NAG
Nearest Correlation Matrix Solutions
ncm_nag

November 3rd, 2016

1 The Numerical Algorithms Group

- 1970 – "Nottingham Algorithms Group" - University project.
- 2016 – Mark 26 of the NAG Libraries.
- As well as Numerical libraries, we also provide:
  - Fortran compiler
  - HPC software engineering services
  - Consultancy work for bespoke application development
- Customers from:
  - Finance
  - Oil and Gas
  - Major Research organizations, etc.

1.1 Why Libraries are Important

- NAG frees the user from worrying about numerical computation which is difficult to do accurately.
- Problems of:
  - Stability
  - Underflow
  - Overflow
  - Condition
- Importance of testing and error analysis & bounds.
- Performance vital too, but after accuracy.

1.2 Products and Environments

- NAG Libraries can be used with:
- Fortran
- C/C++
- C#/.NET
- Java
- **Python**

- Matlab
- Excel
- R
- SAS
- Maple
- ... etc.

1.3 NAG Library Contents

- Over 1700 routines, fully documented with examples.
- Advice on choosing the right algorithm / routine.
  - Root Finding
  - Summation of Series
  - Quadrature
  - Ordinary Differential Equations
  - Partial Differential Equations
  - Numerical Differentiation
  - Integral Equations
  - Mesh Generation
  - Interpolation
  - Curve and Surface Fitting
  - Optimization
  - Approximations of Special Functions
  - Option Pricing
  - Dense Linear Algebra
  - Sparse Linear Algebra
  - Correlation and Regression Analysis
  - Multivariate Methods
  - Analysis of Variance
  - Random Number Generators
  - Univariate Estimation
  - Nonparametric Statistics
  - Smoothing in Statistics
  - Contingency Table Analysis
  - Survival Analysis
  - Time Series Analysis
  - Operations Research

2 Correlation Matrices

- An n-by-n matrix is a Correlation Matrix if:
  - it is symmetric
  - it has ones on the diagonal
  - its eigenvalues are nonnegative (positive semidefinite)

$$Ax = \lambda x, \qquad x \neq 0$$

- The element in the ith row and jth column is the correlation between the ith and jth variables. Stocks, for example.

2.1 Empirical Correlation Matrices

- Empirical correlation matrices are often **not mathematically true** due to inconsistent or missing data.

- Thus we are required to find a true correlation matrix, where our input, G, is an approximate correlation matrix.
- In particular we seek the *nearest* correlation matrix, in most cases.

2.2 Computing Correlation Matrices

- The vector p_i, the ith column of a matrix, P, holds the m observations of the ith variable, of which there are n. $\bar{p}_i$ is the sample mean.

$$S_{ij} = \frac{1}{m-1}(p_i - \bar{p}_i)^T(p_j - \bar{p}_j)$$

- S is a covariance matrix, with s_{ij} the covariance between variables i and j:
- R is a correlation matrix, given by:

$$D_S^{1/2} = \text{diag}(s_{11}^{-1/2}, s_{22}^{-1/2}, \ldots, s_{nn}^{-1/2})$$

$$R = D_S^{1/2} S D_S^{1/2}$$

2.3 Approximate Correlation Matrices

- Now, what if we don't have all observations for each variable?
- We compute each covariance with observations that are available for *both* the *i*th and *j*th variable.
- For example NAG routine **g02bb**.
- We then compute the correlation matrix as before.

3 Missing Stock Price Example

- Prices for 8 stocks on the first working day of 10 consecutive months.

	Stock A	Stock B	Stock C	Stock D	Stock E	Stock F	Stock G	Stock H
Month 1	59.875	42.734	47.938	60.359	54.016	69.625	61.500	62.125
Month 2	53.188	49.000	39.500		34.750		83.000	44.500
Month 3	55.750	50.000	38.938		30.188		70.875	29.938
Month 4	65.500	51.063	45.563	69.313	48.250	62.375	85.250	
Month 5	69.938	47.000	52.313	71.016		59.359	61.188	48.219
Month 6	61.500	44.188	53.438	57.000	35.313	55.813	51.500	62.188
Month 7	59.230	48.210	62.190	61.390	54.310	70.170	61.750	91.080
Month 8	61.230	48.700	60.300	68.580	61.250	70.340		
Month 9	52.900	52.690	54.230		68.170	70.600	57.870	88.640
Month 10	57.370	59.040	59.870	62.090	61.620	66.470	65.370	85.840

