Reprint from
The *Mathematica* Conference
June, 1992 Boston, MA

863.15.10.1992

Statistics

Emily Martin

1 Introduction

Most examples of statistical software are strong in established numerical procedures for data analysis, but weak in programming, graphics, symbolic computation, and new techniques that either have not yet found acceptance by statisticians or are not easily coded.

The tutorial examples presented here are meant to demonstate some of *Mathematica*'s strong points in the statistics field, especially programming flexibility, extensive graphics, symbolic computation, and an array of supporting packages that may not seem immediately applicable to statistics problems.

2 Examples

2.1 Parameterization of Distributions

The standard library of statistics packages includes several univariate distribution packages: `DiscreteDistributions``, `ContinuousDistributions``, and `NormalDistribution``. Each distribution is represented by a name and a set of parameters that do not evaluate but serve as arguments for other functions.

In the case of the multivariate normal distribution and the distribution of a quadratic form, it makes sense to consider a representation that evaluates to an object having the same head but a different parameterization, one more useful to other functions that take the distribution as an argument. For example, random number generation of a multivariate normal makes use of the Cholesky decomposition of the covariance matrix and the characteristic function of a quadratic form (based on a multivariate normal) makes use of the eigenvalues of a matrix derived from the normal covariance matrix and the quadratic form matrix. It seems preferable to do these matrix operations once and save the results for later use inside an object of the same head. Of course, this will work only if the new parameterization has a form different from that of the old.

Load the Cholesky decomposition function.

```
In[1]:= <<LinearAlgebra`Cholesky`
```

This reparameterizes the multivariate normal in terms of the mean vector and a lower triangular matrix l (represented as a triangular list) having the property that $l\,l^T$ gives the original covariance matrix.

```
In[2]:= MultivariateNormalDistribution[
         mu_?VectorQ, cov_?MatrixQ] :=
          Module[{l = Transpose[CholeskyDecomposition[cov]]},
           MultivariateNormalDistribution[mu,
           MapIndexed[Take[#1, #2[[1]]]&, l]]
          ] /; cov == Transpose[cov]
```

Load the definition of `NormalDistribution`.

```
In[3]:= <<Statistics`NormalDistribution`
```

Only n one-dimensional normal variates are required to produce one n-dimensional normal variate.

```
In[4]:= MultivariateNormalDistribution/:
          Random[MultivariateNormalDistribution[
         mu_?VectorQ, l_]] :=
            Module[{z = Table[Random[NormalDistribution[0, 1]],
         {Length[mu]}]},
         mu + MapIndexed[#1 . Take[z, #2[[1]]]&, l]
            ] /; Apply[And,
         MapIndexed[Length[#1] == #2[[1]] &, l]]
```

This gives two three-dimensional normal variates.

```
In[5]:= Table[N[Random[MultivariateNormalDistribution[{0, 1, 1},
          Table[1/(i + j - 1), {i, 3}, {j, 3}]]]], {2}]

Out[5]= {{-1.63994, 0.465138, 0.702986},
  {-0.598517, 0.397885, 0.510592}}
```

This reparameterizes the distribution of the quadratic form $x^T a\, x + b^T x + c$, where x is a multivariate normal with mean vector `mu` and covariance matrix `cov`, in terms of a vector of eigenvalues λ, another vector β, and a scalar α. More simple parameterizations are possible for special cases such as $\mu = \{0, \ldots, 0\}$ or $b = \{0, \ldots, 0\}$.

```
In[6]:= QuadraticFormDistribution[{a_?MatrixQ, b_?VectorQ, c_},
         {mu_?VectorQ, cov_?MatrixQ}] :=
          Module[{lambda, evecs, p,
                 u = CholeskyDecomposition[cov]},
            {lambda, evecs} = Eigensystem[u.a.Transpose[u]];
            evecs /= Sqrt[evecs[[1]].evecs[[1]]];
            QuadraticFormDistribution[lambda,
             evecs.(u.b + 2u.a.mu), mu.a.mu + b.mu + c]
          ] /; cov == Transpose[cov]
```

The characteristic function of the quadratic form is easily expressed in the reduced parameter set.

```
In[7]:= QuadraticFormDistribution/:
          CharacteristicFunction[QuadraticFormDistribution[
         lambda_?VectorQ, beta_?VectorQ, alpha_], t_] :=
           Module[{terms = Map[(1 - 2 t I #)&, lambda]},
            Exp[alpha t I - t^2/2 (beta^2) . (terms^(-1))] *
         Apply[Times, terms^(-1/2)]]
```

