returns true if every element of the parent is zero to within the tolerance of Matrix.ZeroTolerance.

5.4.12 Matrix ViewAsMatrix()

This returns an n by 1 matrix from an n dimensional vector. This method is provided so that a vector can be used in matrix computations such as in ii of 5.4.9.

C# example:

```
Vector x = new Vector(5);
For (int i = 0; i < x.Dim; i++)
{
  x[i] = i + 1.0;
}
Vector v = x.HouseholderVector();
Matrix P = Matrix.Identity(5)-    ...
2.0*v.ViewAsMatrix()*(v.ViewAsMatrix().Transpose())/v.Dot(v);
```

5.4.13 **double MaxL(ref int loc)**

Returns the maximum of the elements of the vector. loc is set to be the index location at which this maximum occurs.

C# example:

```
ComplexVector v = new ComplexVector(10);
...
int loc_max = 0;
int loc_min = 0;
double max = v.Real.MaxL(ref loc_max);
double min = v.Real.MinL(ref loc_min);
```

5.4.14 **double MinL(ref int loc)**

Returns the minimum of the elements of the vector. loc is set to be the index location at which this minimum occurs.

C# example:

See 5.4.14.

5.4.15 **Vector NelderMeanSolve(NelderMeanFit $\boldsymbol{f}$, Vector $\boldsymbol{h}$, Vector $\boldsymbol{x}$, Vector $\boldsymbol{y}$, int ref $\boldsymbol{conv}$)**

Let f be a function mapping $\Re^m \times \Re^n$ to $\Re^p$, where m, n and p are positive integers. Let f_i be the projection of f onto the i th component of $\Re^p$ ($1 \le i \le p$). Let $x \in \Re^m$ and $y \in \Re^p$

Then this function returns $\beta \in \Re^n$ such that $S(\beta) = \| y - f(x,\beta) \|^2 = \sum_{i=1}^{p} \| y_i - f_i(x,\beta) \|^2$ is a minimum. The algorithm used is the Nelder-Mead simplex method. An initial vertex, β_1 is provided by the parent vector. n further vertices are provided by the n dimensional vector h, such that $\beta_i = \beta_1 + e_i h_{i-1}$, $2 \le i \le n+1$, where e_i is a unit vector.

The technique employed is an iterative one involving successive replacements of the vertices until one with sufficiently small residual is obtained. At this point the algorithm terminates returning the vertex with the smallest residual and setting $conv$ to 1.

If the maximum number of iterations, as given by Matrix.MaxNumberOfLoops, is exceeded, the algorithm terminates with $conv$ set to 0.

C# examples:

```
int noParameters = 6;
Vector pFit = new Vector(noParameters);
Vector pInit = new Vector(noParameters);
NelderMeadFit fn = new NelderMeadFit(vlosRotorD);
Vector h = new Vector(noParameters);
h[0] = 10.0 * Math.PI / 180.0;
h[1] = -10.0;
h[2] = 2.0;
h[3] = -3.0;
h[4] = -10.0 * Math.PI / 180.0;
h[5] = 10.0 * Math.PI / 180.0;
Matrix.ZeroTolerance = 1.0E-4;
Matrix.MaxNumberOfLoops = 200;
pFit = pInit.NelderMeadSolve(fn, h, x, y, ref conv);
```

5.4.16 **Vector NonlinearSolve(NonlinearFit f Vector x,Vector y, int ref $conv$)**

Let f be a function mapping $\Re^m \times \Re^n$ to $\Re^p$, where m, n and p are positive integers. Let f_i be the projection of f onto the i th component of $\Re^p$ $(1 \le i \le p)$. Let $x \in \Re^m$ and $y \in \Re^p$

Then this function returns $\beta \in \Re^n$ such that $S(\beta) = \| y - f(x,\beta) \|^2 = \sum_{i=1}^{p} \| y_i - f_i(x,\beta) \|^2$ is a minimum.

The technique employed is an iterative one involving successive approximations β_k to β. The first guess for β is the value of the parent vector. The algorithm terminates successfully ($conv$ set to 1) if $\left\| \beta_k - \beta_{k-1} \right\|$ < Matrix.ZeroTolerance, for some $k <$ Matrix.MaxNumberOfLoops. If the maximum number of iterations, as given by Matrix.MaxNumberOfLoops, is exceeded, the algorithm terminates with $conv$ set to 0.

This function is a generalisation of the classical Levenberg-Marquardt problem in which we have a function h mapping $\Re \times \Re^n$ to $\Re$, together with a 2-D data set (x_i, y_i) and we are trying to minimise $S(\beta) = \sum_{i=1}^{p} | y_i - h(x_i, \beta) |^2$. Section 5.4.17 gives an alternative algorithm which is more computationally intensive to implement, when the full generality is not needed. You should decide carefully whether to use 5.4.16 or 5.4.17 as 5.4.17 is usually much quicker.

C# examples:

Example 1

The following uses the example of [2], section 10.2.

```
Vector pInit = new Vector(3);
int conv = 0;
pInit[0] = 0.4;
pInit[1] = 0.1;
pInit[2] = -0.4;

Vector y = new Vector(3); //ensures y is set to zero
Vector x = new Vector(3); //not used in this example


NonlinearFit fn = new NonlinearFit(burdenfairesexample);

Matrix.ZeroTolerance = 1.0E-8;
Vector pFit = pInit.NonlinearSolve(fn, x, y, ref conv);

//pFit = (0.5, 0.0, -0.52359)


private Vector burdenfairesexample(Vector x, Vector b)
```

```
{
    Vector rtn = new Vector(3);

    rtn[0] = 3.0*b[0]-Math.Cos(b[1]*b[2])-0.5;
    rtn[1] = b[0]*b[0] - 81.0*(b[1]+0.1)*(b[1]+0.1)+Math.Sin(b[2])+1.06;
    rtn[2] = Math.Exp(-1.0*b[0]*b[1])+20.0*b[2]+(10.0*Math.PI-3.0)/3.0;

    return rtn;
}
```

Figure 1

Example 2

The following *pseudocode* shows how the routine may be used to find the parameters describing a change of coordinate system (transformation matrix plus origin translation - 12 parameters), by fitting measured data points to a datum sphere with known centre and radius. In this example m=3*no_points, n=12 and p = 4 (radius and centre of datum sphere).

```
Vector x = new Vector (3*no_points);
//Populate x with the point cloud data of the original coordinate
system
Vector y = new Vector (4);
y[0] = datum_sphere_radius;
y[1] = data_sphere_centre[0];
y[2] = data_sphere_centre[1];
y[3] = data_sphere_centre[2];
NonlinearFit fn = new NonlinearFit(sphere_fit);
Matrix.ZeroTolerance = 1.0E-5;
Matrix.MaxNumberOfLoops = 30;
int conv = 0;
Vector pFit = new Vector(12);
Vector pInit = new Vector(12);
pInit[0]=pInit[4]=pInit[8]=1.0; //(transformation matrix guess is
//a unit matrix. Change of origin is the zero vector.
pFit.Assign(pInit.NonlinearSolve(fn,x,y,ref conv));

//pFit contains the parameters

private Vector sphere_fit(Vector x, Vector b)
{
    Vector rtn = new Vector (4);
    Sphere3D sphere = new Sphere3D();
    //fill array 'points' of Point3D's, in new coordinate system
with the data from x using b.
    Point3DArray array = new Point3DArray(points, x.Dim/3);
    array.FitSphere(sphere, ref conv);
    rtn[0] = sphere.Radius;
    rtn[1] = sphere.Centre[0];
    rtn[2] = sphere.Centre[1];
    rtn[3] = sphere.Centre[2];

    return rtn;
}
```

Figure 2

5.4.17 **Vector NonlinearSolve2(NonlinearFit2** f **Vector[ ]** x **,Vector** y **, int ref** $conv$ **)**

Let f be a function mapping $\Re^m \times \Re^n$ to $\Re$, where m and n are positive integers. Let $x_i \in \Re^m$ and $y_i \in \Re$, $1 \le i \le p$, where p is the dimension of y (or the size of the collection x). Then this function returns $\beta \in \Re^n$ such that $S(\beta) = \sum_{i=1}^{p} \| y_i - f(x_i, \beta) \|^2$ is a minimum.

The technique employed is an iterative one involving successive approximations β_k to β. The first guess for β is the value of the parent vector. The algorithm terminates successfully ($conv$ set to 1) if $\|\beta_k - \beta_{k-1}\|$<Matrix.ZeroTolerance, for some $k <$ Matrix.MaxNumberOfLoops. If the maximum number of iterations, as given by Matrix.MaxNumberOfLoops, is exceeded, the algorithm terminates with $conv$ set to 0.

C# example:

In this example m = 3, n = 7 and p = no_points.

```
Vector pInit = new Vector(7);
pInit[0] = 12.0;   //radius, actual is 10.0
pInit[1] = 1.1;    //centre, actual is 1.0
pInit[2] = 2.2;    //centre, actual is 2.0
pInit[3] = 3.1;    //centre, actual is 3.0
pInit[4] = normal[0]; //axis
pInit[5] = normal[1];
pInit[6] = normal[2];

Vector y = new Vector(no_points);   //Create the zero vector

Vector[] r = new Vector[no_points];
for (i = 0; i < no_points; i++)
{
   r[i] = new Vector(3);
   r[i][0] = points[i][0];
   r[i][1] = points[i][1];
   r[i][2] = points[i][2];
}

NonlinearFit2 fn = new NonlinearFit2(distance_from_point_to_circle);
Matrix.ZeroTolerance = 1.0E-5;
Matrix.MaxNumberOfLoops = 30;
int conv = 0;
Vector pFit = new Vector(7);
pFit.Assign(pInit.NonlinearSolve2(fn,r,y,ref conv));



private Vector distance_from_point_to_circle(Vector r, Vector b)
{
   double rtn;

   //construct the circle
   double radius = b[0];
   Point3D centre = new Point3D(b[1], b[2], b[3]);
   Vector normal = new Vector(3);
   normal[0] = b[4];
   normal[1] = b[5];
   normal[2] = b[6];
   normal = normal.UnitVector;
   Circle3D circle = new Circle3D(radius, centre, normal);

   Point3D point = new Point3D(r);
   rtn = circle.DistanceTo(point);

   return rtn;

}
```
Figure 3

5.4.18 **Polynomial(double d)**

If the parent vector is represented by the sequence $(v_0, v_1, ..., v_{N-1})$ this function computes the expression: $\sum_{i=0}^{N-1} v_i d^i$ i.e. the result of the polynomial evaluated at d. d may also be complex.

C# example:

```
Vector poly_n = new Vector(3)
...

double d = 0.1;
double eval = poly_n.Polynomial(d);
```

5.4.19 **SetSubVector(int i_1,int i_2,Vector v)**

Replaces the elements in the parent between the indices specified with those of vector v. If the indices are out of range or $i_2 < i_1$ or v is of an incompatible dimension an error is thrown.

C# example:

```
v1.SetSubVector(0,2,v2);
```

5.4.20 **Vector Sort(int [ ] *location*)**

This routine performs a bubble sort to return the elements of the vector sorted with respect to size - with the largest element the first element in the sorted vector. Due regard is taken to the sign of the elements. The new sorted location of each original element is held in *location* (zer based)

C# example:

```
//Order the singular values so that the largest appears first along the diagonal.
Vector diag = new Vector(S.Diagonal);
int[]  loc    = new int[cols];
Vector sort = new Vector(cols);
sort          = diag.Sort(loc);
```

```
S.SetDiagonal(sort);
Matrix tempU  = new Matrix(U);
Matrix tempV  = new Matrix(V);

for (int i = 0; i < cols; i++)
{
   U.SetColumnVector(loc[i],tempU.GetColumnVector(i));
   V.SetColumnVector(loc[i],tempV.GetColumnVector(i));
}
```

5.4.21 **void Swap(int i ,int j)**

Interchanges elements i and j . If i or j are not valid array elements an error is thrown.

C# example:

```
Vector v1 = new Vector(4);
...
v1.Swap(0,3);
```

5.4.22 **Vector SubVector(int i_1,int i_2)**

This returns the vector between indices i_1 and i_2 inclusive of the parent. If the indices are out of range or $i_2 < i_1$ then an error is thrown.

C# example:

```
Vector v1 = v2.SubVector(0,2);
```

5.4.23 **Matrix Transpose()**

This method returns the vector as a 1 by n matrix.

C# example:

```
Vector v2 = new Vector(v1);
Matrix A = v2.Transpose();
```

6 Complex class

This class permits the construction and manipulation of complex numbers.

6.1 Constructors

6.1.1 **Complex()**

Constructs a complex number whose real and imaginary parts are both zero.

C# example:

```
Complex c = new Complex();
```

6.1.2 **Complex(Complex c)**

Constructs a complex number whose real and imaginary parts are the same as those of c.

C# example:

```
Complex c1 = new Complex();

c1.Imaginary = 1.5;

Complex c2 = new Complex(c1);
```

6.1.3 **Complex(double $real$,double $imag$)**

Constructs a complex number with real part $real$ and imaginary part $imag$.

C# example:

```
Complex c = new Complex(1.0,2.0);
```

6.2 Properties

6.2.1 **Complex Cos**

Returns the cosine of the complex number z. This is the expression $\frac{e^{iz}+e^{-iz}}{2}$. See 6.2.5

6.2.2 **Complex Exp**

Returns the exponential of the complex number. See 6.2.5

6.2.3 **double Real**

Reads or writes the real part.

6.2.4 **double Imaginary**

Reads or writes the imaginary part.

C# example:

```
double rl;

double im;
```

```
Complex c = new Complex(c2);
rl = c.Real;
im = c.Imaginary;
```

6.2.5 double Magnitude

Reads the magnitude of the complex number. If (x, y) represents the complex number then its magnitude is $\sqrt{x^2 + y^2}$.

C# example:

```
double rl = 1.0;
double im = 5.0;
Complex c = new Complex(rl,im);
Double mag = c.Magnitude;
Complex s = c.Sin;
Complex e = c.Exp;
Complex cs = c.Cos;
```

6.2.6 double Phase

Reads the phase (or argument) of the complex number. This is the angle subtended in the Argand diagram. Phase φ equals $\arctan 2(y, x)$. $-\pi < \varphi \leq \pi$.

C# example:

```
double rl = 1.0;
double im = 5.0;
Complex c = new Complex(rl,im);
double ph = c.Phase;
double sq = c.Sqrt;
```

6.2.7 Complex Sin

Returns the sine of the complex number z . This is the expression $\frac{e^{iz} - e^{-iz}}{2i}$. See 6.2.5.

6.2.8 Complex Sqrt

Returns the principal part of the square root of the complex number z . If $z = x + iy$,($y \neq 0$) , this is the expression $c + id$ where:

$c = \sqrt{\frac{x + \sqrt{x^2 + y^2}}{2}}$, $d = sign(y)\sqrt{\frac{-x + \sqrt{x^2 + y^2}}{2}}$. See 6.2.5.

6.3 Overloaded operators

6.3.1 + operator

If $c1$ and $c2$ are double or complex numbers this operator permits $c1+c2$ to return a complex number.

C# example:

```
Complex c = c1 + c2;
```

6.3.2 Add(Complex c)

Adds c to the parent. To be used in languages which do not permit overloading the + operator. c may also be real.

6.3.3 - operator

If $c1$ and $c2$ are double or complex numbers this operator permits $c1-c2$ to return a complex number.

C# example:

```
Complex c = c1 – c2;
```

6.3.4 Subtract(Complex c)

Subtracts c from the parent. To be used in languages which do not permit overloading the - operator. c may also be real.

6.3.5 * operator

If $c1$ and $c2$ are complex numbers and λ is a scalar real number this operator permits $c1*\lambda$, $\lambda*c2$ and $c1*c2$ to return complex numbers.

C# example:

```
double lambda;

Complex c = c1*c2 + lambda*c2;
```

6.3.6 Multiply(Complex c)

Multiplies the parent by c. To be used in languages which do not permit overloading the * operator. c may also be real.

6.3.7 operator /

If $c1$ and $c2$ are complex numbers and λ is a scalar real number this operator permits $c1/\lambda$, $\lambda/c2$ and $c1/c2$ to return a complex number. If λ or $c2$ are zero when acting as a divisor an error is thrown.

C# example:

```
double lambda;

Complex c = c1/c2 + c2/lambda;
```

6.3.8 **Divide(Complex c)**

Divides the parent by c. To be used in languages which do not permit overloading the / operator. c may also be real.

6.3.9 **operator ==**

Returns true if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise false.

6.3.10 **operator !=**

Returns false if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise returns true.

6.4 Methods

6.4.1 **Complex Conjugate()**

If (x, y) is the complex number this function returns the complex conjugate: $(x, -y)$. In this document the conjugate of c will be denoted by $\bar{c}$

C# example:

```
double lambda;

Complex c = c1.Conjugate();
```

6.4.2 **void Assign(Complex c)**

This permits assignment of one complex number to another.

C# example:

```
Complex c = new Complex();

c.Assign(c1); // performs c = c1
```

7 ComplexVector class

This class permits the construction and manipulation of vectors whose elements are complex numbers.

7.1 Constructors

7.1.1 ComplexVector()

Initialises a 3 dimensional vector all of whose elements are zero.

C# example:

```
ComplexVector v = new ComplexVector();
```

7.1.2 ComplexVector(int $\dim$)

Initialises a vector of dimension $\dim$ all of whose elements are zero.

C# example:

```
int dim = 3;

ComplexVector v = new ComplexVector(dim);
```

7.1.3 ComplexVector(ComplexVector v)

Initialises a vector to have the same elements and dimension as v.

C# example:

```
ComplexVector v1 = new ComplexVector(3);

Complex c = new Complex(3,4);

v1[1]=c;

ComplexVector v2 = new ComplexVector(v1);
```

7.2 Properties

7.2.1 int Dim

Returns the dimension of the vector - read only.

C# example:

```
int dimension;

ComplexVector v = new ComplexVector(3);

dimension = v.Dim
```

7.2.2 [i]

Gets and sets the i th element of the vector. Here i is greater than or equal to zero and less than the dimension of the vector. If access outside of this range is requested an error is thrown.

C# example:

```
Complex element1, element2;
ComplexVector v = new ComplexVector();
element1=new Complex(1.5,1.0);
v[0]=element1;
element2=v[1];
```

7.2.3 **Vector Abs**

Returns the real vector, of the same dimension as the parent, whose *i* th element is the magnitude of the *i* th element of the parent.

C# example:

see 7.2.6 .

7.2.4 **ComplexVector Cos**

Returns the complex vector whose *i* th element is the cosine of the corresponding element in the parent vector.

C# example:

See 7.2.5.

7.2.5 **ComplexVector Exp**

Returns the complex vector whose *i* ith element is the exponential of the corresponding element in the parent vector.

C# example:

```
ComplexVector v = ComplexVector();
element1=new Complex(1.5,1.0);
v[0]=element1;
ComplexVector v2 = v.Exp;
ComplexVector v3 = v.Sin;
Complexvector v4 = v.Cos;
ComplexVector v5 = v.Sqrt;
```

7.2.6 **Vector Imaginary**

Returns the real vector with the same dimension as the parent but each of whose elements is the corresponding imaginary part of the parent.