- So our $P = [p_1, p_2, \ldots, p_n]$ is:

$$P = \begin{bmatrix} 59.875 & 42.734 & 47.938 & 60.359 & 54.016 & 69.625 & 61.500 & 62.125 \\ 53.188 & 49.000 & 39.500 & \text{NaN} & 34.750 & \text{NaN} & 83.000 & 44.500 \\ 55.750 & 50.000 & 38.938 & \text{NaN} & 30.188 & \text{NaN} & 70.875 & 29.938 \\ 65.500 & 51.063 & 45.563 & 69.313 & 48.250 & 62.375 & 85.250 & \text{NaN} \\ 69.938 & 47.000 & 52.313 & 71.016 & \text{NaN} & 59.359 & 61.188 & 48.219 \\ 61.500 & 44.188 & 53.438 & 57.000 & 35.313 & 55.813 & 51.500 & 62.188 \\ 59.230 & 48.210 & 62.190 & 61.390 & 54.310 & 70.170 & 61.750 & 91.080 \\ 61.230 & 48.700 & 60.300 & 68.580 & 61.250 & 70.340 & \text{NaN} & \text{NaN} \\ 52.900 & 52.690 & 54.230 & \text{NaN} & 68.170 & 70.600 & 57.870 & 88.640 \\ 57.370 & 59.040 & 59.870 & 62.090 & 61.620 & 66.470 & 65.370 & 85.840 \end{bmatrix}.$$

- And to compute the covariance between the 3rd and 4th variables:

$$\begin{aligned} v_1^T &= [47.938, 45.563, 52.313, 53.438, 62.190, 60.300, 59.870] \\ v_2^T &= [60.359, 69.313, 71.016, 57.000, 61.390, 68.580, 62.090] \\ S_{3,4} &= \frac{1}{6}(v_1 - \bar{v}_1)^T(v_2 - \bar{v}_2) \end{aligned}$$

3.1 Import required modules and set print options

```
In [5]: import numpy as np
        import nag4py.g02 as nag_g02
        import nag4py.util as nag_util
        import matplotlib.pyplot as plt
        "Plot inline."
        %matplotlib inline
        "Set the print precision."
        np.set_printoptions(precision=4)
```

3.2 Initialize our P matrix of observations

```
In [6]: "Define a 2-d array and use np.nan to set elements as NaNs."
        P = np.array([[59.875, 42.734, 47.938, 60.359, 54.016, 69.625, 61.500, 62.125],
                      [53.188, 49.000, 39.500, np.nan, 34.750, np.nan, 83.000, 44.500],
                      [55.750, 50.000, 38.938, np.nan, 30.188, np.nan, 70.875, 29.938],
                      [65.500, 51.063, 45.563, 69.313, 48.250, 62.375, 85.250, np.nan],
                      [69.938, 47.000, 52.313, 71.016, np.nan, 59.359, 61.188, 48.219],
                      [61.500, 44.188, 53.438, 57.000, 35.313, 55.813, 51.500, 62.188],
                      [59.230, 48.210, 62.190, 61.390, 54.310, 70.170, 61.750, 91.080],
                      [61.230, 48.700, 60.300, 68.580, 61.250, 70.340, np.nan, np.nan],
                      [52.900, 52.690, 54.230, np.nan, 68.170, 70.600, 57.870, 88.640],
                      [57.370, 59.040, 59.870, 62.090, 61.620, 66.470, 65.370, 85.840]])
```

3.3 Compute the covariance, ignoring missing values

```
In [7]: def cov_bar(P):
            """Returns an approximate sample covariance matrix"""
            "P.shape returns a tuple (m, n) that we unpack to m and n."
            m, n = P.shape
            "Initialize an n-by-n zero matrix."
            S = np.zeros((n, n))
            "i = 0, ..., n-1."
            for i in range(0, n):
                "Take the ith column."
```

```
            xi = P[:, i]
            "j = 0, ..., i"
            for j in range(0, i+1):
                "Take the jth column, where j <= i."
                xj = P[:, j]
                "Set mask such that all NaNs are True."
                notp = np.isnan(xi) | np.isnan(xj)
                "Apply the mask to xi"
                xim = np.ma.masked_array(xi, mask=notp)
                "Apply the mask to xj"
                xjm = np.ma.masked_array(xj, mask=notp)
                S[i, j] = np.ma.dot(xim - np.mean(xim), xjm - np.mean(xjm))
                "Take the sum over ~notp to normalize."
                S[i, j] = 1.0 / (sum(~notp) - 1) * S[i, j]
                S[j, i] = S[i, j]
        return S
```

```
In [8]: def cor_bar(P):
            """Returns an approximate sample correlation matrix"""
            S = cov_bar(P)
            D = np.diag(1.0 / np.sqrt(np.diag(S)))
            "This is will only work in Python3"
            return D @ S @ D
```