This gives the characteristic function of the quadratic form $8x_1^2 - 8x_1x_2 + 3x_2^2 - 2x_1 + x_2 + 6$, where $\{x_1, x_2\}$ is distributed according to a bivariate normal with mean $\{-1, 1\}$ and covariance matrix $\{\{1, 1\}, \{1, 2\}\}$. A statistic for classifying a length-2 feature vector according to one of two binormally distributed classes typically follows this distribution. The characteristic function can be used to calculate exact probabilities of error.

```
In[8]:= CharacteristicFunction[QuadraticFormDistribution[
         {{{8, -4}, {-4, 3}}, {-2, 1}, 6},
         {{-1, 1}, {{1,1}, {1,2}}}], t] // Simplify

                                  2           2
                             169 t         4 t
         Power[E, 28 I t - ---------- - ----------]
                           1 - 8 I t    1 - 4 I t
Out[8]= --------------------------------------------
            Sqrt[1 - 8 I t] Sqrt[1 - 4 I t]
```

2.2 Edgeworth Expansions

Mathematica provides the function `O` for representing order terms. However, an expression such as `O[`*n*`]^3` will result in a `SeriesData` object. Sometimes it is useful to manipulate expressions involving terms of different orders using monomials rather than `O` functions, especially if the expression is an equation to be submitted to `Solve`. The subject of Edgeworth expansions of statistical distributions provides some examples of this.

Under certain conditions on a density function (pdf), it may be expanded in a series of derivatives of the normal pdf, where the expansion coefficients are functions of the central moments and the k^{th} expansion term is the k^{th} Hermite polynomial multiplied by the normal density function. This expansion may be written as

$$\sum_{k=0}^{\infty} a_k\, H_k\left(x/\sqrt{2}\right)\ \phi(x)$$

where a_k is a function of the first k moments associated with the pdf, $H_k(x)$ is the *Mathematica* `HermiteH[`*k*`, `*x*`]` polynomial, and $\phi(x)$ denotes the standard normal density. One could use this expansion to do density estimation when estimates of the moments of a pdf are known. The problem with this expansion (sometimes referred to as *Gram-Charlier Type A*) is that the coefficients do not decrease in size with increasing k when the moments decrease with increasing k. Thus it is not possible to take a finite number of terms and neglect the remainder. An *Edgeworth* expansion is formally the same as this expansion although the individual terms are dependent on the cumulants of the pdf (related to the moments) and decrease with decreasing k if the cumulants do. The cumulants of some common sample statistics expressible as sums (*i.e.*, `Mean[`*data*`]`, `TrimmedMean[`*data*`]`, `Variance[`*data*`]`) behave in this manner, so while a sample statistic may be asymptotically normal with increasing sample size, an Edgeworth expansion can be useful for investigating how the distribution deviates from normality for small sample size.

The Edgeworth expansion of the density of a standardized variate may be written as

$$\exp\left[-\kappa_3 \frac{D^3}{3!} + \kappa_4 \frac{D^4}{4!} - \cdots\right] \phi(x)$$

where κ_k is the k^{th} cumulant and D denotes the derivative operator. A function to compute an Edgeworth expansion approximating the pdf of a distribution is given in the following package. This function makes use of an identity relating the k^{th} derivative of the standard normal density $\phi(x)$ and the k^{th} Hermite polynomial,

$$(-D)^k\ \phi(x) = H_k\left(x/\sqrt{2}\right)\ \phi(x)/\,2^{k/2}.$$

```
BeginPackage["Edgeworth`"]

EdgeworthSeries::usage =
"EdgeworthSeries[{k3, k4, ..., kr}, x] generates an Edgeworth series
approximation in x to the pdf of the standardized variate (y - c1)/Sqrt[c2],
where y is an asymptotically normal statistic based on a sample of size n,
having cumulants cr and where the cumulants of x are given by
kr = cr c2^(-r/2).  The expansion is to order n^(1-r/2) under the assumption
that cr is of order n^(1-r)."

Begin["`Private`"]

EdgeworthSeries[cumulant_List, x_] :=
  Module[{r = Length[cumulant] + 2, t, d, expansion},
        expansion = Collect[Series[
                        Exp[cumulant . Table[d^k t^(k-2)/k!, {k, 3, r}]],
                        {t, 0, r-2}], d] /. {t :> 1};
        Exp[-x^2/2]/Sqrt[2 Pi] (expansion //.
                {d^k_ :> HermiteH[k, x/Sqrt[2]]/Sqrt[2]^k // Expand})
  ]

End[]
EndPackage[]
```

The `Edgeworth`` package.

Although the cumulants are assumed to behave as order terms, the trick used in the `EdgeworthSeries` function is to temporarily multiply the cumulants by monomials in `t` to produce the expansion carried out to the desired order.

This loads the `Edgeworth`` package.

```
In[1]:= <<Edgeworth`
```

Next load the definition of `CentralMoment`.

```
In[2]:= <<Statistics`DescriptiveStatistics`
```

Create a set of asymmetrical heavy-tailed data.