C# example:

```
ComplexVector v = new ComplexVector();
v[0]=new Complex(1.5,1.0);
Vector vi = v.Imaginary;
Vector vr = v.Real;

Vector va = v.Abs;
```

7.2.7 double Modulus

Read only property to return the modulus or 2-norm of the vector. If $v = (v_0, v_1, ..., v_{n-1})$ then the modulus is: $\|v\| = \sqrt{v_0 * \bar{v}_0 + ... + v_{n-1} * \bar{v}_{n-1}}$.

C# example:

```
If (v.Modulus < 1.0e-7)

{

   break

}
```

7.2.8 Vector Phase

Returns the real vector with the same dimension as the parent, but each of whose elements is the corresponding phase of the parent.

7.2.9 Vector Real

Returns the real vector with the same dimension as the parent but each of whose elements is the corresponding real part of the parent.

C# example:

see 7.2.6 .

7.2.10 ComplexVector Sin

Returns the complex vector whose i th element is the sine of the corresponding element in the parent vector.

C# example:

See 7.2.5.

7.2.11 ComplexVector Sqrt

Returns the complex vector whose ith element is the sqrt of the corresponding element in the parent vector.

C# example:

See 7.2.5

7.3 Overloaded operators

7.3.1 + operator

If v_1 and v_2 are vectors this permits the expression $v_1 + v_2$ to return a vector. If v_1 and v_2 are not of the same dimension an error is thrown.

C# example:

```
ComplexVector v = v1 + v2;
```

7.3.2 **Add(ComplexVector v)**

Adds vector v to the parent. To be used in languages which can not override the + operator. v may be a real or complex vector.

7.3.3 **- operator**

If v_1 and v_2 are vectors this permits the expression $v_1 - v_2$ to return a vector. If v_1 and v_2 are not of the same dimension an error is thrown.

C# example:

```
ComplexVector v = v1 - v2;
```

7.3.4 **Subtract(ComplexVector v)**

Subtracts vector v from the parent. v may be real or complex. To be used in languages which can not override the - operator.

7.3.5 *** operator**

If v is a vector and λ a scalar this permits both the expression $\lambda * v$ and $v * \lambda$ to return a vector. λ may be real or complex. The multiplication of the vector by the scalar is component-wise.

If $v1$ and $v2$ are vectors this operator permits the component-wise multiplication of $v1$ and $v2$. If $v2$ is zero an error is thrown.

C# example:

```
double lambda1 = 1.4;

Complex lambda2 = new Complex(0.0,0.6);

ComplexVector v = (lambda1 * v1) * lambda2;

ComplexVector w = v*v;
```

7.3.6 **Multiply(double d)**

Multiplies the parent by d. d may be real or complex. Componentwise multiplication of two vectors is also possible. To be used in languages which can not override the * operator.

7.3.7 **operator /**

If v is a vector and λ a scalar this permits the expression $v*/\lambda$ to return a vector. λ may be real or complex. The division of the vector by the scalar is component-wise. If λ is zero an error is thrown.

If $v1$ and $v2$ are vectors this operator permits the component-wise division of $v1$ and $v2$. If a component of $v2$ is zero an error is thrown.

C# example:

```
double lambda1 = 1.4;
Complex lambda2 = new Complex(0.0,0.6);
ComplexVector v = (lambda1 / v1) / lambda2;
ComplexVector w = v/v;  //should be 1 in each component
```

7.3.8 Divide(double d)

Divides the parent by d. d may be real or complex. Componentwise division of two vectors is also possible. To be used in languages which do not permit / to be overridden.

7.3.9 operator ==

Returns true if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise returns false.

7.3.10 Operator !=

Returns false if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise returns true.

7.4 Methods

7.4.1 void Assign(ComplexVector v)

If v_1 and v_2 are two vectors of the same dimension this method is the C# way of performing $v_1 = v_2$. If v_1 and v_2 are not of the same dimension an error is thrown.

C# example:

```
v1.Assign(v2);
```

7.4.2 ComplexVector Conjugate()

Returns a vector whose elements are the complex conjugate of those of the parent.

7.4.3 ComplexMatrix ConjugateTranspose()

For an n dimensional parent vector this method returns the $1 \times n$ matrix whose elements are the complex conjugate of those of the parent.

7.4.4 ComplexVector Convolution(ComplexVector v)

Returns the vector which is the convolution of the parent with v. See 5.4.4.

7.4.5 Complex Dot(ComplexVector v)

Returns the dot product with another vector v. If v is not of the same dimension as the parent an error is thrown.

C# example:

```
Complex c = w.dot(w.Conjugate())
```

```
double d = c.Real - w.Modulus*w.Modulus //should be zero
```

7.4.6 void Edit()

Displays a dialogue in which it is possible to view the vector and edit its contents. To change a cell: select it, type in the new value and click elsewhere so the focus of the cell is lost. Press the update button to commit the changes to the vector. Press the cancel button to quit the form without committing any changes.

C# example:

```
ComplexVector v = new ComplexVector(4);
v.Edit();
```

7.4.7 ComplexVector Fft()

This function uses a Cooley-Tukey method to return the discrete Fourier transform of the parent vector. If x_n is the series representing the parent vector where $0 \le n < N$, N being the dimension of the vector, then the vector returned is X_k $(0 \le k < N)$ where:

$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}nk}$. Here i represents the square root of minus 1.

C# example:

```
Matrix mat = Matrix.CsvRead(str);
Vector r = mat.GetColumnVector(1);
ComplexVector R = new ComplexVector(r.Dim);
R = r.Fft();
```

7.4.8 ComplexVector Ifft()

This function computes the inverse discrete Fourier transform of the parent vector. Using the notation of 7.4.7,

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N}nk}$$

C# example:

```
Matrix mat = Matrix.CsvRead(str);
Vector t = mat.GetColumnVector(0);
Vector r = mat.GetColumnVector(1);

int N = t.dim;
double dt = Math.Abs(t[1] - t[0]) / 1.0E12;
Vector f = new Vector(N);
for (int i = 0; i < N; i++)
{
  f[i] = (1.0 / 1.0E12) * (1.0 / dt) * i / N;
}

ComplexVector R = new ComplexVector(N);
```

```
R = r.Complex.Fft();
ComplexVector ir = R.Ifft();

Graph gr = new Graph();
gr.plot(t, r);
gr.plot(t, ir.Real);
gr.add_title("signal against time"); //graphs should be the same
```

7.4.9 Boolean IsZero()

This function returns true of the magnitude of every element of the parent is zero (less than Matrix.ZeroTolerance). Otherwise the function returns false.

C# example:

```
If (v.IsZero())
{
   Return converged;
}
```

7.4.10 ComplexMatrix ViewAsMatrix()

An n dimensional complex vector is to be viewed as an $n\times 1$ complex matrix.

7.4.11 ComplexVector NonlinearSolve(ComplexNonlinearFit f, ComplexVector x,ComplexVector y, int ref $conv$)

Let f be a function mapping $C^m \times C^n$ to C^p, where m, n and p are positive integers. Here C is used to denote the field of complex numbers. Let f_i be the projection of f onto the i th component of C^p ($1 \le i \le p$). Let $x \in C^m$ and $y \in C^p$ Then this function finds $\beta \in C^n$ such that $S(\beta) = \sum_{i=1}^{p} \| y_i - f_i(x,\beta) \|^2$ is a minimum.

The technique employed is an iterative one involving successive approximations β_k to β.

The algorithm terminates successfully ($conv$ set to 1) if $\|\beta_k - \beta_{k-1}\|$<Matrix.ZeroTolerance, for some $k <$ Matrix.MaxNumberOfLoops. If the maximum number of iterations, as given by Matrix.MaxNumberOfLoops, is exceeded the algorithm terminates with $conv$ set to 0.

For success of the algorithm employed it is necessary that the parent vector starts fairly close to the eventual solution β. You should carefully analyse the situation you are trying to solve to determine of 7.4.11 or 7.4.12 is the better algorithm to employ. 7.4.12 is usually much quicker.

C# example:

The following is an example in which $m = p = 100$ and $n = 3$.

```
int N = 100; // number of observations
ComplexVector x = new ComplexVector(N);
for (int i = 0; i < N; i++)
```

```
{
   x[i] = new Complex(-Math.Log(i+1),0.0);
}

ComplexVector v = new ComplexVector(3);
v[0] = new Complex(2.0,0.6);
v[1] = new Complex(2.0,4.0);
v[2] = new Complex(-0.5,0.4);
ComplexVector y = new ComplexVector(N);
for (int i = 0; i < N; i++)
{
    y[i] = v[0]+v[1]*(v[2]*x[i]).exp + x[i];
}


ComplexNonlinearFit fn = new ComplexNonlinearFit(LMexample);

Matrix.ZeroTolerance = 1.0E-8;
int conv = 0;
ComplexVector pInit = new ComplexVector(3);
pInit[0] = new Complex(2.0, 0.0);
pInit[1] = new Complex(3.0, 4.0);
pInit[2] = new Complex(-0.4, 0.3);
ComplexVector pFit = new ComplexVector(3);
pFit.Assign( pInit.NonlinearSolve(fn, x, y, ref conv) );


ComplexVector ysim = new ComplexVector(N);
for (int i = 0; i < N; i++)
{
   ysim[i] = pFit[0] + pFit[1] * (pFit[2] * x[i]).exp + x[i];
}

Graph gr = new Graph();
gr.plot(y.Real);
gr.plot(ysim.Real);
gr.add_legend("experimental");
gr.add_legend("simulated");

...

private ComplexVector LMexample(ComplexVector x, ComplexVector beta)
{
    ComplexVector rtn = new ComplexVector(100);
    for (int i = 0 ; i < 100; i++)
    {
       rtn[i] = beta[0] + beta[1] * (beta[2] * x[i]).exp + x[i];
    }

    return rtn;
}
```

7.4.12 **Complex NonlinearSolve2(ComplexNonlinearFit2 f , ComplexVector [ ] x ,ComplexVector y , int ref $conv$)**

Let f be a function mapping $C^m \times C^n$ to C , where m, and n are positive integers. Here C is used to denote the field of complex numbers. Let $x_i \in C$ and $y_i \in C$, $1 \le i \le p$, for some integer p . Then this function finds $\beta \in C^n$ such that $S(\beta) = \sum_{i=1}^{p} \| y_i - f(x_i, \beta) \|^2$ is a minimum.

The technique employed is an iterative one involving successive approximations β_k to β . The first guess for β is the value of the parent vector. The algorithm terminates successfully ($conv$ set to 1) if $\left\| \beta_k - \beta_{k-1} \right\| <$ Matrix.ZeroTolerance, for some $k <$ Matrix.MaxNumberOfLoops. If the maximum number of iterations, as given by Matrix.MaxNumberOfLoops, is exceeded, the algorithm terminates with $conv$ set to 0.

For success of the algorithm employed it is necessary that the parent vector starts fairly close to the eventual solution β .

7.4.13 **Void SetSubVector(int $i1$,int $i2$,ComplexVector v)**

Replaces the elements in the parent between the indices specified with those of vector v. If the indices are out of range or $i_2 < i_1$ or v is of an incompatible dimension an error is thrown.

C# example:

```
v1.SetSubVector(0,2,v2);
```

7.4.14 **ComplexVector SubVector(int $i1$,int $i2$)**

This returns the vector between indices i_1 and i_2 inclusive of the parent. If the indices are out of range or $i_2 < i_1$ then an error is thrown.

C# example:

```
ComplexVector v1 = v2.Subvector(0,2);
```

7.4.15 **ComplexMatrix Transpose()**

This is an operation that permits an n dimensional vector to be viewed as a $1 \times n$ matrix.

8 Matrix class

8.1 Constructors

8.1.1 **Matrix()**

The default constructor initialises a 3-by-3 matrix each element of which is zero. Rows are represented by the first index and columns by the second index. Both indices are zero based.

C# example:

```
Matrix A = new Matrix();
```

8.1.2 **Matrix(int m,int n)**

This constructor initialises an m by n matrix each element of which is zero.

C# example:

```
Matrix A = new Matrix(3,4);
```

8.1.3 **Matrix(Matrix A)**

This constructor initialises a matrix with the same number of rows and columns as A. The elements of the matrix are the same as those of A.

C# example:

```
Matrix A = new Matrix(2,3);

Matrix B = new Matrix(A);
```

8.2 Properties

8.2.1 **ComplexMatrix ViewAsComplex**

Views the parent matrix as a complex matrix.

8.2.2 **Diagonal**

Returns the diagonal of the matrix. The matrix does not have to be square.

C# example:

```
Vector diag = new Vector(S.Diagonal);
```

8.2.3 **Max**

Returns the maximum of all the elements of the matrix.

8.2.4 **Min**

Returns the minimum of all the elements of the matrix.

8.2.5 Modulos

Returns the quantity: $\sqrt{\sum_{i,j} A_{i,j}^2}$ where the summation is over all elements of the parent matrix A.

8.2.6 int Nrows

A read only property giving the number of rows of the matrix.

C# example:

```
int m;
m = A.Nrows;
```

8.2.7 Ncols

A read only property giving the number of columns of the matrix.

C# example:

```
int n;
n = A.Ncols;
```

8.2.8 [i, j]

Gets and sets the $[i, j]$ th element of the matrix.

C# example:

```
Double a,b;
a = 1.5;
A[2,3] = a;
b = A[1,2];
```

8.2.9 MaxNumberOfLoops

Permits reading and writing the maximum number of iterations that may be performed in the SVD calculation. The default is 10000.

8.2.10 ZeroTolerance

Permits reading and writing the the value below which a number is considered to be zero for the purposes of numerical computations. The value is a positive number greater than zero, otherwise an error is thrown. Decreasing this value will mean that the SVD algorithm takes more iterations to converge. The default is 1.0E-14.

8.3 Static methods

8.3.1 Matrix CsvRead(String s, int $offset$)

Reads a comma -seperated-variables file. The first $offset$ lines of the file are omitted, as this is presumed to be the header and the format of this does not matter. The remaining lines are parsed and read using a ‘,’ as a separator. If there is no data or the format of the data is not consistent, not numeric or missing, an error is thrown.

C# example:

```
String str = "C:\\downloads\\data.csv";

Matrix A = Matrix.CsvRead(str,1); //first line is the header
```

8.3.2 Matrix CsvRead2(String s, int *offset*, char *options*)

Reads a comma -seperated-variables file. The first $offset$ lines of the file are omitted, as this is presumed to be the header and the format of this does not matter. The remaining lines are parsed and read using a ',' as a separator. If there is no data or the format of the data is not consistent, an error is thrown. *Options* allows behaviour in the event of missing data in a field, or non-numeric data in a field, to be specified:

- 1 - replace missing data with 0.0.
- 2 - replace non-numeric data with 0.0.
- 3 - replace both missing and non-numeric data with 0.0.

C# example:

```
String str = "C:\\downloads\\data.csv";

Matrix A = Matrix.CsvRead(str,1,2); //first line is the header,
//replace non-numeric data with 0.0.
```

8.3.3 Matrix excel_read(String s) - no longer available

Reads the first sheet of a Microsoft Excel document. If excel is not loaded on the client machine, or the sheet is empty, or there are missing elements in the array of cells, an error is thrown.

C# example:

```
String str = "C:\\downloads\\data.xlsx";

Matrix A = Matrix.excel_read(str);
```

8.3.4 Matrix Eye(Int m,Int n)

A static method which return a matrix such that:

$Eye(i, j) = 0$, for $i \neq j$

$Eye(i,i) = 1$

8.3.5 CalculateGivensParameters(double a,double b,double ref c,double ref s)

Given scalars a and b this function computes c and s (where c and s are related by $c = \cos(\theta)$, $s = \sin(\theta)$ for some θ) such that:

$\begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$, for some scalar r .

8.3.6 **Matrix Identity(int m)**

A static method which returns the m-by-m identity matrix.

8.4 Overloaded operators

8.4.1 **+ operator**

If A_1 and A_2 are matrices this operator permits $A_1 + A_2$ to return a matrix. If A_1 and A_2 are not of the same dimension an error is thrown.

C# example:

```
Matrix A = A1 + A1;
```

8.4.2 **Add(Matrix A)**

Adds matrix A to the parent. A may be real or complex. To be used in languages which do not permit + to be overridden.

8.4.3 **– operator**

If A_1 and A_2 are matrices this operator permits $A_1 - A_2$ to return a matrix. If A_1 and A_2 are not of the same dimension an error is thrown.

C# example:

```
Matrix A = A1 – A2;
```

8.4.4 **Subtract(Matrix A)**

Subtracts A from the parent. A may be real or complex. To be used in languages which do not permit - to be overridden.

8.4.5 *** operator**

- If A is a matrix and λ is a scalar this operator permits both $\lambda * A$ and $A * \lambda$ to return a matrix. The multiplication of the matrix by the scalar is component-wise.
- If A_1 and A_2 are two matrices of compatible dimensions for matrix multiplication this operator permits $A_1 * A_2$ to return a matrix. If the dimensions are not compatible an error is thrown.
- If A is a matrix and v a vector this operator permits $A * v$ to return a vector. If A and v are not of compatible dimensions an error is thrown.

C# example:

```
Matrix A = Matrix.Identity();

Matrix B = (lambda1 * A) * lambda2;

Matrix C = A*B;

Vector v = new Vector(3);

V[0]=1.0; v[1] = 2.0;
```

```
Vector w = A*v;
```

8.4.6 Multiply(double d)

Multiplies the parent by d. d may be real or complex. To be used in languages which do not permit * to be overridden.

8.4.7 Multiply(Matrix A)

Permits the parent to be multiplied by A. A may be real or complex. To be used in languages which do not permit * to be overridden.

8.4.8 Multiply(Vector v)

Permits matrix vector multiplication of the parent. v may be real or complex. To be used in languages which do not permit * to be overridden.

8.4.9 operator /

If A is a matrix and λ a scalar this operation returns A/λ in a component-wise manner. If λ is zero an error is thrown.

C# example:

```
Matrix A = (B/2.5);
```

8.4.10 Divide(double d)

Divides the parent by d. d can be real or complex. To be used in languages which do not permit / to be overridden.

8.4.11 operator ==

Returns true if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise returns false.

8.4.12 operator !=

Returns false if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise returns true.

8.5 Methods

8.5.1 Assign

If A_1 and A_2 are two matrices of the same dimensions this is the C# way of performing the assignment $A_1 = A_2$. If A_1 and A_2 are not of the same dimensions an error is thrown.

C# example:

```
A1.Assign(A2);
```

8.5.2 Vector BezierCurve(double t)

8.5.3 Vector BezierCurve(Vector w,double t)

Given $n+1$ control points $b_0, b_1, ..., b_n$ represented by each row of the parent matrix the first function calculates the integral Bezier curve of degree n , $B(t)$, at the value of the parameter t, defined by:

$$B(t) = \sum_{i=0}^{n} b_i B_{i,n}(t)$$

where

$$B_{i,n}(t) = \frac{n!}{(n-i)!i!}(1-t)^{n-i} t^i \quad \text{if } 0 \le i \le n,\ 0 \text{ otherwise. Here } 0 \le t \le 1.0.$$

This function is used to plot integral Bezier curves given their control points.