3.4 Compute the *approximate* correlation matrix

```
In [9]: G = cor_bar(P)
        print(G)
```

```
[[ 1.     -0.325   0.1881  0.576   0.0064 -0.6111 -0.0724 -0.1589]
 [-0.325   1.      0.2048  0.2436  0.4058  0.273   0.2869  0.4241]
 [ 0.1881  0.2048  1.     -0.1325  0.7658  0.2765 -0.6172  0.9006]
 [ 0.576   0.2436 -0.1325  1.      0.3041  0.0126  0.6452 -0.321 ]
 [ 0.0064  0.4058  0.7658  0.3041  1.      0.6652 -0.3293  0.9939]
 [-0.6111  0.273   0.2765  0.0126  0.6652  1.      0.0492  0.5964]
 [-0.0724  0.2869 -0.6172  0.6452 -0.3293  0.0492  1.     -0.3983]
 [-0.1589  0.4241  0.9006 -0.321   0.9939  0.5964 -0.3983  1.    ]]
```

3.5 Compute the eigenvalues of our (indefinite) G.

```
In [10]: print("Sorted eigenvalues of G {}".format(np.sort(np.linalg.eig(G)[0])))
```

```
Sorted eigenvalues of G [-0.2498 -0.016   0.0895  0.2192  0.7072  1.7534  1.9611  3.5355]
```

4 Nearest Correlation Matrices

- We seek to solve:

$$\min \frac{1}{2}\|G - X\|_F^2 = \min \frac{1}{2}\sum_{i=1}^{n}\sum_{i=1}^{n}|G(i,j) - X(i,j)|^2$$

- In order to find X, a true correlation matrix, where G is an approximate correlation matrix.
- Algorithm by Qi and Sun (2006), applies an inexact Newton method to a dual (unconstrained) formulation of this problem.

- Improvements suggested by Borsdorf and Higham (2010 MSc).
- It is globally and quadratic (fast!) convergent.
- This is implemented in NAG routine **g02aa**.

4.1 Using g02aa to compute the nearest correlation matrix in the Frobenius norm

```
In [12]: "Set up the call to the NAG routine."
         order = nag_util.Nag_RowMajor
         Gflat = G.flatten()
         n = G.shape[0]
         pdg = n
         errtol = 0.0
         maxits = 0
         maxit = 0
         Xflat = np.empty_like(Gflat)
         pdx = n
         "Output arguments as 1 element arrays."
         itr = np.array([0])
         feval = np.array([0])
         nrmgrd = np.array([0.0])
         "Set noisy fail, error message printed but program not stopped."
         fail = nag_util.noisy_fail()
         "Call g02aa."
         nag_g02.g02aac(order, Gflat, pdg, n, errtol, maxits,
                        maxit, Xflat, pdx, itr, feval, nrmgrd, fail)
```