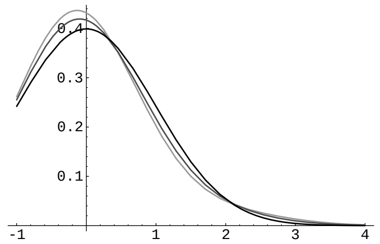
```
In[3]:= data = Table[(Log[Random[]])^2, {10000}];
```

This finds the standardized third and fourth cumulants of `data`. These cumulants account for skewness (asymmetry) and kurtosis (peak and tail behavior).

```
In[4]:= cumulants = Module[{sigma = StandardDeviation[data]},
            {CentralMoment[data, 3]/sigma^3,
              CentralMoment[data, 4]/sigma^4 - 3}]

Out[4]= {5.9178, 56.2735}
```

The cumulants associated with the sample mean's distribution are scaled versions of `cumulants`. A plot of the Edgeworth expansions of the distribution of the sample mean based on samples of size 4 and 5 indicates how the shape of the distribution of the sample mean approaches normality with increasing sample size.

```
In[5]:= Plot[Evaluate[
        {EdgeworthSeries[cumulants {1/4^2, 1/4^3}, x],
         EdgeworthSeries[cumulants {1/5^2, 1/5^3}, x],
         Exp[-x^2/2]/Sqrt[2 Pi]}], {x, -1, 4},
           PlotStyle -> {{GrayLevel[.6]}, {GrayLevel[.3]},
        {GrayLevel[0]}}]
```



Apart from using the Edgeworth expansion to graph an approximation to the pdf of a sample statistic, it can be used to derive a simple symbolic result relating the mode and the median of a sample statistic's distribution.

This defines `f` to be an Edgeworth approximation to the pdf of a standardized sample statistic (having zero mean `c[1]` and unit variance `c[2]`) expanded to include all terms of order $n^{\frac{-3}{2}}$ where n is the sample size.

```
In[6]:= f[x_] := EdgeworthSeries[{c[3], c[4], c[5]}, x]
```

The modes of a density function are located where the function achieves a local maximum. Here the derivative of the approximated pdf `f` is computed and the numerator of this expression is expanded about the mean, `x = 0`. Setting the result to zero expresses the constraint that `x` equals the mode of the distribution closest to the mean.

```
In[7]:= modeEQ = (Normal[Series[Numerator[Together[D[f[x], x]]],
            {x, 0, 2}]] == 0)

                                 3
Out[7]= -3240 c[3] + 4725 c[3]  +

                          2
     x (-6480 + 9450 c[3]  - 4050 c[4]) -

                           2
     4725 c[3] c[4] + x

                              3
      (6480 c[3] - 23625 c[3]  + 18900 c[3] c[4] -

        2430 c[5]) + 810 c[5] == 0
```

This is the first of the two solutions for `x` satisfying the mode equation. In this form it is difficult to identify the terms of order n^{-2}.

```
In[8]:= modeSOLN = Solve[modeEQ, x][[1,1,2]]

                                    2
Out[8]= (-((-192 + 280 c[3]  - 120 c[4]) /

                             3
          (48 c[3] - 175 c[3]  + 140 c[3] c[4] -

            18 c[5])) +

                                 2            4
        (8 Sqrt[576 - 528 c[3]  - 4655 c[3]  +

                      6                          2
            6125 c[3]  + 720 c[4] + 3990 c[3]  c[4] -

                       4                    2
            11025 c[3]  c[4] + 225 c[4]  +

                      2     2
            4900 c[3]  c[4]  - 720 c[3] c[5] +

                      3
            1680 c[3]  c[5] - 1470 c[3] c[4] c[5] +

                     2
            108 c[5] ]) /

                                3
         (-48 c[3] + 175 c[3]  - 140 c[3] c[4] +

           18 c[5])) / 8
```

Since the r^{th} cumulant is a term of order $n^{1-\frac{r}{2}}$ where n is the sample size, `c[r]` is replaced with `c[r] t^(r-2)`. The symbol `t` effectively represents $n^{\frac{-1}{2}}$. Once the dominant terms are found using a `Series` expansion, `t` is set to unity. The result is the mode up through terms of order n^{-2}.

```
In[9]:= modeAPPROX = (Normal[Series[modeSOLN /.
          {c[r_] :> c[r] t^(r-2)},
          {t, 0, 4}]] /. t->1) // Expand
```

```
                     3
        -c[3]    c[3]    5 c[3] c[4]    c[5]
Out[9]= ----- + ----- - ----------- + ----
          2        4         12          8
```

This defines `F` to be an approximation to the cumulative distribution function (cdf) of the statistic. Loading the package ``Calculus`Limit` `` enhances the capabilities of the built-in `Limit`.

```
In[10]:= (integral = Integrate[f[x], x];
          <<Calculus`Limit`;
          F[x_] := integral - Limit[integral,  x->-Infinity])
```