If there are less than 2 control points or there are less than 3 columns in the matrix an error is thrown.

If w contains the corresponding positive scalar weights $w_0, ..., w_n$ the second function calculates:

$$B(t) = \frac{\sum_{i=0}^{n} w_i b_i B_{i,n}(t)}{\sum_{i=0}^{n} w_i B_{i,n}(t)},$$

the rational Bezier curve of degree n. A wider range of curves including all the conics can be represented by rational Bezier curves. If all weights are zero an error is thrown.

C# example:

The following example plots the rational Bezier curve representing a quadrant of a circle.

```
int N = 3;
// Initialise the N control points
Vector [] b = new Vector[N];
for (int i = 0; i < N; i++)
{
   b[i] = new Vector(3);
}
b[0][0]=1.0;  b[0][1]=0.0; b[0][2]=0.0;
b[1][0]=1.0;  b[1][1]=1.0; b[1][2]=0.0;
b[2][0]=0.0;  b[2][1]=1.0; b[2][2] = 0.0;

//Plot the graphs of the control points
Graph g = new Graph();
for (int i = 0; i < N-1; i++)
{
    Vector u = new Vector(100);
    Vector v = new Vector(100);
    for (int t = 0; t < 100; t++)
    {
      u[t] = b[i][0] + (t/99.0)*(b[i+1][0]-b[i][0]);
      v[t] = b[i][1] + (t / 99.0) * (b[i + 1][1] - b[i][1]);
    }
    g.Plot(u,v);
}
```

```
//Plot the Bezier curves
Matrix M = new Matrix(N,3);
Vector weights = new Vector(N);
weights[0] = 1.0; weights[1] = 1.0; weights[2] = 2.0;
for (int i = 0; i < N; i++)
{
     M.SetRowVector(i,b[i]);
}
Vector u1 = new Vector(100);
Vector v1 = new Vector(100);
for (int t = 0; t < 100; t++)
{
    Vector w = M.BezierCurve(weights,t/99.0);
    u1[t] = w[0];
    v1[t] = w[1];
}
g.Plot(u1,v1); //Should be a quadrant of a circle.
```

8.5.4 Vector BezierCurve(double t,Matrix L,Matrix R)

8.5.5 Vector BezierCurve(double t,Vector w,Vector wL,Matrix L,Vector wR,Matrix R)

The first of these functions stores in each of matrix L and R, $n+1$ control points such that a reprameterization between 0.0 and 1.0 using these new control points will trace the original curve from b_0 to $B(t)$ in the case of L and from $B(t)$ to b_n in the case of R.

The second function stores, in addition the corresponding weights to retrace both of the two curves.

If the various matrices and vectors are not of compatible dimensions an error is thrown.

C# example:

```
...

double alpha = 0.3;
Matrix left = new Matrix(N,3);
Matrix right = new Matrix(N,3);
Vector weightsLeft = new Vector(N);
Vector weightsRight = new Vector(N);
Vector w2 = M.BezierCurve(alpha,weights,weightsLeft,left,weightsRight,right);
Vector u2 = new Vector(1000);
Vector v2 = new Vector(1000);
Vector u3 = new Vector(1000);
Vector v3 = new Vector(1000);
for (int t = 0; t < 1000; t++)
{
     Vector w3 = left.BezierCurve(weightsLeft, t / 999.0);
     u2[t] = w3[0];
     v2[t] = w3[1];
     Vector w4 = right.BezierCurve(weightsRight, t / 999.0);
     u3[t] = w4[0];
     v3[t] = w4[1];
}

g.Plot(u2, v2); //The curve from start to B(0.3).
g.Plot(u3, v3); //The curve from B(0.3) to end.
```

8.5.6 **Bidiagonalization(Matrix U ,Matrix V)**

If A is the m-by-n parent matrix, ($m \geq n$), this method overwrites A with $B = U^T AV$, where B is upper bi-diagonal, U is an m-by - m orthogonal matrix, and V is an n- by - n orthogonal matrix. A can be recovered by calculating UBV^T. Rows and columns are zeroed using a series of Givens transformations.

C# example:

```
Matrix A = new Matrix(4,3);
...
Matrix U = Matrix.Identity(4);
Matrix V = Matrix.Identity(3);
A.Bidiagonalization(U,V);
```

8.5.7 **BidiagonalizationLight(Matrix U ,Matrix V)**

Let A be the original parent m by n matrix.

If $m \geq n$:

The algorithm computes an m by n matrix U with orthogonal columns, and an n by n orthogonal matrix V. The n by n left-most upper portion of A is overwritten by an n by n upper bidiagonal matrix B. The lower portion is zeroed.

If $m < n$:

The algorithm computes an m by m orthogonal matrix U, and an n by m matrix V with orthogonal columns. The m by m left-most upper portion of A is overwritten by an m by m lower bidiagonal matrix B. The right portion is zeroed.

If the dimensions of the input matrices do not correspond to these dimensions an error is thrown.

In all cases A is given by $A = UBV^T$.

C# example:

```
Matrix A = new Matrix(4,3);
...
Matrix U = new Matrix(4,3);
Matrix V = new Matrix.Identity(3);
A.BidiagonalizationLight(U,V);
```

8.5.8 **ColumnHouseholder(Vector v)**

Given the m-by-n matrix A and a non-zero n-vector v with $v[0]=1.0$,this method overwrites A with AP , where $P = I - 2\frac{vv^T}{v^T v}$. The calculations are performed in a computationally efficient manner. If v is not an n-vector an error is thrown.

8.5.9 **double Determinant(ref int $converged$)**

Calculates the determinant of a square matrix. If the matrix is non-square an error is thrown. If the underlying SVD routine has converged, $converged$ is set to 1. Otherwise it is set to 0.

C# example:

```
int conv = 1;
double det = A.Determinant(ref conv);
```

8.5.10 **double Dot(Matrix $\boldsymbol{M}$)**

If A is the parent matrix and both matrices are of dimension m by n this method returns the value: $\sum_{i=1}^{m}\sum_{j=1}^{n} A_{i,j}M_{i,j}$. That is a pointwise multiplication of the two matrices.

8.5.11 **void Edit()**

This method displays a dialogue which permits the elements of the matrix to be changed by the user. To change a cell: select it, type in the new value and click elsewhere so the focus of the cell is lost. Press the update button to commit the changes to the matrix. Press the cancel button to quit the form without committing any changes. Rows or columns of the matrix are plotted individually.

C# example:

```
Matrix A = new Matrix(4,4);
A.Edit();
```

8.5.12 **int Eigenvalues(ComplexVector E)**

If A is the n by n parent matrix this routine calculates the n complex numbers, λ, which together with n vectors, v ,satisfy the equation $Av = \lambda v$ .The eigenvalues λ are returned as a complex vector in E. The eigenvalues are sorted in terms of magnitude with the largest first. If the underlying numerical routine used has not converged zero is returned, otherwise one is returned. The routine used is a QR algorithm employing a Francis QR step. This should converge in most cases. The convergence criteria is determind by *Matrix.ZeroTolerance* and the maximum number of steps in the iteration by *Matrix.MaxNumberOfLoops.* It may be possible to allow the routine to converge by increasing one or both of these parameters.

If the matrix is not square an error is thrown.

C# example:

```
ComplexVector E = new ComplexVector(7);
int Eig_return1 = 1;
Eig_return1 = A.Eigenvalues(E);
```

8.5.13 **int Eigenvectors(ComplexVector $Eval$,ComplexMatrix $Evec$)**

8.5.14 **int OrthogonalEigenvectors(ComplexVector $Eval$,ComplexMatrix $Evec$)**

If A is the n by n parent matrix these routines calculates the n Eigenvalues, λ, which together with the n Eigenvectors v, satisy the equation $Av = \lambda v$. The Eigenvalues are returned as a complex vector in $Eval$. The corresponding Eigenvectors are returned as columns of the complex matrix $Evec$. The Eigenvectors and their corresponding Eigenvalues are sorted in order of magnitude, with the largest first. If the underlying numerical routine has not converged, or there is some kind of problem with the result, zero is returned, otherwise one is returned.

The routine 'Eigenvectors' uses a QR algorithm employing a Francis QR step. This should converge in most cases. Convergence will be problematic in the case of an orthogonal matrix. In this case use the routine 'OrthogonalEigenvectors'. The convergence criteria is determind by *Matrix.ZeroTolerance* and the maximum number of steps in the iteration by *Matrix.MaxNumberOfLoops.* It may be possible to allow the routine to converge by increasing one or both of these parameters.

If the matrix is not square an error is thrown.

C# example:

```
Eig_return = A.Eigenvectors(E,P);
if (Eig_return == 1)
{
    for (i = 0; i < n; i++)
    {
        ComplexVector diff = new ComplexVector(n);
        diff = E[i] * P.GetColumnVector(i) - A *
P.GetColumnVector(i);
        if (diff.Modulus > (((double)n)*1.0E-8))
        {
            MessageBox.Show("Complex Eigenvector routine has
failed");
            return;
        }
    }
}
```

8.5.15 **double Eigenvector(Vector $Eval$,Vector $initial$, ref int $conv$)**

Use this routine when it is known that the matrix has a single dominant eigenvalue i.e. $|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots \geq |\lambda_n|$. In this case the routine will converge to λ_1 (the return value) and the associated dominant eigenvector ($Eval$). Note, that in this case λ_1 must be real. The initial guess to the dominant eigenvector must have a component in the direction of the dominant eigenvector. The routine employs the power method algorithm. This routine is much less computational intensive than the ones for returning all the eigenvector/values, and should be used when some information is known about the matrix.

8.5.16 **void ElementaryRowOperation(int i ,int j)**

Swaps rows i and j. If i or j are not valid row indices an error is thrown.

C# example:

```
Matrix U = new Matrix (6,5);
...
```

```
U.ElementaryRowOperation(k, imax);
```

8.5.17 **void GaussPreMultiplication(int k,Vector g)**

Let A be the m by n parent matrix. And let g be a Gauss vector of dimension $m-k$. Let $L(k)$ be the lower triangular m by m matrix of the form:

$$\begin{pmatrix} 1 & 0 & 0 \\ \dots & & \\ & 1 & \\ & g[1] & \\ & \dots & \\ 0 & g[n-1] & 1 \end{pmatrix}$$, where the Gauss vector occurs in the kth column.

Then this function sets A to $L(k)*A$, in a computationally efficient manner.

C# example:

```
//Get Gauss vector
Vector gauss = matrixU.GetColumnVector(k).SubVector(k,Nrows -
1).GaussVector();

//Perform left Gauss operation to update U
matrixU.GaussPreMultiplication(k,gauss);
```

8.5.18 **int GaussSeidel(Vector b, Vector xo, Vector x)**

If A is the parent matrix this routine uses the Gauss-Seidel iterative method to solve the linear system $Ax = b$, starting with the initial estimate of xo. If the system converges successfully 1 is returned. If the system fails to converge after *max_no_loops* iterations 0 is returned. If x_k is the estimate at the kth step the algorithm is deemed to have converged when $\left\| x_k - x_{k-1} \right\| < ZeroTolerance$. If the dimensions of A, b, x or xo are incompatible or the matrix is not square an error is thrown. If the algorithm encounters a pivot element ($A_{i,i}$) which is zero an error is thrown.

C# example:

See 8.5.33.

8.5.19 **Vector GetColumnVector(int i)**

Return the ith column of the matrix as a vector. If i is less than zero or greater or equal to the number of columns then an error is thrown.

8.5.20 **Vector GetRowVector(int i)**

Returns the ith row of the matrix as a vector. If i is less than zero or greater or equal to the number of rows then an error is thrown.

8.5.21 **GivensPostMultiplication(int i ,int k ,double c ,double s)**

Let $G(i,k,c,s)$ be the Givens matrix, where c and s are calculated according to 8.3.5. This is the identity except that

$$G(i,k,c,s)(i,i) = c$$
$$G(i,k,c,s)(k,k) = c$$
$$G(i,k,c,s)(i,k) = s$$
$$G(i,k,c,s)(k,i) = -s$$

The Givens post multiplication of a matrix A replaces A with $AG(i,k,c,s)$.

8.5.22 **Givens_pre_multiplication(int i ,int k ,double c ,double s)**

Let $G(i,k,c,s)$ be the Givens matrix, where c and s are calculated according to 8.3.5. This is the identity except that

$$G(i,k,c,s)(i,i) = c$$
$$G(i,k,c,s)(k,k) = c$$
$$G(i,k,c,s)(i,k) = s$$
$$G(i,k,c,s)(k,i) = -s$$

The Givens pre multiplication of a matrix A replaces A with $G(i,k,c,s)^T A$.

8.5.23 **Void HessenbergReduction()**

The parent is operated upon to reduce it to an upper Hessenberg form i.e. zeros on all elements below the sub-diagonal. The reduction is done using a series of Givens transformations. If the matrix is not square an error is thrown.

8.5.24 **Void HessenbergReduction(Matrix Q)**

This method operates on the parent matrix, A, to reduce it to an upper Hessenberg form H. The reduction is done using a series of Givens transformations. An orthogonal matrix, Q, is constructed such that $Q^T AQ = H$. If the matrix is not square an error is thrown.

C# example:

```
Matrix A = new Matrix(7,7);

...

Matrix Q = Matrix.Identity(4);

A.HessenbergReduction(Q);
```

8.5.25 **HouseholderBidiagonalization(Matrix U ,Matrix V)**

If A is the m-by-n matrix, ($m \geq n$), this method overwrites A with $B = U^T AV$, where B is upper bi-diagonal matrix, U is an m-by - m orthogonal matrix, and V is an n- by - n orthogonal matrix. A can be recovered by calculating UBV^T. Rows and columns are zeroed using a series of Householder transformations.

C# example:

```
Matrix A = new Matrix(4,3);

...
```

```
Matrix U = Matrix.Identity(4);
Matrix V = Matrix.Identity(3);
A.HouseholderBidiagonalization(U,V);
```

8.5.26 **HouseholderQRDecmposition(Matrix Q ,Matrix R)**

Given the m by n matrix A, this method computes an m by m orthogonal matrix Q and an m by n upper triangular matrix R such that $A = QR$. Columns are zeroed using Householder transformations.

C# example:

```
Matrix A = new Matrix(3,3);
A[0,0]=1.0;A[0,1]=2.0;A[0,2]=3.0;
A[1,0]=2.0;A[1,1]=1.0;A[1,2]=5.0;
A[2,0]=6.0;A[2,1]=5.0;A[2,2]=1.0;
Matrix U = Matrix.Identity(3);
Matrix V = Matrix.Identity(3);
A.HouseholderQRDecomposition(Q,R);
Matrix B = A - Q*R; //should be the zero matrix
```

8.5.27 **void Image()**

Let the decomposition of the parent m by n matrix, $A = USV^T$, into principal (rank 1) components be $\sum_{i=1}^{p} u_i \sigma_i v_i^T$, where $p = \min(m, n)$. Several plot types are provided:

- The ‘Contour’, ‘FilledContour’ and ‘XYColorPlot’ options permit the user to view, both the data and the contour plot of $\sum_{i=s}^{e} u_i \sigma_i v_i^T$, where $1 \le s \le e \le p$. The colouring is from smallest (blue) through yellow to highest (red). The colours for any of the plots are relative to the spectrum for the whole matrix A, with yellow representing the colour of the contour of the current plotted component which is nearest to the mean of the values of A. It is possible to vary the ranges of both axes in order to zoom in on an area.
- The ‘SingularValues’ option permits the singular values to be plotted. Again the range and the singular values displayed can be varied by the user. In this case the data displayed, to the left, is that of the matrix S.
- The ‘SingularVectors’ option permits the singular vectors to be plotted. Again the range and the singular vectors displayed can be varied by the user. At most 10 legends are displayed to avoid cluttering the graph. In this case the data displayed, to the left, is that of the matrix V.

It is possible to print the current view, save it in various file formats, and to write the respective data displayed to a csv file.

C# example:

```
A.Image();
```

A typical filled contour plot (the ‘FilledContour’ option) is shown in Figure 4.

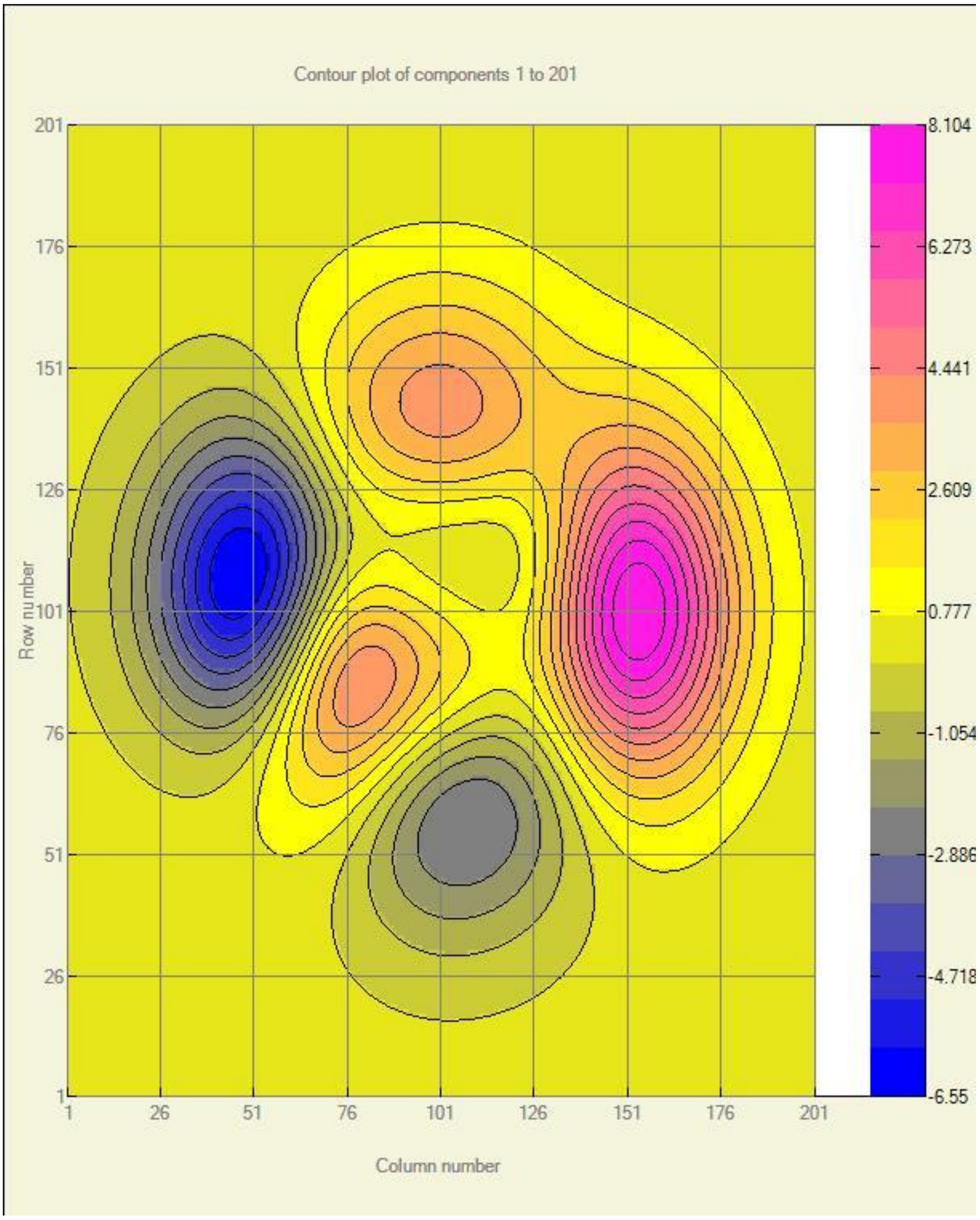


Figure 4 - Typical contour plot

8.5.28 **int Inverse(Matrix *inv*)**

Calculates (in *inv*) the inverse of a square matrix.