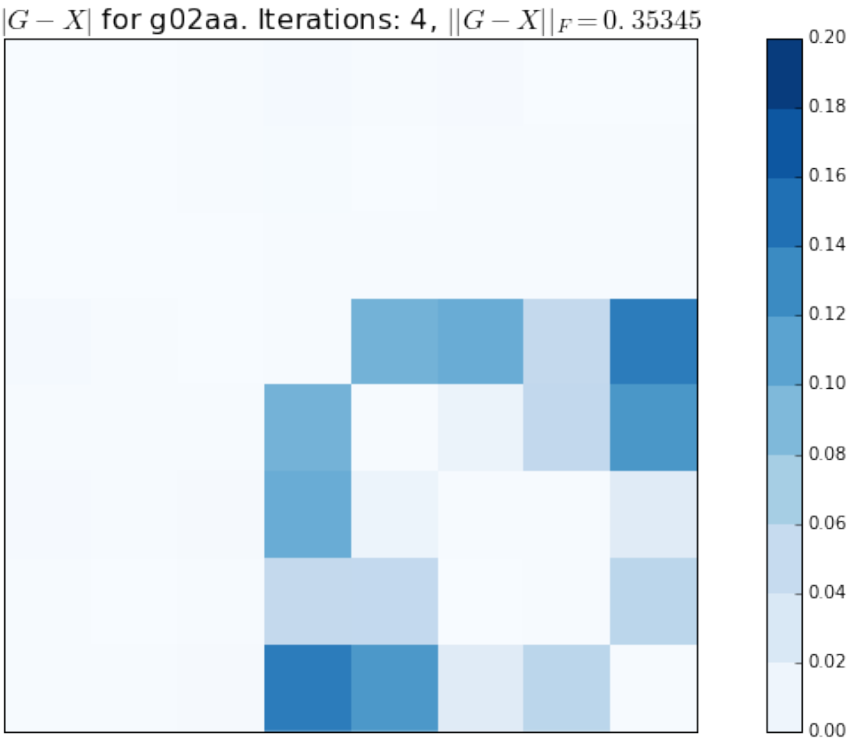
```
In [13]: "Unflatten X to have the same shape as G for comparison."
         X = np.reshape(Xflat, G.shape)
         print(X)
```

```
[[ 1.     -0.3112  0.1889  0.5396  0.0268 -0.5925 -0.0621 -0.1921]
 [-0.3112  1.      0.205   0.2265  0.4148  0.2822  0.2915  0.4088]
 [ 0.1889  0.205   1.     -0.1468  0.788   0.2727 -0.6085  0.8802]
 [ 0.5396  0.2265 -0.1468  1.      0.2137  0.0015  0.6069 -0.2208]
 [ 0.0268  0.4148  0.788   0.2137  1.      0.658  -0.2812  0.8762]
 [-0.5925  0.2822  0.2727  0.0015  0.658   1.      0.0479  0.5932]
 [-0.0621  0.2915 -0.6085  0.6069 -0.2812  0.0479  1.     -0.447 ]
 [-0.1921  0.4088  0.8802 -0.2208  0.8762  0.5932 -0.447   1.    ]]
```

```
In [14]: print("Sorted eigenvalues of X [{0}]".format(''.join(
                     ['{:.4f} '.format(x) for x in np.sort(np.linalg.eig(X)[0])])))
```

```
Sorted eigenvalues of X [-0.0000 -0.0000 0.0380 0.1731 0.6894 1.7117 1.9217 3.4661 ]
```

```
In [25]: "Plot the difference between G and X."
         fig1, ax1 = plt.subplots(figsize=(14, 7))
         cax1 = ax1.imshow(abs(X-G), interpolation='none', cmap=plt.cm.Blues,
                           vmin=0, vmax=0.2)
         cbar = fig1.colorbar(cax1, ticks = np.linspace(0.0, 0.2, 11, endpoint=True),
                              boundaries=np.linspace(0.0, 0.2, 11, endpoint=True))
         cbar.set_clim([0, 0.2])
         ax1.tick_params(axis='both', which='both',
                         bottom='off', top='off', left='off', right='off',
                         labelbottom='off', labelleft='off')
         ax1.set_title(
             r'$|G-X|$ for g02aa. Iterations: {0}, $||G-X||_F = {1:.5f}$'.format(itr[0],
                                                    np.linalg.norm(X-G)), fontsize=16)
         plt.show()
```



5 Weighting rows and columns of elements

- However, we note that for Stocks A to C we have a complete set of observations.

$$P = \begin{bmatrix} 59.875 & 42.734 & 47.938 & 60.359 & 54.016 & 69.625 & 61.500 & 62.125 \\ 53.188 & 49.000 & 39.500 & \text{NaN} & 34.750 & \text{NaN} & 83.000 & 44.500 \\ 55.750 & 50.000 & 38.938 & \text{NaN} & 30.188 & \text{NaN} & 70.875 & 29.938 \\ 65.500 & 51.063 & 45.563 & 69.313 & 48.250 & 62.375 & 85.250 & \text{NaN} \\ 69.938 & 47.000 & 52.313 & 71.016 & \text{NaN} & 59.359 & 61.188 & 48.219 \\ 61.500 & 44.188 & 53.438 & 57.000 & 35.313 & 55.813 & 51.500 & 62.188 \\ 59.230 & 48.210 & 62.190 & 61.390 & 54.310 & 70.170 & 61.750 & 91.080 \\ 61.230 & 48.700 & 60.300 & 68.580 & 61.250 & 70.340 & \text{NaN} & \text{NaN} \\ 52.900 & 52.690 & 54.230 & \text{NaN} & 68.170 & 70.600 & 57.870 & 88.640 \\ 57.370 & 59.040 & 59.870 & 62.090 & 61.620 & 66.470 & 65.370 & 85.840 \end{bmatrix}.$$

- Perhaps we wish to preserve part of the correlation matrix?
- We could solve the *weighted* problem, NAG routine **g02ab**.

$$\|W^{\frac{1}{2}}(G - X)W^{\frac{1}{2}}\|_F$$

- Where W is a diagonal matrix.
- We can also force the resulting matrix to be positive definite.

5.1 Use g02ab to compute the nearest correlation matrix with row and column weighting

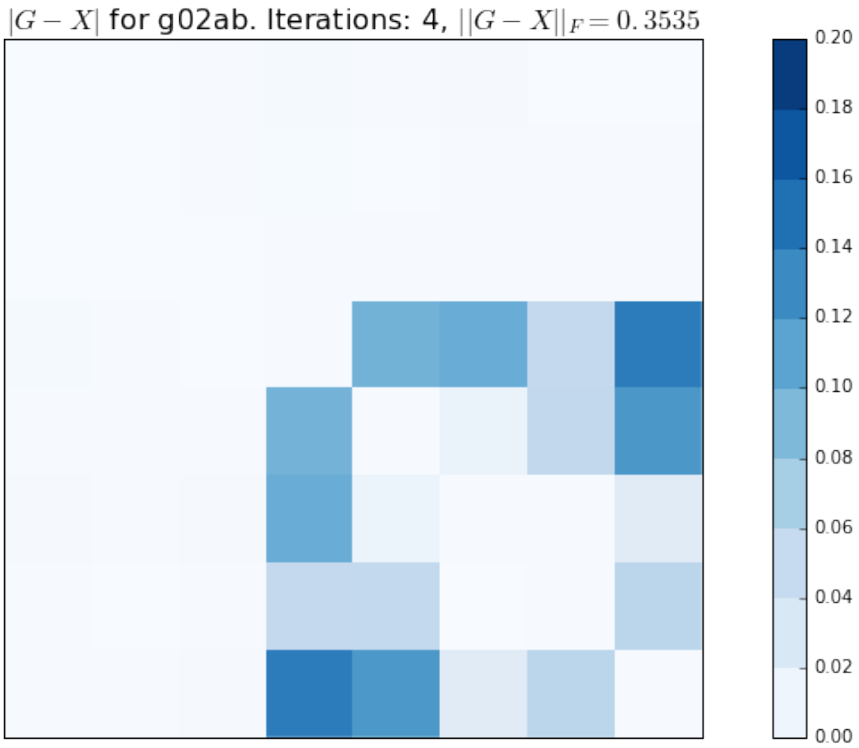
```
In [26]: "Define an array of weights."
         W = np.array([10, 10, 10, 1, 1, 1, 1, 1], dtype = np.float64)

In [27]: order = nag_util.Nag_RowMajor
         Gflat = G.flatten()
         n = G.shape[0]
         opt = nag_util.Nag_Both
         "Set the smallest eigenvalue."
         alpha = 0.001
         pdg = n
         errtol = 0.0
         maxits = 0
         maxit = 0
         Xflat = np.empty_like(Gflat)
         pdx = n
         itr = np.array([0])
         feval = np.array([0])
         nrmgrd = np.array([0.0])
         fail = nag_util.noisy_fail()
         nag_g02.g02abc(order, Gflat, pdg, n, opt, alpha, W, errtol, maxits,
                        maxit, Xflat, pdx, itr, feval, nrmgrd, fail)

In [28]: X = np.reshape(Xflat, G.shape)
         print(X)
```

```
[[ 1.     -0.3247  0.1879  0.5733  0.0067 -0.609  -0.0721 -0.1596]
 [-0.3247  1.      0.2046  0.2423  0.4056  0.2735  0.2867  0.4232]
 [ 0.1879  0.2046  1.     -0.1321  0.7654  0.2756 -0.6164  0.8995]
 [ 0.5733  0.2423 -0.1321  1.      0.2083 -0.0889  0.5948 -0.1804]
 [ 0.0067  0.4056  0.7654  0.2083  1.      0.655  -0.2777  0.8748]
 [-0.609   0.2735  0.2756 -0.0889  0.655   1.      0.049   0.574 ]
 [-0.0721  0.2867 -0.6164  0.5948 -0.2777  0.049   1.     -0.4545]
 [-0.1596  0.4232  0.8995 -0.1804  0.8748  0.574  -0.4545  1.    ]]
```

```
In [29]: print("Sorted eigenvalues of X [{0}]".format(''.join(
                     ['{:.4f} '.format(x) for x in np.sort(np.linalg.eig(X)[0])])))
```

```
Sorted eigenvalues of X [0.0020 0.0020 0.0315 0.1655 0.6767 1.7708 1.8901 3.4614 ]
```

```
In [30]: fig1, ax1 = plt.subplots(figsize=(14, 7))
         cax1 = ax1.imshow(abs(X-G), interpolation='none', cmap=plt.cm.Blues, vmin=0,
                           vmax=0.2)
         cbar = fig1.colorbar(cax1, ticks = np.linspace(0.0, 0.2, 11, endpoint=True),
                              boundaries=np.linspace(0.0, 0.2, 11, endpoint=True))
         cbar.set_clim([0, 0.2])
         ax1.tick_params(axis='both', which='both',
                         bottom='off', top='off', left='off', right='off',
                         labelbottom='off', labelleft='off')
         ax1.set_title(
             r'$|G-X|$ for g02ab. Iterations: {0}, $||G-X||_F = {1:.4f}$'.format(itr[0],
                                                    p.linalg.norm(X-G)), fontsize=16)
         plt.show()
```