The return value of the function indicates the following:

Return value	Meaning
0	Successful calculation of the inverse.
1	The matrix is non-square - an error is thrown.
2	The numerical method would not converge on a solution
3	The matrix is singular (i.e. an inverse doesn't exist.

C# example:

```
int inverse calculated = 0;
inverse_calculated = A.Inverse(inv);
```

8.5.29 **Boolean IsBidiagonal()**

Returns TRUE if the matrix is upper bidiagonal, to within the tolerance given by Matrix.ZeroTolerance. Otherwise the function returns FALSE.

8.5.30 **Boolean IsHessenberg()**

Returns TRUE if the matrix is upper Hessenberg, to within the tolerance given by Matrix.ZeroTolerance. Otherwise the function returns FALSE.

C# example

```
if (!S.IsHessenberg())
{
    S.HessenbergReduction(Q);
}
```

8.5.31 **Boolean IsOrthogonal()**

Returns TRUE if the square matrix is orthogonal, to within the tolerance given by Matrix.zero_tolerance. Otherwise the function returns FALSE.

C# example

```
if ( (M.ncols !=M.Nrows) || (M.Ncols != Dim) || (M.IsOrthogonal() == false) )
{
  throw new System.InvalidOperationException("Invalid coordinate transformation
matrix");
}
```

8.5.32 **Boolean IsZero()**

Return true if all entries in the matrix are zero to within the tolerance given by Matrix.zero_tolerance. Otherwise the function returns false.

8.5.33 **int Jacobi(Vector b , Vector xo, Vector x)**

If A is the parent matrix this routine uses the Jacobi iterative method to solve the linear system $Ax = b$, starting with the initial estimate of xo. If the system converges successfully 1 is returned. If the system fails to converge after *MaxNumberOfLoops* iterations 0 is returned. If x_k is the estimate at the kth step the algorithm is deemed to have converged when $\|x_k - x_{k-1}\| < ZeroTolerance$. If the dimensions of A, b, x or xo are incompatible or the matrix is not square an error is thrown. If the algorithm encounters a pivot element ($A_{i,i}$) which is zero an error is thrown.

C# example:

```
Matrix A = new Matrix(4,4);;
Vector b = new Vector(4);
Vector x = new Vector(4);
Vector xo = new Vector(4);
double w = 1.25;
int conv;

A[0,0] = 10.0; A[0,1] = -1.0; A[0,2] = 2.0; A[0,3] = 0.0;
A[1, 0] = -1.0; A[1, 1] = 11.0; A[1, 2] = -1.0; A[1, 3] = 3.0;
A[2, 0] = 2.0; A[2, 1] = -1.0; A[2, 2] = 10.0; A[2, 3] = -1.0;
A[3, 0] = 0.0; A[3, 1] = 3.0; A[3, 2] = -1.0; A[3, 3] = 8.0;

b[0] = 6.0; b[1] = 25.0; b[2] = -11.0; b[3] = 15.0;

Matrix.ZeroTolerance = 1.0E-8;
Matrix.MaxNumberOfLoops = 100;

conv = A.Jacobi(b,xo,x);        //Jacobi method
conv = A.GaussSeidel(b,xo,x); //Gauss-Seidel method
conv = A.SymmetricOverRelaxation(b,xo,w,x);         //SOR method

//x should be [1,2,-1,1]
```

8.5.34 **void LUDecomposition(Matrix L ,Matrix U ,Matrix P)**

Given the m by n parent matrix A , this decomposition calculates an upper diagonal m by n matrix U , a lower diagonal m by m matrix L with ones on the main diagonal, and an m by m permutation matrix P such that $PA = LU$. The method is successful regardless of the rank deficiency, or not, of A .

C# example:

```
Matrix A = new Matrix(4,4);;
Matrix P = Matrix.Identity(4);
Matrix L = Matrix.Identity(4);
Matrix U = new Matrix(4,4);

A[0,0] = 10.0; A[0,1] = -1.0; A[0,2] = 2.0; A[0,3] = 0.0;
A[1, 0] = -1.0; A[1, 1] = 11.0; A[1, 2] = -1.0; A[1, 3] = 3.0;
A[2, 0] = 2.0; A[2, 1] = -1.0; A[2, 2] = 10.0; A[2, 3] = -1.0;
A[3, 0] = 0.0; A[3, 1] = 3.0; A[3, 2] = -1.0; A[3, 3] = 8.0;

Matrix.ZeroTolerance = 1.0E-10;


A.LUDecomposition(L,U,P)
Matrix D = P*A-L*U; //D should be zero
```

8.5.35 **Matrix Minor(int i ,int j)**

Returns the matrix which is the minor at the (i,j) station. That is, the matrix whose elements are those of the original matrix but with the i th row and j th column removed.

C# example:

See the code corresponding to the Determinant function.

8.5.36 **Matrix Pdot(Matrix M)**

If A is the parent matrix and both A and M have dimensions $m$by n, then this function returns the matrix whose (i,j)th element is $A_{i,j}M_{i,j}$.

8.5.37 **QRDecomposition(Matrix Q , matrix R)**

Given the m by n matrix A , this method computes an m by m orthogonal matrix Q and an m by n upper triangular matrix R such that $A = QR$. Columns are zeroed using a series of Givens transformations.

C# example:

```
Matrix A = new Matrix(3,3);
A[0,0]=1.0;A[0,1]=2.0;A[0,2]=3.0;
A[1,0]=2.0;A[1,1]=1.0;A[1,2]=5.0;
A[2,0]=6.0;A[2,1]=5.0;A[2,2]=1.0;
Matrix U = Matrix.Identity(3);
Matrix V = Matrix.Identity(3);
A.QRDecomposition(Q,R);
Matrix B = A – Q*R; //should be the zero matrix
```

8.5.38 **Int Rank()**

Returns the numerial rank of the matrix. That is the maximum number of linearly independant rows or columns. Matrix.zero_tolerance is used as the criterion for a singular value being zero or not in this determination.

C# example:

```
int p;
p = Math.Min(Nrows,Ncols);

Vector sing = new Vector(p);
int conv = 0;

sing = this.SingularValues(conv);

int rank = 0;
for (rank = 0; rank < p; rank++)
{
  if (sing[rank] < Matrix.zero_tolerance)
  {
    break;
  }
}
```

8.5.39 **int RealSchurDecomposition(Matrix Q ,Matrix T)**

This routine calculates an orthogonal matrix Q and a quasi-triangular matrix T , such that $Q^T AQ = T$. A matrix is quasi-triangular if its diagonal elements are either 1 by 1 or 2 by 2 matrices. In the latter case, all 2 by 2 diagonal elements have complex eigenvalues.

If the underlying numerical routine has not converged zero is returned, otherwise one is returned.

The routine used is a QR algorithm employing a Francis QR step. This should converge in most cases. The convergence criteria is determind by *Matrix.ZeroTolerance* and the maximum number of steps in the iteration by *Matrix.MaxNumberOfLoops.* It may be possible to allow the routine to converge by increasing one or both of these parameters.

C# example:

```
Matrix A = new Matrix(5,5);
Matrix Q = Matrix.Identity(5);

A[0,0] = 2.0; A[0,1] = 3.0; A[0,2] = 0.0; A[0,3] = 1.0; A[0,4] = 0.0;
A[1,0] = 1.0; A[1,1] = 4.0; A[1,2] = 0.0; A[1,3] = 3.0; A[1,4] = 0.0;
A[2,0] = 10.0; A[2,1] = 1.0; A[2,2] = 4.0; A[2,3] = 4.0; A[2,4] = 0.0;
A[3,0] = 3.0; A[3,1] = 2.0; A[3,2] = 50.0; A[3,3] = 6.0; A[3,4] = 0.0;
A[4,0] = 2.5; A[4,1] = 4.0; A[4,2] = 0.0; A[4,3] = 8.0; A[4,4] = 7.0;

Matrix T = Matrix.Identity(5);

A.RealSchurDecomposition(Q,T);
```

8.5.40 **RowHouseholder(Vector v)**

Given the m-by -n matrix A and a non-zero m-vector v with $v[0] = 1.0$,this method overwrites A with PA, where $P = I - 2\frac{vv^T}{v^T v}$. The calculations are performed in a computationally efficient manner. If v is not an m-vector an error is thrown.

8.5.41 **SetColumnVector(int i ,Vector v)**

Sets the i th column of the matrix to have the same elements as the vector v .If i is not a valid column index or the dimension of v does not equal the number of rows of the matrix an error is thrown.

8.5.42 **SetDiagonal(Vector v)**

Sets the diagonal entries of a square matrix to be those of the elements of v . The matrix does not have to be square. If the dimension of v is does not equal the minimum of the number of rows and columns of the matrix an error is thrown.

8.5.43 **SetRowVector(int i ,Vector v)**

Sets the i th row of the matrix to have the same elements as the vector v .If i is not a valid row index or the dimension of v does not equal the number of columns of the matrix an error is thrown.

8.5.44 **SetSubMatrix(int i_1,int i_2,int j_1,int j_2,Matrix A)**

Sets the elements spanning rows i_1 to i_2 and columns j_1 to j_2 to the elements of the matrix A. j_2. If $i_2 < i_1$ or $j_2 < j_1$ or the i's or j's do not represent valid indices or the dimensions of A are not compatible an error is thrown.

8.5.45 **int SingularValues(Vector $sValues$)**

Populates the vector $sValues$ containing the p singular values of the matrix ordered in descending order of magnitude. Here $p = Min(nrows, ncols)$. If the underlying iteration converges 1 is returned, otherwise 0 is returned.

8.5.46 **Matrix Softmax_normalisation()**

For each column m of the parent M this function computes the normalised column $\bar{m}$ where

$$\bar{m}_i = \frac{\exp{(m_i)}}{\sum_{j=1}^{n} exp(m_j)}$$

where $1 \le i \le n$. n is the number of rows of M. This puts every element of the column in the range [0, 1]. The new matrix $\bar{M}$ so constructed is returned.

8.5.47 **int SymmetricOverRelaxation(Vector b,Vector xo, double w, Vector x)**

If A is the parent matrix this routine uses the SOR iterative method to solve the linear system $Ax = b$, starting with the initial estimate of xo. If the system converges successfully 1 is returned. If the system fails to converge after *MaxNumberOfLoops* iterations 0 is returned. If x_k is the estimate at the kth step the algorithm is deemed to have converged when $\|x_k - x_{k-1}\| < ZeroTolerance$. If the dimensions of A, b, x or xo are incompatible or the matrix is not square an error is thrown. If the algorithm encounters a pivot element ($A_{i,i}$) which is zero an error is thrown.

C# example:

See 8.5.33.

8.5.48 **Matrix SubMatrix(int i_1,int i_2,int j_1,int j_2)**

Gets the submatrix with rows spanning i_1 to i_2 and columns spanning j_1 to j_2. If $i_2 < i_1$ or $j_2 < j_1$ or the i's or j's do not represent valid indices an error is thrown.

8.5.49 **int SVD(Matrix U,Matrix S,Matrix V)**

8.5.50 **int JacobiSVD(Matrix U,Matrix S,Matrix V)**

Given an m by n matrix , these methods computes an m by m orthogonal matrix U, an n by n orthogonal matrix V and an m by n diagonal matrix S such that $A = USV^T$. The symmetry of the decomposition means that both the cases $m \ge n$, and $m < n$ are handled by one routine. If the SVD algorithm converges this function returns 1; otherwise it returns 0.

The diagonal elements of S contain the singular values in descending order of magnitude.

The first min(m, n) columns of U contain the corresponding left singular vectors, u_i, and the first min(m, n) columns of V contain the corresponding right singular vectors, v_i, where $Av_i = \sigma_i u_i \ (1 \le i \le \min(m,n))$. If $m >> n$ or $n << m$, and all the left and right singular vectors are not required and workspace memory is at a premium, it is recommended that the algorithm SVDLight is used instead.

The number of iterations that will be performed in an attempt to converge on the solution is governed by *MaxNumberOfLoops*. The threshold for a number being zero is given by *ZeroTolerance*. If the algorithm does not converge you can always try to ensure a convergence by increasing *MaxNumberOfLoops* or *ZeroTolerance* or both.

Jacobi_SVD uses a series of Jacobi transformations to successively zero off-diagonal elements. SVD uses a Golub-Kahan step to reduce the magnitude of off diagonal elements of a bidiagonal matrix. Jacobi_SVD takes many more iterations to zero all off diagonal elements but the amount of computation done at any one iteration is smaller than that of SVD.

C# example:

```
int converged = 0;
Matrix S = new Matrix(A);
Matrix U = Matrix.Identity(A.nrows);
Matrix V = Matrix.Identity(A.ncols);
converged = A.SVD(U,S,V);
//converged = A.JacobiSVD(U,S,V);
Matrix B = A - U*S*V.Transpose(); //should be the zero matrix
```

8.5.51 **int SVDLight(Matrix U ,Matrix S ,Matrix V)**

Let the parent A be an m by n matrix. If $m \ge n$, this method computes an m by n matrix U whose columns are orthogonal, and n by n orthogonal matrix V and an n by n diagonal matrix S . If $m < n$, this method computes an m by m orthogonal matrix U , an n by m matrix V whose columns are orthogonal and an m by m diagonal matrix S . In both cases $A = USV^T$.

If the SVD algorithm converges this function returns 1; otherwise it returns 0.

The diagonal elements of S contain the singular values in descending order of magnitude.

The first columns of U contain the corresponding left singular vectors, u_i, and the columns of V contain the corresponding right singular vectors, v_i, where $Av_i = \sigma_i u_i$ $(1 \le i \le \min(m,n))$.

The number of iterations that will be performed in an attempt to converge on the solution is governed by *MaxNumberOfLoops*. The threshold for a number being zero is given by *ZeroTolerance*. If the algorithm does not converge you can always try to ensure a convergence by increasing *MaxNumberOfLoops* or *ZeroTolerance* or both.

C# example:

```
int converged = 0;
Matrix S = new Matrix.Identity(A.Ncols);
Matrix U = new Matrix(A.nrows,A.Ncols);
Matrix V = Matrix.Identity(A.Ncols);
converged = A.SVDLight(U,S,V);
Matrix B = A - U*S*V.Transpose(); //should be the zero matrix
```

8.5.52 **SVDSolve(Vector b, ref int $conv$)**

Given the m by n matrix A and the m vector b, this method solves (in a least squares sense) for the n vector x ,where $Ax = b$. That is, x minimizes $\|Ax - b\|$ and has the smallest norm of all minimizers. $conv$ indicates if the underlying SVD routine converged ($conv$ set to 1) or not (0).

```
int converged = 0;
Vector x = A.SVDSolve(b, ref converged);
```

8.5.53 **Matrix Transpose()**

Returns the transpose of the matrix.

9 ComplexMatrix class

9.1 Constructors

9.1.1 **ComplexMatrix()**

The default constructor initialises a 3-by-3 matrix, of complex numbers, each element of which is zero. Rows are represented by the first index and columns by the second index. Both indices are zero based.

C# example:

```
ComplexMatrix A = new ComplexMatrix();
```

9.1.2 **ComplexMatrix(int m,int n)**

This constructor initialises an m by n matrix, of complex numbers, each element of which is zero.

C# example:

```
ComplexMatrix A = new ComplexMatrix(3,4);
```

9.1.3 **ComplexMatrix(ComplexMatrix A)**

This constructor initialises a matrix with the same number of rows and columns as A. The elements of the matrix are the same as those of A.

C# example:

```
ComplexMatrix A = new ComplexMatrix(2,3);

ComplexMatrix B = new ComplexMatrix(A);
```

9.2 Properties

9.2.1 **Diagonal**

Returns the vector containing the diagonal entries of the matrix. The matrix does not have to be square.

9.2.2 **int Nrows**

A read only property giving the number of rows of the matrix.

C# example:

```
int m;

m = A.Nrows;
```

9.2.3 **Ncols**

A read only property giving the number of columns of the matrix.

C# example:

```
int n;
n = A.Ncols;
```

9.2.4 [i, j]

Gets and sets the $[i, j]$ th element of the matrix.

C# example:

```
Complex a,b;
a = new Complex(1.5,0.5);
A[2,3] = a;
b = new Complex(A[1,2]);
```

9.2.5 Matrix Real

Returns the matrix whose elements are the real part of the corresponding elements of the parent.

C# example:

```
Matrix real_part = new Matrix(A.nrows,A.ncols);
Matrix imag_part = new matrix(A.nrows,A.ncols);
real_part = A.Real;
imag_part = A.Imaginary;
```

9.2.6 Matrix Imaginary

Returns the matrix whose elements are the imaginary part of the corresponding elements of the parent.

C# example:

See 9.2.5.

9.3 Static Methods

9.3.1 CalculateGivensParameters(Complex a,Complex b,Complex ref c,Complex ref s)

Given complex scalars a and b this function computes c and s such that:

$\begin{pmatrix} c & \bar{s} \\ -s & c \end{pmatrix}^T \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix}$, for some complex scalar r. Here $\bar{s}$ denotes the conjugate of s. s and c are related by $c^2 + |s| = 1$, and c is real.

9.4 Overloaded operators

9.4.1 **+ operator**

If A_1 and A_2 are real or complex matrices this operator permits $A_1 + A_2$ to return a complex matrix. If A_1 and A_2 are not of the same dimension an error is thrown.

C# example:

```
ComplexMatrix A = A1 + A1;
```

9.4.2 **Add(ComplexMatrix A)**

Adds matrix A to the parent. A may be real or complex. To be used in languages which do not permit + to be overridden.

9.4.3 **– operator**

If A_1 and A_2 are real or complex matrices this operator permits $A_1 - A_2$ to return a complex matrix. If A_1 and A_2 are not of the same dimension an error is thrown.

C# example:

```
ComplexMatrix A = A1 – A2;
```

9.4.4 **Subtract(ComplexMatrix A)**

Subtracts A from the parent. A may be real or complex. To be used in languages which do not permit - to be overridden.

9.4.5 *** operator**

- If A is a complex matrix and λ is a scalar this operator permits both $\lambda * A$ and $A * \lambda$ to return a matrix. The multiplication of the matrix by the scalar is component-wise.
- If A_1 and A_2 are two complex matrices of compatible dimensions for matrix multiplication this operator permits $A_1 * A_2$ to return a complex matrix. If the dimensions are not compatible an error is thrown.
- If A is a complex matrix and v a vector this operator permits $A * v$ to return a complex vector. If A and v are not of compatible dimensions an error is thrown.