6 Weighting Individual Elements

- Would it be better to be able to *weight individual elements* in our approximate matrix?
- In our example the top left 3-by-3 block of exact correlations, perhaps.
- Element-wise weighting means we wish to find the minimum of:

$$\|H \circ (G - X)\|_F$$

- So individually: $h_{ij} \times (g_{ij} - x_{ij})$
- However, this is a more "difficult" problem, and more computationally expensive.
- Implemented in the NAG routine **g02aj**.

6.1 Use g02aj to compute the nearest correlation matrix with element-wise weighting

```
In [37]: "Set up our matrix of weights."
         H = np.ones([n, n])
         H[0:3, 0:3] = 100
         print(H);
```

```
[[ 100.  100.  100.    1.    1.    1.    1.    1.]
 [ 100.  100.  100.    1.    1.    1.    1.    1.]
 [ 100.  100.  100.    1.    1.    1.    1.    1.]
 [   1.    1.    1.    1.    1.    1.    1.    1.]
```

```
 [   1.    1.    1.    1.    1.    1.    1.    1.]
 [   1.    1.    1.    1.    1.    1.    1.    1.]
 [   1.    1.    1.    1.    1.    1.    1.    1.]
 [   1.    1.    1.    1.    1.    1.    1.    1.]]
```

In [38]:
```
Gflat = G.flatten()
n = G.shape[0]
alpha = 0.001
Hflat = H.flatten()
pdg = n
pdh = n
errtol = 0.0
maxits = 0
maxit = 500
Xflat = np.empty_like(Gflat)
pdx = n
itr = np.array([0])
norm = np.array([0.0])
fail = nag_util.noisy_fail()
nag_g02.g02ajc(Gflat, pdg, n, alpha, Hflat, pdh, errtol,
               maxit, Xflat, pdx, itr, norm, fail)
```

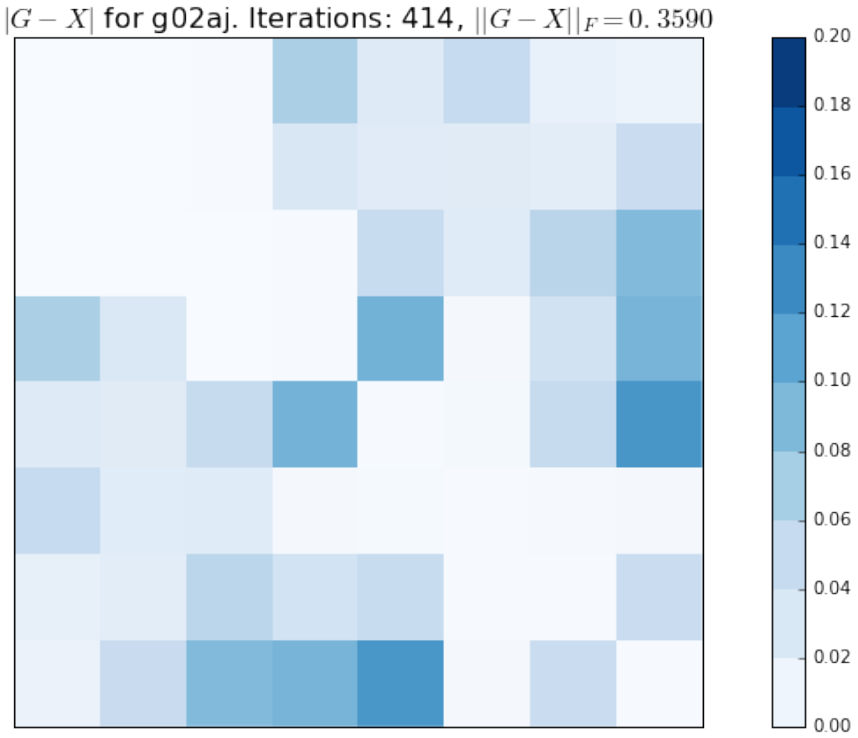
In [39]:
```
X = np.reshape(Xflat, G.shape)
print(X)
```

```
[[ 1.     -0.3247  0.188   0.5091  0.0306 -0.5611 -0.0569 -0.1701]
 [-0.3247  1.      0.2047  0.2146  0.3837  0.2504  0.2676  0.3781]
 [ 0.188   0.2047  1.     -0.1331  0.7175  0.2534 -0.5607  0.812 ]
 [ 0.5091  0.2146 -0.1331  1.      0.2081  0.0145  0.6079 -0.2272]
 [ 0.0306  0.3837  0.7175  0.2081  1.      0.6622 -0.282   0.8732]
 [-0.5611  0.2504  0.2534  0.0145  0.6622  1.      0.0482  0.5982]
 [-0.0569  0.2676 -0.5607  0.6079 -0.282   0.0482  1.     -0.4438]
 [-0.1701  0.3781  0.812  -0.2272  0.8732  0.5982 -0.4438  1.    ]]
```

In [40]:
```
print("Sorted eigenvalues of X [{0}]".format(''.join(
                    ['{:.4f} '.format(x) for x in np.sort(np.linalg.eig(X)[0])])))
```

```
Sorted eigenvalues of X [0.0020 0.0129 0.1162 0.2104 0.7367 1.6700 1.8892 3.3627 ]
```

In [41]:
```
fig1, ax1 = plt.subplots(figsize=(14, 7))
cax1 = ax1.imshow(abs(X-G), interpolation='none', cmap=plt.cm.Blues, vmin=0,
                  vmax=0.2)
cbar = fig1.colorbar(cax1, ticks = np.linspace(0.0, 0.2, 11, endpoint=True),
                     boundaries=np.linspace(0.0, 0.2, 11, endpoint=True))
cbar.set_clim([0, 0.2])
ax1.tick_params(axis='both', which='both',
                bottom='off', top='off', left='off', right='off',
                labelbottom='off', labelleft='off')
ax1.set_title(
    r'$|G-X|$ for g02aj. Iterations: {0}, $||G-X||_F = {1:.4f}$'.format(itr[0],
                                                np.linalg.norm(X-G)), fontsize=16)
plt.show()
```



7 Fixing a Block of Elements

- We probably really wish to *fix* our leading block of true correlations, so it does not change at all.
- New at Mark 25 was the NAG routine **g02an**.
- This routine fixes a leading block, which we require to be positive definite.
- We apply the *shrinking algorithm* of Higham, Strabic and Sego. The approach is **not** computationally expensive.
- What we find is the smallest α, such that X is a true correlation matrix:

$$X = \alpha \begin{pmatrix} G_{11} & 0 \\ 0 & I \end{pmatrix} + (1-\alpha)G, \qquad G = \begin{pmatrix} G_{11} & G_{12} \\ G_{12}^T & G_{22} \end{pmatrix}$$

- G_{11} is the leading k-by-k block of the approximate correlation matrix that we wish to fix.
- α is in the interval $[0, 1]$.

7.1 Use g02an to compute the nearest correlation matrix with fixed leading block

```
In [42]: Gflat = G.flatten()
         n = G.shape[0]
         pdg = n
         "Set the size of block we are fixing."
         k = 3
```

```
errtol = 0.0
eigtol = 0.0
maxits = 0
maxit = 500
Xflat = np.empty_like(Gflat)
pdx = n
alpha = np.array([0.0])
itr = np.array([0])
eigmin = np.array([0.0])
norm = np.array([0.0])
fail = nag_util.noisy_fail()
nag_g02.g02anc(Gflat, pdg, n, k, errtol, eigtol, Xflat, pdx,
               alpha, itr, eigmin, norm, fail)
```