C# example:

```
ComplexMatrix A = ComplexMatrix.Identity();

ComplexMatrix B = (lambda1 * A) * lambda2;

ComplexMatrix C = A*B;

ComplexVector v = new ComplexVector(3);

V[0]=1.0; v[1] = 2.0;

ComplexVector w = A*v;
```

9.4.6 **Multiply(double d)**

Multiplies the parent by d. d may be real or complex. To be used in languages which do not permit * to be overridden.

9.4.7 **Multiply(ComplexMatrix A)**

Permits the parent to be multiplied by A. A may be real or complex. To be used in languages which do not permit * to be overridden.

9.4.8 **Multiply(ComplexVector v)**

Permits matrix vector multiplication of the parent. v may be real or complex. To be used in languages which do not permit * to be overridden.

9.4.9 **operator /**

If A is a complex matrix and λ a scalar this operation returns A/λ in a component-wise manner. If λ is zero an error is thrown.

C# example:

```
ComplexMatrix A = (B/2.5);
```

9.4.10 **Divide(double d)**

Divides the parent by d. d can be real or complex. To be used in languages which do not permit / to be overridden.

9.4.11 **operator ==**

Returns true if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise returns false.

9.4.12 **operator !=**

Returns false if two classes are identical i.e. they refer to the same instance, or two instances are element-wise identical. Otherwise returns true.

9.5 Methods

9.5.1 **Assign**

If A_1 and A_2 are two complex matrices of the same dimensions this is the C# way of performing the assignment $A_1 = A_2$. If A_1 and A_2 are not of the same dimensions an error is thrown.

C# example:

```
A1.Assign(A2);
```

9.5.2 Bidiagonalization(ComplexMatrix U ,Complexmatrix V)

If A is the m-by-n parent matrix, ($m \geq n$), this method overwrites A with $B = U^H AV$, where B is upper bi-diagonal, U is an m-by - m unitary matrix, and V is an n- by - n unitary matrix. A can be recovered by calculating UBV^H . Rows and columns are zeroed using a series of Givens transformations.

C# example:

```
ComplexMatrix A = new ComplexMatrix(4,3);
...
ComplexMatrix U = ComplexMatrix.Identity(4);
ComplexMatrix V = ComplexMatrix.Identity(3);
A.Bidiagonalization(U,V);
```

9.5.3 ColumnHouseholder(Vector v)

Given the m-by -n matrix complex A and a non-zero n-vector v with $v[0] = 1.0$,this method overwrites A with AP , where $P = I - 2\frac{vv^H}{v^H v}$. The calculations are performed in a computationally efficient manner. If v is not an n-vector an error is thrown.

9.5.4 ComplexMatrix ConjugateTranspose()

Returns the conjugate transpose of the matrix.

9.5.5 double Determinant(ref int *converged*)

Calculates the determinant of a square matrix directly. If the matrix is non-square an error is thrown. If the underlying numerical routine has converged *converged* is set to 1. Otherwise it is set to zero.

C# example:

```
int conv = 1;
double det = A.Determinant(ref conv);
```

9.5.6 void Edit()

This method displays a dialogue which permits the elements of the matrix to be changed by the user. To change a cell: select it, type in the new value and click elsewhere so the focus of the cell is lost. Press the update button to commit the changes to the matrix. Press the cancel button to quit the form without committing any changes. Rows or columns of the matrix are plotted individually. It is possible to choose either the real, imaginary or magnitude of each element, to be plotted.

C# example:

```
ComplexMatrix A = new ComplexMatrix(4,4);
A.Edit();
```

9.5.7 **int Eigenvalues(ComplexVector E)**

If A is the n by n parent matrix this routine calculates the n complex numbers, λ, which together with n vectors, v ,satisfy the equation $Av = \lambda v$ .The eigenvalues λ are returned as a complex vector E. The eigenvalues are sorted in terms of magnitude with the largest first. If the underlying numerical routine used has not converged zero isn returned, otherwise one is returned. The routine used is a QR algorithm employing a double shift. This should converge in most cases. The convergence criteria is determind by *Matrix.ZeroTolerance* and the maximum number of steps in the iteration by *Matrix.MaxNumberOfLoops.* It may be possible to allow the routine to converge by increasing one or both of these parameters.

If the matrix is not square an error is thrown.

C# example:

```
ComplexVector E = new ComplexVector(7);
int Eig_return1 = 1;
Eig_return1 = A.Eigenvalues(E);
```

9.5.8 **int Eigenvectors(ComplexVector $Eval$, ComplexMatrix $Evec$)**

If A is the n by n parent matrix this routine calculates the n Eigenvalues, λ, which together with the n Eigenvectors v, satisy the equation $Av = \lambda v$. The Eigenvalues are returned as a complex vector in $Eval$. The corresponding Eigenvectors are returned as columns of the complex matrix $Evec$. The Eigenvectors and their corresponding Eigenvalues are sorted in order of magnitude, with the largest first. If the underlying numerical routine has not converged zero is returned, otherwise one is returned.

The routine used is a QR algorithm employing a double shift. This should converge in most cases. The convergence criteria is determind by *Matrix.ZeroTolerance* and the maximum number of steps in the iteration by *Matrix.MaxNumberOfLoops.* It may be possible to allow the routine to converge by increasing one or both of these parameters.

If the matrix is not square an error is thrown.

C# example:

```
Eig_return = A.Eigenvectors(E,P);
if (Eig_return == 1)
{
    for (i = 0; i < n; i++)
    {
        ComplexVector diff = new ComplexVector(n);
        diff = E[i] * P.GetColumnVector(i) - A *
P.GetColumnVector(i);
        if (diff.Modulus > (((double)n)*1.0E-8))
        {
            MessageBox.Show("Complex Eigenvector routine has
failed");
            return;
        }
    }
}
```

9.5.9 **ComplexMatrix Eigenvectors2(ComplexMatrix Q ,ComplexMatrix T)**

A is the n by n parent matrix and Q is a unitary matrix an T an upper triangular matrix such that $Q^H AQ = T$. Then this routine computes the eigenvectors of A as the return matrix. The i th column of the return matrix corresponds to the ith eigenvalue on the diagonal of T .

9.5.10 **Complex Eigenvector(ComplexVector $Eval$,ComplexVector $initial$, ref int $conv$)**

Use this routine when it is known that the matrix has a single dominant eigenvalue i.e. $|\lambda_1|>|\lambda_2|\geq|\lambda_3|\geq\ldots\geq|\lambda_n|$. In this case the routine will converge to λ_1 (the return value) and the associated dominant eigenvector ($Eval$). The initial guess to the dominant eigenvector must have a component in the direction of the dominant eigenvector. The routine employs the power method algorithm. This routine is much less computational intensive than the ones for returning all the eigenvector/values, and should be used when some information is known about the matrix.

9.5.11 **ComplexVector GetColumnVector(int i)**

Return the i th column of the matrix as a vector. If i is less than zero or greater or equal to the number of columns then an error is thrown.

9.5.12 **ComplexVector GetRowVector(int i)**

Returns the i th row of the matrix as a vector. If i is less than zero or greater or equal to the number of rows then an error is thrown.

9.5.13 **GivensPostMultiplication(int i ,int k ,Complex c ,Complex s)**

Let $G(i,k,c,s)$ be the Givens matrix, where c and s are calculated according to 9.3.1. This is the identity except that

$$\begin{aligned} G(i,k,c,s)(i,i) &= c \\ G(i,k,c,s)(k,k) &= c \\ G(i,k,c,s)(i,k) &= \bar{s} \\ G(i,k,c,s)(k,i) &= -s \end{aligned}$$

The Givens post multiplication of a matrix A replaces A with $AG(i,k,c,s)$.

9.5.14 **GivensPreMultiplication(int i ,int k ,Complex c ,Complex s)**

Let $G(i,k,c,s)$ be the Givens matrix, where c and s are calculated according to 9.3.1. This is the identity except that

$$\begin{aligned} G(i,k,c,s)(i,i) &= c \\ G(i,k,c,s)(k,k) &= c \\ G(i,k,c,s)(i,k) &= \bar{s} \\ G(i,k,c,s)(k,i) &= -s \end{aligned}$$

The Givens pre multiplication of a matrix A replaces A with $G(i,k,c,s)^T A$.

9.5.15 **void HessenbergReduction()**

The parent is operated upon to reduce it to an upper Hessenberg form i.e. zeros on all elements below the sub-diagonal. The reduction is done using a series of Givens transformations. If the matrix is not square an error is thrown.

9.5.16 **void HessenbergReduction(ComplexMatrix Q)**

This method operates on the parent matrix, A, to reduce it to an upper Hessenberg form H. The reduction is done using a series of Givens transformations. A unitary matrix, Q, is constructed such that $Q^{*T}AQ = H$. If the matrix is not square an error is thrown.

C# example:

```
ComplexMatrix A = new ComplexMatrix(7,7);
...
ComplexMatrix Q = ComplexMatrix.Identity(4);
A.HessenbergReduction(Q);
```

9.5.17 **HouseholderBidiagonalization(ComplexMatrix U ,ComplexMatrix V)**

If A is the m-by-n matrix, ($m \geq n$), this method overwrites A with $B = U^H AV$, where B is upper bi-diagonal, U is an m-by - m unitary matrix, and V is an n- by - n unitary matrix. Here the 'H' subscript will be used to denote conjugate transpose. A can be recovered by calculating UBV^H. Rows and columns are zeroed using a series of Householder transformations.

C# example:

```
ComplexMatrix A = new ComplexMatrix(4,3);
ComplexMatrix U = ComplexMatrix.Identity(4);
ComplexMatrix V = ComplexMatrix.Identity(3);
A.HouseholderBidiagonalization(U,V);
```

9.5.18 **HouseholderQRDecmposition(ComplexMatrix Q ,ComplexMatrix R)**

Given the m by n complex matrix A, this method computes an m by m orthogonal matrix Q and an m by n upper triangular matrix R such that $A = QR$. Columns are zeroed using Householder transformations.

C# example:

```
ComplexMatrix A = new ComplexMatrix(3,3);
A[0,0]=new Complex(1.0,0.0);A[0,1]=new Complex(2.0,0.0);A[0,2]=new
Complex(3.0,0.0);
A[1,0]=new Complex(2.0,0.0);A[1,1]=new Complex(3.0,0.0);A[1,2]=new
Complex(5.0,1.0);
A[2,0]=new Complex(6.0,1.0);A[2,1]=new Complex(5.0,0.0);A[2,2]=new
Complex(1.0,1.0);
ComplexMatrix Q = ComplexMatrix.Identity(3);
ComplexMatrix Q = ComplexMatrix.Identity(3);
A.HouseholderQRDecomposition(Q,R);
ComplexMatrix B = A - Q*R; //should be the zero matrix
```

9.5.19 **int Inverse(ComplexMatrix inv)**

Calculates (in inv) the inverse of a square matrix.

The return value of the function indicates the following:

Return value	Meaning
0	Successful calculation of the inverse.
1	The matrix is non-square - an error is thrown.
2	The numerical method would not converge on a solution
3	The matrix is singular (i.e. an inverse doesn't exist.

C# example:

```
int inverse calculated = 0;
inverse_calculated = A.Inverse(inv);
```

9.5.20 **Boolean IsBidiagonal()**

Returns TRUE if the matrix is upper bidiagonal, to within the tolerance given by *Matrix.ZeroTolerance*. Otherwise the function returns FALSE.

9.5.21 **Boolean IsHessenberg()**

Returns TRUE if the matrix is upper hessenberg, to with the tolerance given by

Matrix.ZeroTolerance. Otherwise the function returns FALSE.

9.5.22 **Boolean IsUnitary()**

Returns TRUE if the square matrix is unitary, to within the tolerance given by Matrix.zero_tolerance. Otherwise the function returns FALSE.

9.5.23 **Boolean IsZero()**

Return true if all entries in the matrix are zero to within the tolerance given by Matrix.zero_tolerance. Otherwise the function returns false.

9.5.24 **ComplexMatrix Minor(int i ,int j)**

Returns the matrix which is the minor at the (i, j) station. That is, the matrix whose elements are those of the original matrix but with the i th row and j th column removed.

C# example:

See the code corresponding to the Determinant function.

9.5.25 **RealBidiagonalization(ComplexMatrix U, ComplexMatrix V)**

If A is the m-by-n parent matrix, ($m \geq n$), this method overwrites A with $B = U^H AV$, where B is a real and upper bi-diagonal, U is an m-by - m unitary matrix, and V is an n- by - n unitary matrix. A can be recovered by calculating UBV^H. Rows and columns are zeroed, and transformed to real, using a series of orthogonal transformations.

C# example:

```
ComplexMatrix A = new ComplexMatrix(4,3);

...

ComplexMatrix U = ComplexMatrix.Identity(4);

ComplexMatrix V = ComplexMatrix.Identity(3);

A.RealBidiagonalization(U,V);
```

9.5.26 **Int Rank()**

Returns the numerial rank of the matrix. That is the maximum number of linearly independant rows or columns. Matrix.zero_tolerance is used as the criterion for a singular value being zero or not in this determination.

C# example:

```
int p;
p = Math.Min(nrows,ncols);

Vector sing = new Vector(p);
int conv = 0;

conv = this.SingularValues(sing);

int rank = 0;
for (rank = 0; rank < p; rank++)
{
  if (sing[rank] < Matrix.zero_tolerance)
  {
    break;
  }
}
```

9.5.27 **RowHouseholder(Vector v)**

Given the m-by-n complex matrix A and a non-zero m-vector v with $v[0]=1.0$,this method overwrites A with PA, where $P = I - 2\frac{vv^H}{v^H v}$. The calculations are performed in a computationally efficient manner. If v is not an m-vector an error is thrown.

9.5.28 **int SchurDecomposition(ComplexMatrix Q, ComplexMatrix T)**

This routine calculates a unitary matrix Q and an upper triangular matrix T , such that $Q^H AQ = T$.

If the underlying numerical routine has not converged zero is returned, otherwise one is returned.

The routine used is a QR algorithm employing a double shift. This should converge in most cases. The convergence criteria is determind by *Matrix.ZeroTolerance* and the maximum number of steps in the iteration by *Matrix.MaxNumberOfLoops.* It may be possible to allow the routine to converge by increasing one or both of these parameters.

9.5.29 SetColumnVector(int i ,Vector v)

Sets the i th column of the matrix to have the same elements as the vector v.If i is not a valid column index or the dimension of v does not equal the number of rows of the matrix an error is thrown.

9.5.30 SetDiagonal(ComplexVector v)

Sets the diagonal entries of a square matrix to be those of the elements of v. The matrix does not have to be square. If the dimension of v is does not equal the minimum of the number of rows and columns of the matrix an error is thrown.

9.5.31 SetRowVector(int i ,ComplexVector v)

Sets the i th row of the matrix to have the same elements as the vector v.If i is not a valid row index or the dimension of v does not equal the number of columns of the matrix an error is thrown.

9.5.32 SetSubMatrix(int i_1,int i_2 ,int j_1,int j_2 ,ComplexMatrix A)

Sets the elements spanning rows i_1 to i_2 and columns j_1 to j_2 to the elements of the matrix A. j_2. If $i_2 < i_1$ or $j_2 < j_1$ or the i 's or j 's do not represent valid indices or the dimensions of A are not compatible an error is thrown.

9.5.33 int SingularValues(Vector $sValues$)

Populates the vector $sValues$ containing the p singular values of the matrix ordered in descending order of magnitude. Here $p = Min(nrows, ncols)$. If the underlying iteration converges 1 is returned, otherwise 0 is returned.

9.5.34 ComplexMatrix SubMatrix(int i_1,int i_2 ,int j_1,int j_2)

Gets the submatrix with rows spanning i_1 to i_2 and columns spanning j_1 to j_2. If $i_2 < i_1$ or $j_2 < j_1$ or the i 's or j 's do not represent valid indices an error is thrown.

9.5.35 int SVD(ComplexMatrix U ,ComplexMatrix S ,ComplexMatrix V)

Given an m by n complex matrix , this method computes an m by m unitary matrix U , an n by n unitary matrix V and an m by n diagonal matrix S such that $A = USV^H$. The symmetry of the decomposition means that both the cases $m \geq n$, and $m < n$ are handled by one routine. If the SVD algorithm converges this function returns 1; otherwise it returns 0.

The diagonal elements of S contain the singular values in descending order of magnitude. The number of iterations that will be performed in an attempt to converge on the solution is governed by *MaxNumberOfLoops*. The threshold for a number being zero is given by *ZeroTolerance*. If the algorithm does not converge you can always try to ensure a convergence by increasing *MaxNumberOfLoops* or *ZeroTolerance* or both.

C# example:

```
int converged = 0;
ComplexMatrix S = new ComplexMatrix(A);
ComplexMatrix U = ComplexMatrix.Identity(A.nrows);
ComplexMatrix V = ComplexMatrix.Identity(A.ncols);
converged = A.SVD(U,S,V);
```

```
ComplexMatrix B = A - U*S*V.ConjugateTranspose(); //should be the
zero matrix
```

9.5.36 **SVDSolve(ComplexVector b, ref int $conv$)**

Given the m by n matrix A and the m vector b, this method solves (in a least squares sense) for the n vector x ,where $Ax = b$. That is, x minimizes $\|Ax - b\|$ and has the smallest norm of all minimizers. $conv$ indicates if the underlying SVD routine converged ($conv$ set to 1) or not (0).

```
int converged = 0;
ComplexVector x = A.SVDSolve(b, ref converged);
```

9.5.37 **ComplexMatrix Transpose()**

Returns the transpose of the matrix.

10 Neural Network class

The methods and properties in this class are aimed at solving those classes of problems that can broadly be classed as categorisation. The typical problem will have two matrices of data associated with it. Firstly an input matrix , each of whose columns contain data describing the features of an object. Each element of the rows is the data of a feature. For example the objects might be people's faces (taking with your camera) and one particular feature could be the distance between the two pupils. Another feature could be an integer representing the eye colour etc. Secondly a target matrix T, each of whose columns represents the category of object that the data of the corresponding column of the input matrix belongs to. The column will have as many elements as there are categories and will be all zeroes with the exception of a one at the location of the category to which the data is associated. For example the data may represent the faces of pupils in a school and the categories are the classes the pupils belong to. The job of the neural network is to provide a system, which given the features of an unknown object, will permit this object to be categorised into one of a finite number of categories. The mathematical way in which it does this is described next.

Let s be the number of features, n the number of categories and N the number of elements in the data set. Then the neural network finds a weight matrix W, such that:

$$F(W) = \frac{1}{N}\|f(WI) - T\|^2$$

Equation 1

is minimised. Here f is the activation function (described later). Normally this is the softmax transfer function whose job is to normalise the columns of WI, such that all elements are between 0 and 1 and hence are more comparable to the columns of T. The minimisation is essentially a least square one summing over all N columns of $f(WI) - T$. W is an n by s matrix. That is if e_i is the ith column of $f(WI) - T$, $F(W) = \frac{1}{N} min_W \sum_{i=1}^{N} e_i^T e_i$.