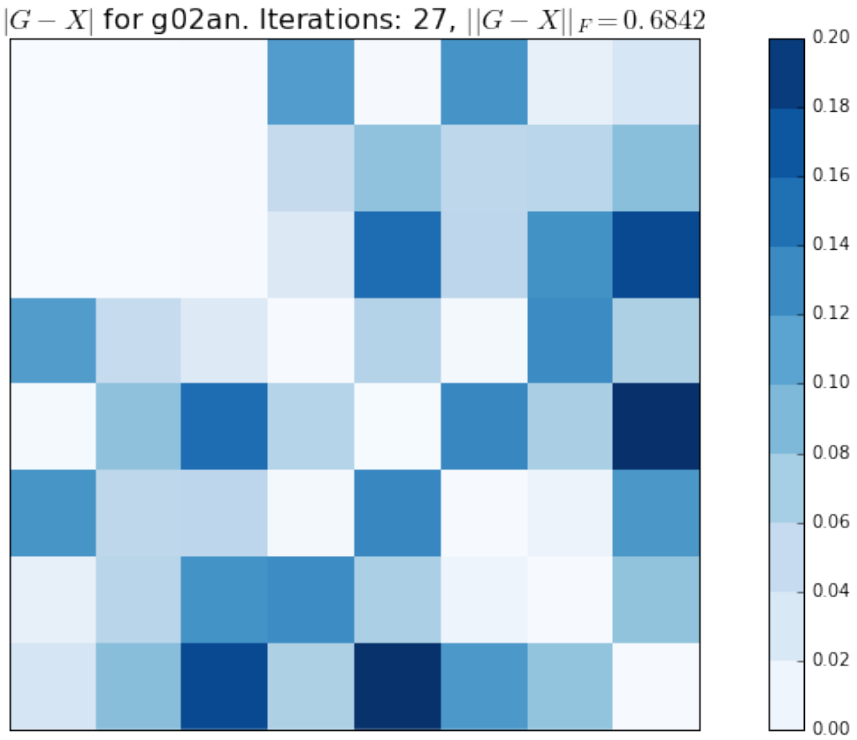
In [43]:
```
X = np.reshape(Xflat, G.shape)
print(X)
```

```
[[ 1.     -0.325   0.1881  0.4606  0.0051 -0.4887 -0.0579 -0.1271]
 [-0.325   1.      0.2048  0.1948  0.3245  0.2183  0.2294  0.3391]
 [ 0.1881  0.2048  1.     -0.106   0.6124  0.2211 -0.4936  0.7202]
 [ 0.4606  0.1948 -0.106   1.      0.2432  0.0101  0.516  -0.2567]
 [ 0.0051  0.3245  0.6124  0.2432  1.      0.532  -0.2634  0.7949]
 [-0.4887  0.2183  0.2211  0.0101  0.532   1.      0.0393  0.4769]
 [-0.0579  0.2294 -0.4936  0.516  -0.2634  0.0393  1.     -0.3185]
 [-0.1271  0.3391  0.7202 -0.2567  0.7949  0.4769 -0.3185  1.    ]]
```

In [44]:
```
print("Sorted eigenvalues of X [{0}]".format(''.join(
                ['{:.4f} '.format(x) for x in np.sort(np.linalg.eig(X)[0])])))
print("Value of alpha returned: {:.4f}".format(np.asscalar(alpha)))
```

```
Sorted eigenvalues of X [0.0000 0.1375 0.2744 0.3804 0.7768 1.6263 1.7689 3.0356 ]
Value of alpha returned: 0.2003
```

In [45]:
```
fig1, ax1 = plt.subplots(figsize=(14, 7))
cax1 = ax1.imshow(abs(X-G), interpolation='none', cmap=plt.cm.Blues, vmin=0,
                  vmax=0.2)
cbar = fig1.colorbar(cax1, ticks = np.linspace(0.0, 0.2, 11, endpoint=True),
                     boundaries=np.linspace(0.0, 0.2, 11, endpoint=True))
cbar.set_clim([0, 0.2])
ax1.tick_params(axis='both', which='both',
                bottom='off', top='off', left='off', right='off',
                labelbottom='off', labelleft='off')
ax1.set_title(
    r'$|G-X|$ for g02an. Iterations: {0}, $||G-X||_F = {1:.4f}$'.format(itr[0],
                                            np.linalg.norm(X-G)), fontsize=16)
plt.show()
```



8 Fixing Arbitrary Elements

- In Mark 26 of the NAG libraries we will have the new routine **g02ap**.
- This routine fixes arbitrary elements by finding the smallest α, such that X is a true correlation matrix in:

$$X = \alpha T + (1 - \alpha)G, \quad T = H \circ G, \quad h_{ij} \in [0, 1]$$

- A "1" in H fixes corresponding elements in G.
- $0 < h_{ij} < 1$ weights corresponding element in G.
- α is again in the interval $[0, 1]$.

8.1 Alternating Projections

- First method proposed to solve our original problem, however, it is very slow.
- The idea is we alternate projecting onto two sets, which are:
  - the set of smeidefinite matrices (S1), and
  - matrices with unit diagonal (s2)
- We do this until we converge on a matrix with both properties.

8.2 Alternating Projections with Anderson Acceleration

- A new approach by Higham and Strabic uses *Anderson Acceleration*, and makes the method worthwhile.
- In particulary, we will be able to fix elements whilst finding the nearest true correlation matrix in the Frobenius norm.
- Our projections are now:
  - the set of (semi)definite matrices with some minimum eigenvalue, and
  - matrix with elements $G_{i,j}$ for some given indices i and j
- To appear in a future NAG Library.

9 Visit the Website

9.1 Library routines:

http://www.nag.com/nag-c-library

9.2 More on using NAG with Python:

http://www.nag.com/nag-library-python and
http://www.nag.com/content/downloads-nag-python-bindings-nag4py