The technique used is a conjugate gradient like one where W is incrementally calculated from an initial estimation W_0 where at the kth step we have, essentially:

$$W_k = W_{k-1} + \mu d_k$$

Equation 2

$$d_k = -\nabla F(W) + \beta d_{k-1}$$

Equation 3

Here $\nabla F(W)$ is an n by s matrix whose i,j th element is $\frac{\partial F}{\partial W_{ij}}$, β is a scalar. Possible values for β are given by 10.1.5 and 10.1.6. $\nabla F(W)$ can be computed analytically. μ is a scaling factor (the distance we move along d_k) whose value is computed to minimise $F(W_k)$ at the kth step.

This process is referred to as training the network. Once trained and given an input vector p not present in the training data set, the category this object belongs to can be estimated by selecting the one corresponding to the index of the maximal element in $f(Wp)$.

You should use a neural network technique like this when:

- An analytic technique is not available.
- You have a large enough data set.
- You do not mind that the wrong category may sometimes be allocated after training, for a different input vector.

A typical example of the use of the methods and properties of this class is:

```
NeuralNetwork net = new NeuralNetwork();
net.inputs = Data_I.Transpose();
net.targets = Data_T.Transpose();
net.maxNumberOfExperiments = 10000;
net.mu = 0.005;
net.gradientTolerance = 1.0E-7;
net.fixedStepsize = false;
net.showGraphic = true;
net.addBias = true;
net.train();

Graph gr = new Graph();

Matrix a = net.weights * net.inputs;
Matrix a_ = a.Softmax_normalisation();
Matrix error = net.targets - a;
int N = net.targets.Ncols;
double mod = 0.0;
Vector ind = new Vector(N);
Vector errorV = new Vector(N);
for (int i = 0; i < N; i++)
{
    ind[i] = i;
    mod = error.GetColumnVector(i).Modulus;
    errorV[i] = mod * mod / N;
}

gr.Plot(ind, errorV);

Matrix TbarT = a_.Transpose();

double success = 0.0;
for (int i = 0; i < N; i++)
{
    int loc1 = 0, loc2 = 0;
    TbarT.GetRowVector(i).MaxL(ref loc1);
    Data_T.GetRowVector(i).MaxL(ref loc2);
    if (loc1 == loc2)
    {
        success = success + 1.0;
    }
}

MessageBox.Show("Success rate is:" + (100.0 * success /
N).ToString() + "%");
```

The inputs and targets are added. The parameters of training are set and the network is then trained to find the weights. Then in this example the original data is used to reconstruct the targets which are compared with the original ones. Finally the success rate is displayed.

10.1 Properties

10.1.1 **Boolean addBias**

When set to true this instructs the algorithm to use a bias vector (see property bias), with the train method. The default is false.

10.1.2 **Boolean addBias1, addBias2**

When set to true these instruct the algorithm to use a bias vector (see property bias1, bias2), with the trainHL method, associated with weights $W_1$and W_2respectively. Each bias can be set independently. The default for each is false.

10.1.3 **Boolean accelerateConvergence**

When set to false a more conservative method for updating the weights is used at each iteration. This may mean the training algorithm may take much longer to converge but it may also mean that ultimately weights with a greater predictive categorisation capacity are obtained. The default is true and should produce very good results in most cases.

10.1.4 **int activationFunction**

This specifies the function f in Equation 1.

0 (default) - softmax transfer function.

1 - linear transfer function (the identity).

10.1.5 **double beta**

The value β in Equation 3, as provided by the user. The default value is 0.9.

10.1.6 **int betaType**

This allows specification of the way that β in Equation 3 is calculated. Various options are available and each one's performance will vary depending upon the values in I and T.

0 (the default), due to Hestenes - Steifel:

$$\beta = \frac{g_k \cdot \Delta g_k}{d_k \cdot \Delta g_k}$$

Equation 4

1 , due to Fletcher-Reeves:

$$\beta = \frac{g_k \cdot g_k}{g_{k-1} \cdot g_{k-1}}$$

Equation 5

2, β as provided by the user according to property 10.1.5.

3, due to Polak-Ribiere:

$$\beta = \frac{g_k \cdot \Delta g_k}{g_{k-1} \cdot g_{k-1}}$$

Equation 6

In these expressions g_k is the gradient at the kth iteration and $\Delta g_k = g_k - g_{k-1}$. The '.' Is the dot product of the two vectors.

10.1.7 **Vector bias**

This is a *single* n dimensional vector b, such that if for each i , $\bar{e}_i = f(Wp_i) - b$ (p_i is the ith column of I) then we find a matrix $\bar{W}$ that minimizes $\bar{F}(\bar{W}) = \frac{1}{N}\sum_{i=1}^{N} \bar{e}_i^T \bar{e}_i$. Let $\bar{I}$ be the matrix of dimension $s + 1$ by N whose first s rows are filled with the elements of I , and whose last row contains all -1's. Let $\bar{W}$ be the n by $s + 1$ matrix whose first s columns are filled with the elements of W and whose last column is filled with the elements of b. Then $\bar{W}$ and $\bar{I}$ are sometimes termed the augmented matrices. They have the property that when the term $f(\bar{W}\bar{p}_i)$ is evaluated ($\bar{p}_i$ is the ith column of $\bar{I}$) the index of the maximum value corresponds to the category (if successful) corresponding to the features. The beauty is that all algebraic expressions are analogous for this approach and the approach without a bias vector. The bias can be got but not set. When using a bias is selected the properties inputs and weights return the corresponding augmented matrices.

Experience suggests that using a bias can often slightly improve the reliability of the neural method.

10.1.8 **Vector bias1, bias2**

These are *single* n dimensional vectors b_1 and b_2, such that if for each i, $\bar{e}_i = f(W_2\tau(W_1 p_i - b_1) - b_2)$ (p_i is the ith column of I) then we find matrices $\bar{W}_1$ and $\bar{W}_2$that minimize $\bar{F}(\bar{W}_1, \bar{W}_2) = \frac{1}{N}\sum_{i=1}^{N} \bar{e}_i^T \bar{e}_i$. Let $\bar{I}$ be the matrix of dimension $s+1$ by N whose first s rows are filled with the elements of I , and whose last row contains all -1's. Let h be the size of the hidden layer. Let $\bar{W}_1$ be the h by $s+1$ matrix whose first s columns are filled with the elements of W_1 and whose last column is filled with the elements of b_1. Let $\bar{W}_2$ be the n by $h+1$ matrix whose first h columns are filled with the elements of W_2 and whose last column is filled with the elements of b_2. $\bar{W}_1$and $\bar{W}_2$and $\bar{I}$ are sometimes termed the augmented matrices, as before. They have the property that when the term, $f\left(\bar{W}_2\overline{\tau(\bar{W}_1\bar{p}_\iota - b_1)} - b_2\right)$ is evaluated the index of the maximum value corresponds to the category (if successful) corresponding to the features. Here $\overline{\tau(\bar{W}_1\bar{p}_\iota - b_1)}$ is another augmented input. As with the single weight case all algebraic expressions are analogous to the approach without bias vector(s). A coding fragment as to how to use these matrices is given in 10.2.11.

10.1.9 **conjugateGradientRestart**

When set to true the search direction d_k is reset to $-\nabla F(W)$ if certain conditions are met. The default is true.

10.1.10 **Boolean fixedStepsize**

A Boolean whose value determines whether μ can be varied at each step. The default is true and if false is selected a fixed step is employed (this is as provided by the user or else default to 0.005). A fixed step size is not possible with the trainHL method.

10.1.11 **double gamma**

This property allows the user to set a constant value for γ in Equation 7. The default value when specification via this property is selected, is 0.9.

10.1.12 **int gammaType**

If *conjugateGradientRestart* is set to true, then if the gradient's modulos has not decreased sufficiently, at an iteration, the algorithm is restarted by setting the search direction d_k to $-g_k$. Then, if t is the iteration at which the restart occurs, then for all $k > t+1$,

$$d_k = -g_k + \beta d_{k-1} + \gamma d_t$$

Equation 7

This approach is due to Beale and Powell.

This property provides various options for calculating γ. The options are analogous to those for calculating β (see 10.1.6). It is not necessary to use the same option as that of β, but this is usually what is done.

0 (the default) -

$$\gamma = \frac{g_k \cdot \Delta g_{t+1}}{g_t \cdot \Delta g_{t+1}}$$

Equation 8

1 - $$\gamma = \frac{g_{t+1} \cdot g_{t+1}}{g_t \cdot g_t}$$

Equation 9

2 - γ is set to a value as provided by the user, according to property 10.1.11.

3 - $$\gamma = \frac{g_k \cdot \Delta g_{t+1}}{g_t \cdot g_t}$$

Equation 10

10.1.13 **double gradientTolerance**

If $g_i$is a row of $\nabla F(W)$, where W is a weight matrix, the iteration of the training terminates if $\sum_{i=1}^{n} g_i{}^T g_i$ falls below gradientTolerance. Note that where the iteration should be converging to a local minima, we are expecting $\nabla F(W)$ to tend to the zero matrix. However it will not necessarily be zero unless an analytic solution exists which is unlikely, or otherwise a neural network technique should not be employed. Hence this test is a natural one. And in any case, when it is true, then we are not updating the guess for W any longer, anyway.

10.1.14 **int initialGuess**

This allows the user to control the way the weights are initialised at the start of the training algorithm. Experience shows that the initial guess has a marked effect upon the convergence of the algorithm and even upon the value converged to, although in all cases a value which gives a low value to the performance function will be obtained. Also, in all cases the training will be deterministic and repeatable - i.e. the results of the training depend only upon the inputs, targets and the parameters of the method such as the initial guess.

It is possible to select two methods for finding the initial weights:

0 - use a very simple method in which the leading diagonal of the weight(s) matrices are set to 1.0, all other elements being zero.

1 - use a method based upon the SVD.

10.1.15 **Matrix inputs**

The matrix I as provided by the user.

10.1.16 **double maxmu**

The maximum value that μ can reach during the minimisation. Defaults to 1.0E07. If this figure is attained the algorithm stops. Sometimes the success of training can be improved by increasing this parameter to allow greater swings in μ during the optimisation process that occurs to establish the value of μ at each iteration.

10.1.17 **int maxNumberOfExperiments**

The maximum number of iterations of the conjugate gradient like step that are permitted. Defaults to 1000.

10.1.18 **Boolean normaliseInputs**

Setting this property to true normalises the inputs according to 10.2.2 and 10.2.6. An example of when this might be used is when N pairs $(x, y)$are provided and we are trying to find the elements of the fixed matrix which relates x and y according to $y = Ax$. In this case we set the activation function to *linear* (0), and normaliseInputs to false. The default value of this property is true.

A synthetic example of how this property can be used to find the matrix A, mentioned in the above paragraph is:

```
int noFeatures = 6;
int noCategories = 12;
int noElements = 1000;
```

```
Matrix weights = new Matrix(noCategories, noFeatures);
Matrix inputs = new Matrix(noFeatures, noElements);
Matrix targets = new Matrix(noCategories, noElements);

 //Set the weights matrix
 Random rnd = new Random();
 for (int i = 0; i < weights.Nrows; i++)
 {
    for (int j = 0; j < weights.Ncols; j++)
    {
        weights[i, j] = rnd.NextDouble();
    }
}

//Set the inputs and generate the targets
for (int i = 0; i < noElements; i++)
{

     Vector x = new Vector(noFeatures);
     for (int j = 0; j < x.Dim; j++)
     {
         x[j] = rnd.NextDouble();
     }
     inputs.SetColumnVector(i, x);
     targets.SetColumnVector(i, weights * x);

}

//Now, recalculate the weights using a neural network technique.
NeuralNetwork net = new NeuralNetwork();
net.inputs = inputs;
net.targets = targets;
net.maxNumberOfExperiments = 1000;
net.mu = 0.005;
net.minmu = 1.0E-10;
net.maxmu = 1000.0;
net.performanceTolerance = 1.0e-8;
net.gradientTolerance = 1.0E-9;
net.fixedStepsize = false;
net.showGraphic = true;
net.addBias = false;
net.accelerateConvergence = true;
net.initialGuess = 0;
net.activationFunction = 0;
net.steepestDescentOnly = true;
net.conjugateGradientRestart = false;
net.normaliseInputs = false;
net.train();

MessageBox.Show("Number of experiments, mu  is:" + net.numberOfExperiments.ToString()
+ "," + net.mu.ToString());

//compare with the original ones
(net.weights - weights).Edit();
```

10.1.19 int numberOfFeatures

The number of rows in the inputs matrix.

10.1.20 **int minmu**

The minimum value that μ can reach during the minimisation. Defaults to 0.00001. If this figure is attained the algorithm stops. Sometimes the success of the training process can be increased by reducing this parameter, as it tends to increase the number of iterations permitted.

10.1.21 **double mu**

The initial step size μ. Defaults to 0.005.

10.1.22 **int numberOfFeatures**

The number , s, of rows in I. This is the number of features in the data.

10.1.23 **Int numberOfTargets**

The number, , of rows in T. This is the number of categories.

10.1.24 **Vector inputScale, Vector inputGain, double inputYmin, double input Ymax**

Prior to training each row k of the input matrix I, is normalised by resetting $I[k,i]$ equal to

$$(I[k,i] - inputScale[k]) * inputGain[k] + \ inputYmin$$

Here $inputScale[k]$ is the minimum of the elements of the kth row of I. $inputYmin$is as provided by the user but defaults to -1 (it is recommended to use this value). $inputYmax$ is as provided by the user but defaults to 1 (it is recommended to use this value).

$$inputGain[k] = \frac{inputYmax - inputYmin}{\max element\ row\ k - \min element\ row\ k}$$

That is we are normalising the rows of I to be in the range $[inputYmin, inputYmax]$ i.e. usually [-1,1].

10.1.25 **double performance**

For a given W or (W_1, W_2) pair this is the value of $F(W)$ or $F(W_1, W_2)$ respectively.

10.1.26 **double performanceTolerance**

When $F(W)$is below this level the training algorithm ends. The default is 0.0001.

10.1.27 **Boolean showGraphic**

If set to true (the default) a graphic is displayed during training that enables you to see the progression of $\nabla F(W)$ or $\nabla_1 F(W_1, W_2)$ and $\nabla_2 F(W_1, W_2)$ respectively. The modulus of the rows are ploted in each case.

10.1.28 **int sizeOfHiddenLayer**

This allows the user to set the size of the hidden layer when employing trainHL (see 10.2.11).

10.1.29 **Boolean steepestDescentOnly**

When set to true this ensures β is zero in Equation 3. The default is false.

10.1.30 **Matrix targets**

The matrix T as provided by the user.

10.1.31 **Matrix weights**

The matrix W, as determined during training.

10.1.32 Matrix weights1

The matrix W_1as found using the trainHL method.

10.1.33 Matrix weights2

The matrix W_2as found using the trainHL method.

10.2 Methods

10.2.1 Matrix Augment_input(Matrix $\boldsymbol{I}$)

Augments the input matrix I as described in 10.1.1 and 10.1.2 and returns the resultant matrix.

10.2.2 void Calculate_normalisation_coefficients(Matrix $\boldsymbol{M}$)

For a matrix M, this routine computes for each rows the vectors *inputScale* and *inputGain* described in section 10.1.24.

10.2.3 double Calculate_performance(Matrix $\boldsymbol{T}$, Matrix $\bar{\boldsymbol{T}}$)

Given a matrix of targets T and those of calculated targets (e.g. from $f(WI)$), $\bar{T}$, this function calculates $\frac{1}{N}\|T - \bar{T}\|^2$ - the error vector.

10.2.4 void Calculate_softmax_gradient_function(Matrix $\boldsymbol{W}$, Matrix I,Matrix T)

Given weights W, inputs I and targets T this function returns $\nabla F(W)$.

10.2.5 void Calculate_softmax_gradients_function(Matrix $\boldsymbol{W1}$, Matrix $\boldsymbol{W2}$, Matrix $\boldsymbol{I}$, Matrix $\boldsymbol{T}$)

Given weights W_1 and W_2 , inputs I and targets T this function computes $\boldsymbol{\nabla_1} F(W_1, W_2)$ and $\boldsymbol{\nabla_2} F(W_1, W_2)$. Here the (r, s)th element of $\boldsymbol{\nabla_1} F(W_1, W_2)$ is $\frac{\partial F}{\partial {W_1}_{r,s}}$ where (r, s) ranges of the index elements of W_1, and likewise for $\boldsymbol{\nabla_2} F(W_1, W_2)$.

10.2.6 void Perform_normalisation(Matrix $\boldsymbol{M}$)

For a given matrix M, this routine works along the rows of Mapplying the normalisation of section 10.1.24.

10.2.7 Matrix Softmax_transfer_function(Matrix $\boldsymbol{M}$)

For each column m of M this function computes the normalised column $\bar{m}$ where

$$\bar{m}_i = \frac{\exp{(m_i)}}{\sum_{j=1}^{n} exp(m_j)}$$

where $1 \leq i \leq n$. This puts every element of the column in the range [0, 1]. The new matrix $\bar{M}$ so constructed is returned.

10.2.8 double tansig(double $\boldsymbol{z}$)

Returns the expression $\frac{2}{1+e^{-2z}} - 1$.

10.2.9 Matrix Tansig_transfer_function(Matrix $\boldsymbol{M}$)

Applies the tansig function of 10.2.8 to each element of the matrix M, and returns the matrix so formed.

10.2.10 **void train()**

This (key) function calculates a weight matrix using the procedure outlined by Equation 1 and Equation 2. The method employed is a deterministic one, in that given a set of inputs and targets and the parameters of the calculation (i.e. maximum number of iterations, initial μ etc) the results of the calculation are always the same (i.e. weights, number of iterations to converge etc.). This is because there is no random initial guess to seed the calculation.

10.2.11 **void trainHL()**

This is a training method which calculates the weights in two weight matrices $W_1$and W_2. The performance function that is minimised is

$$F(W_1, W_2) = \frac{1}{N}\left\| f\left(W_2\left(\tau(W_1 I)\right)\right) - T\right\|^2$$

Equation 11

Here τ is the tansig transfer function, and f is the softmax transfer function. The method is sometimes described as one which employs a ‘hidden layer’. The size of $W_2$is n by h and that of W_1 h by s, where h is the size of the hidden layer. The algorithm employed is a conjugate gradient like one, similar to Equation 2 for each weight matrix. Again the method employed is a deterministic, repeatable one.

Once trained and given an input vector p not present in the training data set, the category this object belongs to can be estimated by selecting the one corresponding to the index of the maximal element in $f\big(W_2\tau(W_1 p)\big)$.

For most applications, the train method (either with or without a bias) will suffice. However for some applications it is possible to obtain an improvement in the accuracy of the categorisation by training with trainHL. TrainHL typically takes much longer than train.

Note that an improvement of the accuracy of the categorisation on the training set may not lead over to a greater predictive power. It is possible to, too closely fit the weights to the training data such that poor predictive powers for data, not present in the training set, result. This is the phenomena of over-fitting. This is again a reason why the algorithm train is recommended for most applications, and why the default of accelerated convergence is preferred.

An example of the use of this function with 2 bias vectors is shown below.

```
NeuralNetwork net = new NeuralNetwork();
net.inputs = Data_I.Transpose();
net.targets = Data_T.Transpose();
net.maxNumberOfExperiments = 1000;
net.mu = 0.005;
net.gradientTolerance = 1.0E-7;
net.fixedStepsize = false;
net.showGraphic = true;
net.addBias1 = true;
net.addBias2 = true;

net.trainHL();

Graph gr = new Graph();

Matrix a = net.weights1 * net.inputs;
Matrix a_ = net.Tansig_transfer_function(a);
Matrix a__ = net.Softmax_transfer_function(net.weights2 * net.Augment_input(a_));
Matrix error = net.targets - a__;
int N = net.targets.Ncols;
double mod = 0.0;
Vector ind = new Vector(N);
Vector errorV = new Vector(N);
for (int i = 0; i < N; i++)
{
    ind[i] = i;
    mod = error.GetColumnVector(i).Modulus;
    errorV[i] = mod * mod / N;
}

gr.Plot(ind, errorV);

Matrix TbarT = a__.Transpose();

double success = 0.0;
for (int i = 0; i < N; i++)
{
   int loc1 = 0, loc2 = 0;
   TbarT.GetRowVector(i).MaxL(ref loc1);
   Data_T.GetRowVector(i).MaxL(ref loc2);
   if (loc1 == loc2)
   {
       success = success + 1.0;
   }
}

MessageBox.Show("Success rate is:" + (100.0 * success /
N).ToString() + "%");
```

11 Graph class

This class can be used to open a window in which vectors (line graphs) or matrices (contour plots) are plotted.

11.1 Examples

11.1.1 Example 1

```
Vector v = new Vector(100);
Vector x = new Vector(100);
double t;
for (int i=0;i<100;i++)
{
    x[i] = i;
    t = i/10.0;
    v[i] = Math.Sin(2.0*Math.PI*t);
}

Graph gr = new Graph();

gr.Plot(x,v);

gr.AddTitle("Practice plot");
gr.AddXLabel("X label");
gr.AddYLabel("Y label");
gr.AddLegend("first series");
gr.AddXLimits(0.0,100.0);
gr.AddYLimits(-1.1,1.1);
```

11.1.2 Example 2

```
//Function with a saddle point
int ms = 100;
int ns = 100;
Vector XS = new Vector(ms);
Vector YS = new Vector(ns);
Matrix ZS = new Matrix(ms,ns);
for (int i = 0 ;i < ms; i++)
{
    XS[i] = -11 + i*(22.0/ms);
    YS[i] = -11 + i*(22.0/ns);
}

for (int i = 0; i < ms; i++)
{
    for (int j = 0; j < ns; j++)
    {
       ZS[i,j] = XS[i]*XS[i] - YS[j]*YS[j];
    }
}
```

```
// Get the max and the min of the data
double maxZ = -1.0E12;
double minZ = 1.0E12;
for (int i = 0; i < m; i++)
{
    for (int j = 0; j < n; j++)
    {
        if (Z[i, j] > maxZ)
        {
           maxZ = Z[i, j];
        }
        if (Z[i, j] < minZ)
        {
            minZ = Z[i, j];
        }
   }
}

Vector v = new Vector(7);
for (int c = 0; c < 7; c++)
{
    v[c] = minZ + (c + 1) * (maxZ - minZ) / 8;
}

g.Plot(ZS);
//All the following should produce similar graphs:
//g.Plot(ZS,7);
//g.Plot(ZS,v);
//g.Plot(ZS,XS,YS);
//g.Plot(ZS,XS,YS,7);
//g.Plot(ZS,XS,YS,v);
g.AddTitle("Saddle function");
```

An image of the graph of example 2 is shown in Figure 5.

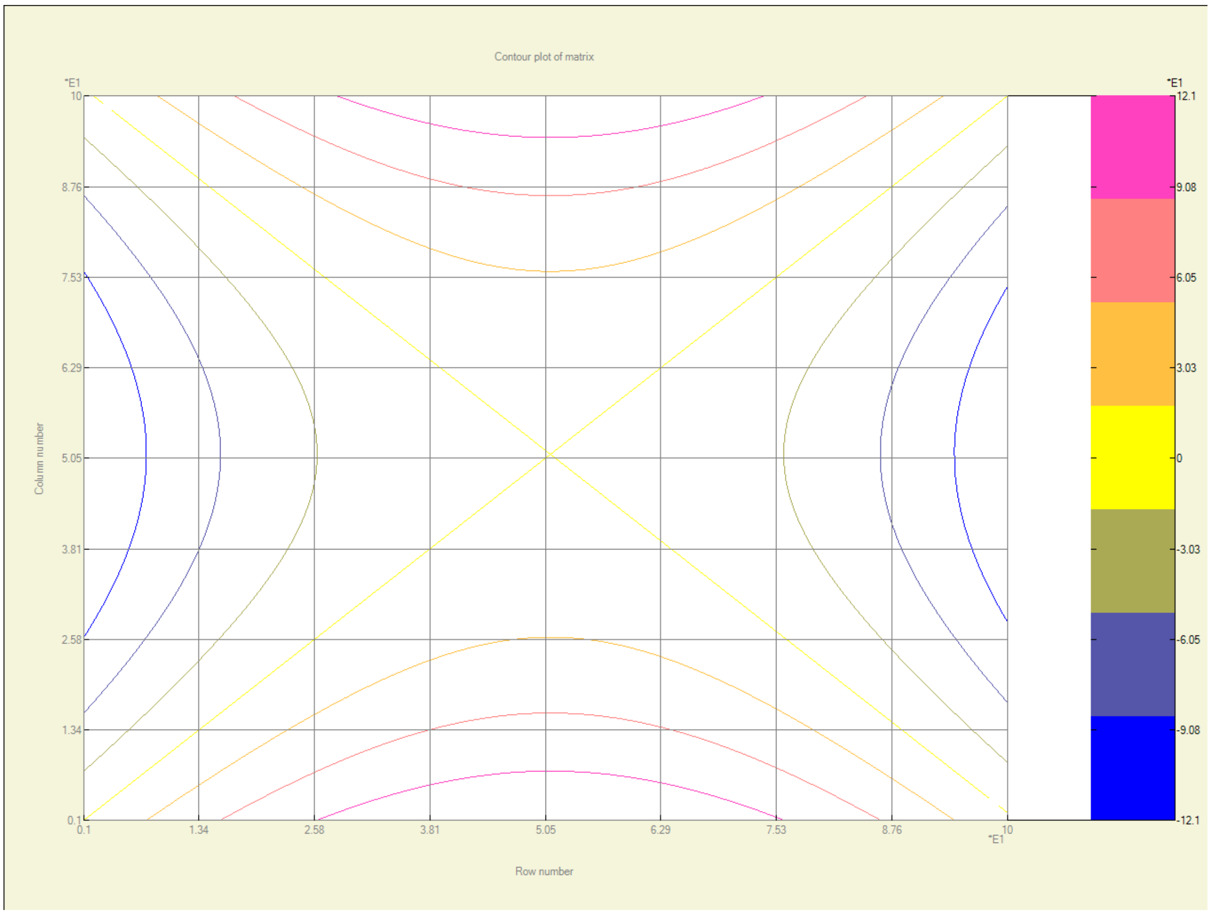


Figure 5 - Contour plot of Example 2

11.2 Constructor

11.2.1 **Graph()**

Creates a windows form with a plotting area and menus to

(i) Print the graph.
(ii) Adjust the range of the axes.
(iii) Save the figure in various file formats.

The data can be zoomed into by clicking in the graph region and scrolling the mouse wheel forward. Then when zoomed use the bottom and left scroll bars to pan the viewpoint. Scroll backwards to revert the graph to the starting view.

11.3 Methods

11.3.1 **Histogram(Vector $\mathbf{v}$)**

11.3.2 **Histogram(Vector $\mathbf{v}$, int $\mathbf{N}$)**

11.3.3 **Histogram(Vector $\mathbf{v}$, Vector $\mathbf{bins}$)**

Plots a histogram of the elements of a vector between the minimum and the maximum values in the vector - see Figure 6. The number of bins in the histogram is either 10, N or the number of elements in $bins$. The third routine allows the user to specify the threshold values for the bins. Otherwise they are deduced proportionately from the data.

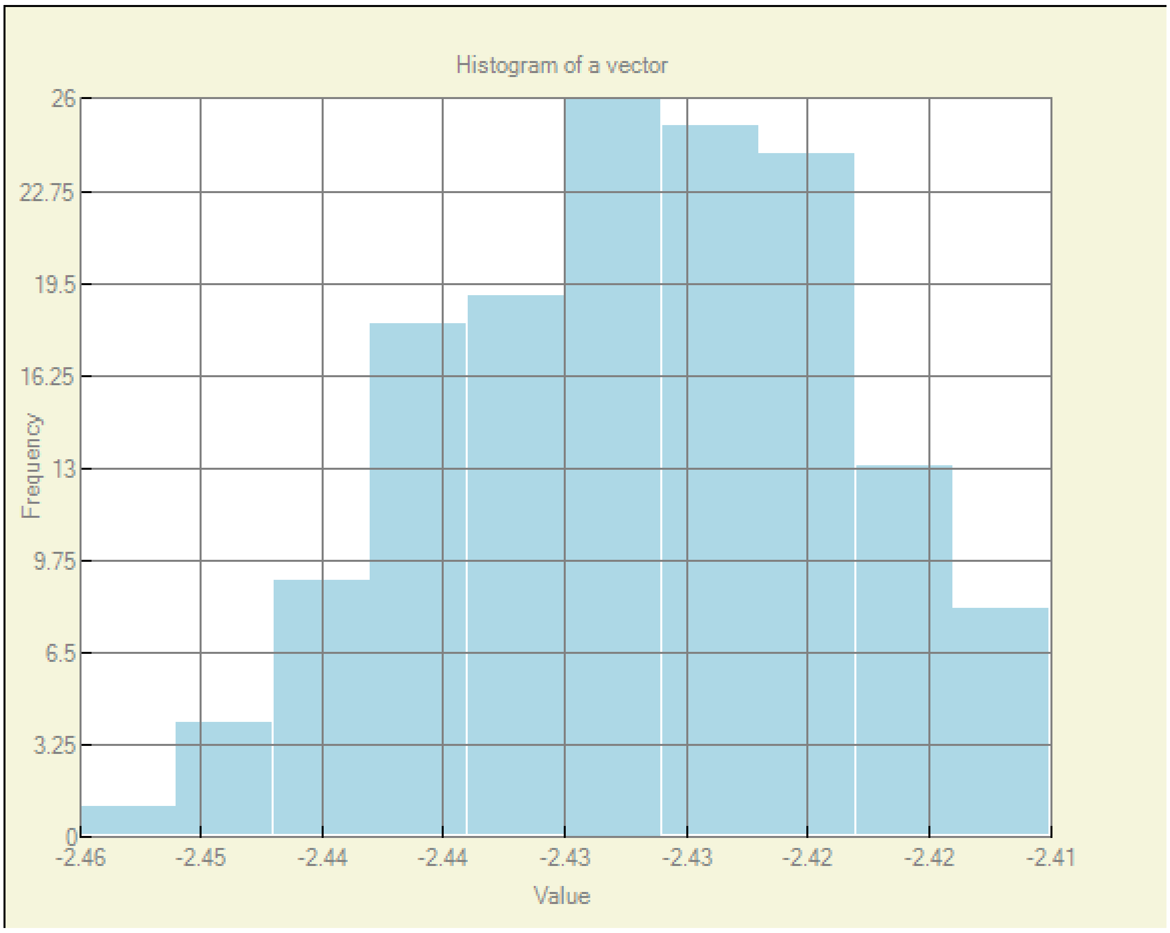


Figure 6 - Typical Histogram.

C# example:

```
Graph gr = new Graph();
double max = data.GetColumnVector(0).Max;
double min = data.GetColumnVector(0).Min;
int N = 10;
Vector bins = new Vector(N);
for (int i = 0; i < N; i++)
{
    bins[i] = min + (Convert.ToDouble(i) / N) * (max - min);
}
gr.Histogram(data.GetColumnVector(0), bins);
```

11.3.4 **Plot(Vector** *v* **)**

Plots the elements of vector *v* against their index number.

11.3.5 **Plot(Vector** *v* **,String** *colour* **,String** *style* **,int** *weight* **)**

Plots the elements of vector *v* against their index number. In addition the colour style and weight of the series may be specified.

colour may be one of: “Black”, “Red”, “Blue”, “Yellow”, “Green”, “Cyan”, “Magenta”.

style may be one of: “Solid”, “Dash”, “DashDot”, “DashDotDot”, “Dot”.

weight is a number between 1 and 10 giving the width in pixels of the line.

11.3.6 **Plot(Vector v,Vector w)**

Plots vector w against v. The two vectors must be of the same dimension, otherwise an error is thrown.

11.3.7 **Plot(Vector v,Vector w, String *colour* ,String *style* ,int *weight*)**

Plots vector w against v. The two vectors must be of the same dimension, otherwise an error is thrown. In addition the colour, style and weight of the series may be specified as described in 11.3.5.

11.3.8 **Plot(Matrix m)**

Plots a contour map of the elements of m with seven contour levels which are equally spaced between the minimum and maximum elements of m. The range of the x-axis is the number of columns of m and the range of the y-axis is the number of rows of m. The colour ranges from blue (low values) through yellow to red (high values). If a dimension of the matrix is less than 2 an error is thrown.

11.3.9 **Plot(Matrix m,Vector v)**

Plots a contour map of the elements of m with contour levels which are given by the elements of v. v must be monotonic increasing. The range of the x-axis is the number of columns of m and the range of the y-axis is the number of rows of m. If an element of v lies outside of the bounds of m, then no contour will be drawn. The colour ranges from blue (low values) through yellow to red (high values. If a dimension of the matrix is less than 2 an error is thrown.

11.3.10 **Plot(Matrix m, int Nc)**

Plots a contour map of the elements of m with Nc contour levels which are equally spaced between the minimum and maximum elements of m. The range of the x-axis is the number of columns of m and the range of the y-axis is the number of rows of m. The colour ranges from blue (low values) through yellow to red (high values). If a dimension of the matrix is less than 2 an error is thrown.

11.3.11 **Plot(Matrix m,Vector X ,Vector Y)**

Here X is a vector with dimension equal to the number of columns of m and Y is a vector with dimension equal to the number of rows of m. If this is not the case an error is thrown.

It is understood that:

(i) $m[i, j]$ is the value at x-coordinate $X[j]$ and y-coordinate $Y[i]$.

(ii) The elements of X and Y are both monotonically increasing and distinct - otherwise an error is thrown.

The range of the x and y axes are given by the ranges for X and Y respectively. Seven equally spaced contours are drawn. No grid lines are drawn.

11.3.12 **Plot (Matrix m,Vector X ,Vector Y ,Vector v)**

This is as described as in 11.3.11 except that the contours are given by the values of v, which must be monotonically increasing.

11.3.13 **Plot (Matrix m ,Vector X ,Vector Y ,int Nc)**

This is as described as in 11.3.11, except that the number of contours is as given by Nc.

11.3.14 **Plot (Matrix m, Vector X, Vector Y, int Nc, Color[] $colours$)**

This is as in 11.3.13, with a vector of Colors (System.Drawing.Color). There must be the same number of elements in $colours$ as is the value of Nc, otherwise an error is thrown.

11.3.15 **AddImage(String $name$, Bitmap $image$, int $blend$)**

Overlays the plot area with $image$, of name $name$**.** The blend parameter (from 0 to 100) describes the transparency of the overlay. 100 is 100% image (no plot visible), 0 is 100% plot, no image visible. 50 is a good compromise value.

11.3.16 **Size (int $width$,int $height$)**

Specifies the graph' form width and height in pixels. The default is 581 by 437.

11.3.17 **AddAnnotation(double x ,double y ,String s)**

Adds the string s to the plotting area at location (x, y). The coordinates x and y are relative to the data being plotted.

11.3.18 **AddTitle(String s)**

Adds a title to the plot as given by the string s.

11.3.19 **AddXLabel(String s)**

Adds an x-axis label as given by string s.

11.3.20 **AddYLabel(String s)**

Adds a y-axis label as given by string s.

11.3.21 **AddLegend(String s)**

Adds an item to the legend as given by the string s. Each series plotted is associated consecutively with a legend string until there are no more series to associate. The use of a legend within a contour plot will throw an error.

11.3.22 **AddXLlimits(double $left$,double $right$)**

Sets the left and right extremities of the x-axis data to display.

11.3.23 **AddYlimits(double $bottom$,double top)**

Sets the bottom and top extremities of the y-axis data to display.

11.3.24 **AddXInterval(double itr)**

Adds the grid points along the x axis.

11.3.25 **AddYInterval(double itr)**

Adds the grid points along the y axis.

11.3.26 **Fill (Boolean $fill$)**

When $fill$ is true the contours are filled with colour. Otherwise they are not. The default value is false.

11.3.27 **Fill (Boolean** ***fill,*** **Boolean** ***contours*****, int** ***fillSensitivity*****)**

This function affords more control over the colouring in process. When *fill* is set to true the contours are filled in with colour. Otherwise they are not. When *contours* is set to true the contours are draw in addition (in black). Otherwise they are not drawn. *fillSensitivity* is an integer, which is a power of 2, that controls the granularity of the colouring in process. The values permissible are 8, 16, 32, 64 and 128. Higher values result in increasingly fine detail yet take longer to do. A value of 16 is a good compromise.

11.3.28 **Cls()**

Clear the screen. The series, the legend(s), the title and the x and y axis labels are removed.

12 GraphExport class

This class provides rudimentary functions for creating a contour plot of a 2-dimensional vector, manipulating it in a palette and saving both it and the associated colour bar to file.

12.1 Constructor

12.1.1 **GraphExport(Matrix Z, Vector X, Vector Y, int Nc)**

Opens a windows form in preparation for plotting a contour plot of matrix Z, with number of contours Nc and axes X and Y.

12.2 Methods

12.2.1 **Plot()**

Constructs the contour plot.

12.2.2 **Fill(Boolean $fill$,Boolean $contours$, int $fillSensitivity$)**

This function behaves according to the description given in 11.3.27.

12.2.3 **AddImage(string $name$, Bitmap *image*, int $blend$)**

The description of this function is as in 11.3.15.

A coding example of the use of this function is shown below:

```
double a = 4.5;
double b = 6.0;
double c = 3.0;
double xmin = -a * 0.4;
double xmax = a * 0.4;
double ymin = -b * 0.3;
double ymax = b * 0.3;
double xspacing = 0.01;
double yspacing = 0.01;
int xnumber = Convert.ToInt32((xmax - xmin) / xspacing) + 1;
int ynumber = Convert.ToInt32((ymax - ymin) / yspacing) + 1;
Matrix z = new Matrix(ynumber, xnumber);
Vector xv = new Vector(xnumber);
Vector yv = new Vector(ynumber);

for (int i = 0; i < xnumber; i++)
{
    double x = xmin + i * xspacing;
    xv[i] = x;
    for (int j = 0; j < ynumber; j++)
    {
        double y = ymin + j * yspacing;
        yv[j] = y;
        //Get the elevation at the (i,j)th data point, corrresponding to the
        //(x,y)th value
        //Ellipsoid
        z[j, i] = c * Math.Sqrt(1.0 - (x * x / (a * a) + y * y / (b * b)));
    }
}

Vector levels = new Vector(20);
double minZ = z.Min;
double maxZ = z.Max;
for (int k = 0; k < 20; k++)
{
    levels[k] = minZ + (k + 1) * (maxZ - minZ) / (20 + 1);
}

GraphExport g = new GraphExport(z, xv, yv, 7);

Boolean fillIn = true;
Boolean contoursAsWell = true;
int fillSensitivity = 16;
g.Fill(fillIn, contoursAsWell, fillSensitivity);
g.Plot();
```

13 Graph3D class

This permits the display of 3D points, whilst allowing the user to adjust the elevation, ω and azimuth, θ, of the parallel projection view of the data. The coordinate system employed, and the definition of azimuth and elevation with respect to this system is shown in Figure 7.

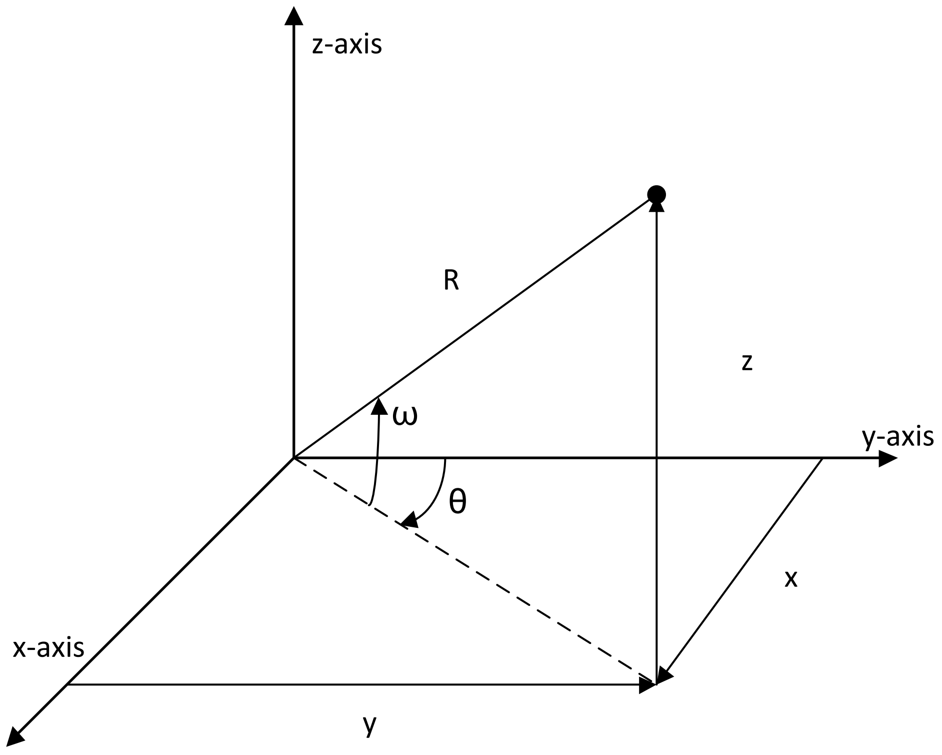
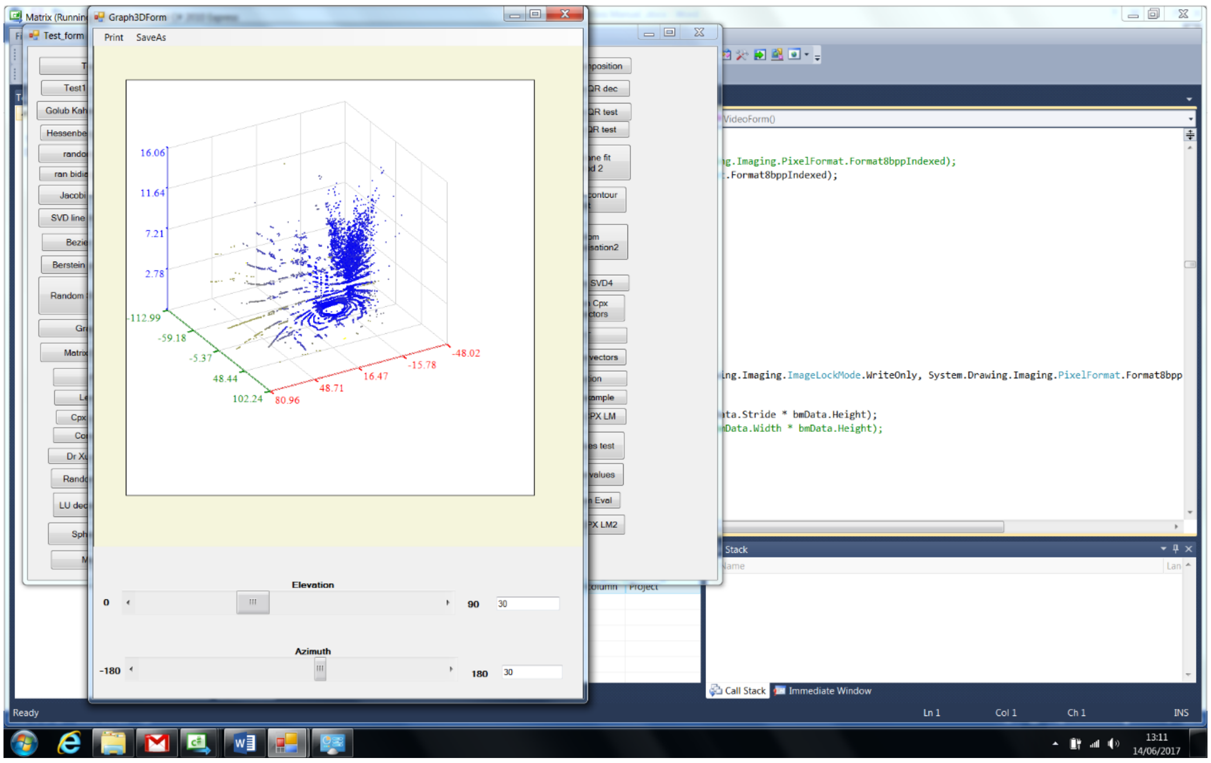


Figure 7 - Coordinate system employed for the 3D plotting software.

A screenshot of a typical view is shown in **Figure 8**. The x axis is coloured red, the y axis is coloured green and the z axis is coloured blue.



```
ng.Imaging.PixelFormat.Format8bppIndexed);
.Format8bppIndexed);

ing.Imaging.ImageLockMode.WriteOnly, System.Drawing.Imaging.PixelFormat.Format8bpp

ta.Stride * bmData.Height);
nData.Width * bmData.Height);
```

Figure 8 - 3D Graph screenshot

13.1 Constructor

13.1.1 **Graph3D()**

Creates the display region. It is necessary to call this constructor before doing anything else.

C# example:

```
Graph3D gr3D = new Graph3D();
```

13.2 Properties

13.2.1 **Boolean drawAxes**

Determines if grid, axes, axes tick marks and axes labels are drawn or not.

13.2.2 **Boolean drawGrid**

Determines if a grid is to be drawn or not. Only applies if drawAxes is set to true.

13.2.3 **int numberOfTicks**

Sets or gets the number of tick positions on the axes. Grid lines are drawn at the tick positions.

13.3 Methods

13.3.1 **void AddXAxisLimits(double** ***XMin*****, double** ***XMax)***

Sets the X axis limits. If these are provided then these limits will not be calculated from the data itself.

13.3.2 **void AddYAxisLimits(double** *YMin*, double *XMin*)

Sets the Y axis limits. If these are provided then these limits will not be calculated from the data itself.

13.3.3 **void AddZAxisLimits(double** *ZMin*, **double** *YMin*)

Sets the Z axis limits. If these are provided then these limits will not be calculated from the data itself.

13.3.4 **void BackgroundColour(Color** *backgroundColour***)**

Permits the background colour of the display region to be specified.

13.3.5 **Cls()**

Clears the screen.

13.3.6 **Plot(Vector** *x*, **Vector** *y*, **Vector** *z,* **Vector** s**)**

Plots points whose x, y and z coordinates are provided in the three vectors. S is a vector of intensity values from 1 to 100. The colour plotted ranges from blue (1) to yellow (100) These vectors must be of the same length, otherwise an error is thrown. The coordinates of any particular point are represented by the same index position in each vector – so that care must be taken to provide the data in this way.

C# example:

```
Graph3D gr3D = new Graph3D();
int L = iAdd - 1;
gr3D.Cls();
if (L > 0)
{

   gr3D.numberOfTicks = 4;
   gr3D.BackgroundColour(Color.White);
   gr3D.drawGrid = true;
   gr3D.Plot(X_.SubVector(0, L), Y_.SubVector(0, L), Z_.SubVector(0,
L), S_.SubVector(0, L));
}
```

13.3.7 **Plot(Vector x, Vector y, Vector z, Color c)**

Plots points whose x, y and z coordinates are provided in the three vectors. C is the color in which all points are to be plotted. The vectors must all be the same length, otherwise an error is thrown.

14 Video class

This class permits the display of a vector or matrix against time either in replay or in real time.

14.1 Constructor

14.1.1 Video()

Creates the display region. It is necessary to call this constructor before doing anything else.

C# example:

```
Video vid = new Video();
```

14.2 Methods

14.2.1 void AddSeries(Vector v)

Adds a series of points for subsequent replay. See 14.2.2.

14.2.2 Void PlayVideo(Vector t)

Plays a video of series previously added via the command AddSeries. The rate at which these are replayed is governed by the time vector supplied t.

C# example:

```
Video vid = new Video();

double f = 1.0; //Hz
for (int i = 0; i < noTimePoints; i++)
{
    t[i] = i;
    v1[i] = Math.Sin(2.0 * Math.PI * f * t[i] / 100.0);
    v2[i] = Math.Cos(2.0 * Math.PI * 2 * f * t[i] / 20.0);

    vid.AddMatrix(m);
}





vid.AddSeries(v1);
vid.AddSeries(v2);

vid.AddTitle("Test Vector Video");
vid.AddXLabel("Index");
vid.AddYLabel("Signal");
vid.AddLegend("Sinusoid 1");
vid.AddLegend("Sinusoid 2");

vid.PlayVideo(t);
```

14.2.3 **void AddMatrix(Matrix *m*)**

Add a matrix of values representing an grayscale image which is to be displayed later.

14.2.4 **void PlayMatrixVideo(Vector *t*)**

Plays a video of images previously added via the commandAddMatrix. The rate at which these are replayed is governed by the time vector supplied t.

C# example:

```
Video vid = new Video();
vid.SetContrast(200);

Random r = new Random();

double f = 1.0; //Hz
for (int i = 0; i < noTimePoints; i++)
{
     Matrix m = new Matrix(358, 128);
     t[i] = i;

     for (int j = 0; j < m.Nrows; j++)
     {
          for (int k = 0; k < m.Ncols; k++)
          {
              m[j, k] = r.Next(255);
          }
     }


     vid.AddMatrix(m);
}


vid.PlayMatrixVideo(t);
```

14.2.5 **void AddRealTimeMatrix(Matrix *m*)**

This function plots an image represented by a matrix in real time as it is provided. It is the responsibility of the caller to manage the rate of play, stopping, pausing etc.

C# example:

```
Video vid = new Video();
for (int i = 0; i < noTimePoints; i++)
{
    for (int j = 0; j < m.Nrows; j++)
    {
         for (int k = 0; k < m.Ncols; k++)
         {
             m[j, k] = r.Next(255);
         }
    }

    vid.AddRealTimeMatrix(m);

    System.Threading.Thread.Sleep(1000);
}
```

14.2.6 **void AddRealTimePoint(int** *index*, **double** *pt*, **double** *t*)

This function plots pt against t as it is supplied, allowing real time graphing of a signal.

The index value allows multiple signal values to be plotted in the same video.

C# example:

```
Video vid = new Video();

double f = 1.0; //Hz
for (int i = 0; i < noTimePoints; i++)
{
    t[i] = i;
    v1[i] = Math.Sin(2.0 * Math.PI * f * t[i] / 100.0);
    v2[i] = Math.Cos(2.0 * Math.PI * 2 * f * t[i] / 20.0);

    vid.AddMatrix(m);
}






vid.AddSeries(v1);
vid.AddSeries(v2);

vid.AddTitle("Test Vector Video");
vid.AddXLabel("Index");
vid.AddYLabel("Signal");
vid.AddLegend("Sinusoid 1");
vid.AddLegend("Sinusoid 2");
vid.AddYLimits(-1.0, 1.0);


for (int i = 0; i < 1000; i++)
{
      vid.AddRealTimePoint(0, v1[i], t[i]);
      vid.AddRealTimePoint(1, v2[i], t[i]);
      System.Threading.Thread.Sleep(1000);
}
```

14.2.7 **void AddRealTimeSeries(int** *index*, **Vector** *x*,**Vector** *y*)

Adds a series of points whose x and y coordinates are represented by x and y respectively to be plotted in real time. The use of the index parameter is special for this function. When it is set to zero the screen is cleared and plotting of series starts again. This is useful for plotting (for example) radar detections at each sweep, for autonomous car use.

C# example:

```
Video vid = new Video();

do
{
    if (targetsPerLine.Count == 0)
    {
         if (chkBlankscreenOnNoDetections.Checked)
         {
              vid.Cls();
         }
    }
    else
    {
        for (int i = 0; i < targetsPerLine.Count; i++)
        {

           vid.AddRealTimeSeries(i, bearingsPerLine[i], targetsPerLine[i]);

           vid.AddAnnotation(bearingsPerLine[i][0], targetsPerLine[i][0],
targetsPerLine[i].Dim.ToString());
           vid.AddLegend("Target" + (i + 1).ToString());
        }
    }

} while(t < 1000);
```

14.2.8 **void AddRealTimeSeries(int** *index*, **Vector** *x*, **Vector** *y*, **String** *color*, **String** *style*, **int** *weight***)**

Adds a series of points whose x and y coordinates are represented by x and y respectively to be plotted in real time. The use of the index parameter is special for this function. When it is set to zero the screen is cleared and plotting of series starts again. This is useful for plotting (for example) radar detections at each sweep, for autonomous car use.

In addition the colour, style and weight of the line may be specified.

The colour may be one of the following strings:

- Black
- Red
- Blue
- Yellow
- Green
- Cyan
- Magenta
- White

The line style may be one of the following (the default is solid):

- Point – in this case individual points are not connected. In all of the following cases they are connected.
- Solid
- Dash
- DashDot
- DashDotDot
- Dot

C# example:

```
String color;
String style = "Point";
int weight = 2;
if (cbBackgroundColour.Text == "White")
{
     color = "Black";
}
else
{
     color = "White";
}

vid.DisplayLegends(false);

if (targetsPerFrame.Count == 0)
{
     vid.Cls();
}
else
{
     int iAdd = 0;
     for (int i = 0; i < targetsPerFrame.Count; i++)
     {
           if (targetsPerFrame[i].detections != null)
           {
                if (targetsPerFrame[i].detections.Count > 0)
                {
                     Vector vy = new Vector(targetsPerFrame[i].detections.Count);
                     Vector vx = new Vector(targetsPerFrame[i].detections.Count);
                     for (int j = 0; j < vy.Dim; j++)
                     {
                          vx[j] = targetsPerFrame[i].detections[j].X;
                          vy[j] = targetsPerFrame[i].detections[j].Y;
                     }
                     vid.AddRealTimeSeries(iAdd, vx, vy, color, style, weight);

                     iAdd++;
                 }
             }

         }
     }
```

14.2.9 **void AddRealTimeMatrix(Matrix *m*)**

Adds a matrix representing an image to be plotted in real time.

C# example:

```
Video vid = new Video();
vid.SetContrast(0);


for (int i = 0; i < noTimePoints; i++)
{

     for (int j = 0; j < m.Nrows; j++)
     {
         for (int k = 0; k < m.Ncols; k++)
         {
             m[j, k] = r.Next(255);
         }
     }

     vid.AddRealTimeMatrix(m);

     System.Threading.Thread.Sleep(1000);

}
```

14.2.10 **void AddTitle(String *title*)**

Adds a title to the graphic.

14.2.11 **void AddXLabel(String** *xlabel)*

Adds an x axis label to the graphic.

14.2.12 **void AddXLimits(double *left*, double *right*)**

Imposes limits for the X axis.

14.2.13 **void AddYLimits(double** *top*, **double** *bottom*)

Imposes limits for the Y axis.

14.2.14 **void AddXInterval(double** *interval***)**

Sets the interval for the labelling on the x axis.

14.2.15 **void AddYInterval(double** *interval*)

Sets the interval for the labelling on the y axis.

14.2.16 **void AddLegend(String** *s)*

Adds a legend represented by the string s.

Each series plotted is associated consecutively with a legend string until there are no more series to associate.

14.2.17 **void AddAnnotation(double** *x,* **double** *y* **, String** *ann)*

This adds the string ann at the location specified by x and y. The location is with respect to the data.

14.2.18 **Void AddAnnotation(double** *x*, **double** *y*, **String** *ann*, **String** *colour***)**

Adds an annotation of the specified colour. *Colour* may be as specified in 14.2.8.

14.2.19 **Void Cls()**

Clears the video form.

14.2.20 **Void DisplayLegends(Boolean** *display*)

If *display* is set to false, legends are inhibited, otherwise they are displayed.

C# example:

```
vid.DisplayLegends(false);
```

14.2.21 **Void SetBackgroundColour(String** *colour*)

Sets the background colour. *Colour* may be any of the strings listed in 14.2.8.

14.2.22 **void SetContrast(byte** *contrast)*

Allows a number to be entered which determines the contrast of the greyscale image plotted via PlayMatrixVideo or AddRealTimeMatrix. The value is between 0 and 255. 0 gives rise to a darkened image (the plot of the maximum value of the data would be black) and 255 yields a brightened image (the plot of the minimum value of the data would be white).

See 14.2.9 for an example.

15 Delegate types

15.1.1 **delegate Vector NelderMeanFit(Vector *w*, Vector *beta*)**

A delegate function used in Nelder-Mean simplex method minimisation. See 5.4.15.

15.1.2 **delegate Vector NonlinearFit(Vector** *w* **, Vector** *beta* **)**

A delegate function used in non linear fitting. See 5.4.16.

15.1.3 **delegate double NonlinearFit2(Vector** *w* **, Vector** *beta* **)**

A delegate function used in non-linear fitting. See 5.4.17.

15.1.4 **delegate ComplexVector ComplexNonlinearFit(ComplexVector** *w* **, ComplexVector** *beta* **)**

A delegate function used in non linear fitting. See 7.4.11.

15.1.5 **delegate Complex ComplexNonlinearFit2(ComplexVector** *w* **, ComplexVector** *beta* **)**

A delegate function used in non linear fitting. See 7.4.12.

17 References

[1] Matrix Computations (2nd Edition), Gene H Golub, Charles F. Van Loan, The John Hopkins University Press, 1989.

[2] Numerical Analysis, 4th Edition, Richard L. Burdon, J. Douglas Faires, PWS-KENT Publishing Company, 1989.

[3] Restart Procedures for the Conjugate Gradient Method, M J D Powell, Mathematical Programming 12 (1977), p241-254.

[4] A Derivation of Conjugate Gradients, E M L Beale, Numerical Methods for Non Linear Optimisation, Academic Press (1972), F A Lootsma (Ed.), p39-43.

18 Contact

Paul D. Foy – paulfoy@mathematicalservices.co.uk