The median of a distribution function is located where the function is equal to one half. Here the approximated cdf `F` is expanded about the mean, `x = 0`. Setting this to one half expresses the constraint that `x` equals the median of the distribution.

```
In[11]:= medianEQ = (Normal[Series[F[x], {x, 0, 2}]] == 1/2)
```

```
         1                               2
Out[11]= - + (x  (6480 - 270 (5 c[3]  - 3 c[4]))) /
         2

    (6480 Sqrt[2 Pi]) +

     2                             3
   (x  (-3 (360 c[3] - 700 c[3]  + 675 c[3] c[4] -

           108 c[5]) -

                                  3
        (3 (360 c[3] - 175 c[3]  + 225 c[3] c[4] -

             54 c[5])) / 2)) / (6480 Sqrt[2 Pi]) +

                              3
   (360 c[3] - 175 c[3]  + 225 c[3] c[4] -

                                       1
     54 c[5]) / (2160 Sqrt[2 Pi]) ==  -
                                       2
```

This is the first of the two solutions for `x` satisfying the median equation.

```
In[12]:= medianSOLN = Solve[medianEQ, x][[1,1,2]]
```

```
Out[12]= (-(

                                  2
           -30240 + 6300 c[3]  - 3780 c[4]
       ----------------------------------------) \
                        3
       24 c[3] - 35 c[3]  + 35 c[3] c[4] - 6 c[5]

                                                   2
     + (21 Sqrt[40] Sqrt[51840 - 12960 c[3]  -

                     4             6
         14550 c[3]  + 6125 c[3]  + 12960 c[4] +

                     2                   4
         15300 c[3]  c[4] - 14000 c[3]  c[4] +

                   2            2     2
         810 c[4]  + 7875 c[3]  c[4]  -

                                          3
         3456 c[3] c[5] + 2940 c[3]  c[5] -

                                                2
         3240 c[3] c[4] c[5] + 324 c[5] ]) /

                          3
      (-24 c[3] + 35 c[3]  - 35 c[3] c[4] + 6 c[5]))\
    / 630
```

The technique used for finding the mode up through terms of order n^{-2} is used here to find the median.

```
In[13]:= medianAPPROX = (Normal[Series[medianSOLN /.
          {c[r_] :> c[r] t^(r-2)},
          {t, 0, 4}]] /. t->1) // Expand
```

```
                            3
         -c[3]    23 c[3]     c[3] c[4]    c[5]
Out[13]= ----- + -------- - --------- + ----
           6        432          12         40
```

From these approximations based on an assumption about the order of the cumulants of a sample statistic, one can conclude that the distance from the mean to the mode is approximately three times that from the mean to the median. A comment should be made about the solutions to `modeEQ` and `medianEQ` that were not pursued. `Series` expansions of these solutions in terms of `t` about `t = 0` reveal that these solutions grow with increasing sample size. Since the mode and median should approach the mean as the sample size increases, these solutions may be discarded.

■ 2.3 Nonlinear Fitting Diagnostics

Mathematica Version 2.1 of the standard package library includes the new package `Statistics`NonlinearFit`` for fitting data to nonlinear models using a least squares criterion. It is well known that in the case of linear models, parameter estimates found by least squares are unbiased, normally distributed, and have minimum variance. However, in the case of nonlinear models, least squares parameter estimates have these favorable properties only asymptotically with increasing sample size. To find parameter estimates that not only yield a small value for χ^2 (sum of squares), but also come close to having these properties for finite sample size, it makes sense to seek a model/data combination such that the solution locus behaves linearly in the neighborhood of the least squares solution. For some simple problems these ideas can be visualized using *Mathematica*'s parametric plotting. For realistic problems with large sample sizes and several parameters, *Mathematica* may be used to compute diagnostics that reveal the degree of linearity achieved by a least squares solution to a nonlinear fit.

This nonlinear model is sometimes called a *yield-density* model where y represents yield per plant, x plant density, $1/\alpha = \lim_{x \to 0} y$ "genetic potential" of a plant variety, and $1/\beta = \lim_{x \to \infty} x\,y$ "environmental potential".

```
In[1]:= y[x_] := (alpha + beta x)^(-1)
```

Create a set of three $\{x, y\}$ pairs for a specific instance of the model.

```
In[2]:= data = Map[{#, y[#] + .01 (Random[] - .5)}&,
        {1., 4., 7.}] /.
        {alpha -> 14., beta -> 3.}

Out[2]= {{1., 0.0543842}, {4., 0.0397645},
  {7., 0.0271704}}
```

This loads the package `Statistics`NonlinearFit``.

```
In[3]:= <<Statistics`NonlinearFit`
```

The default starting value of the parameters α and β is 1. The default method of fitting is Levenberg-Marquardt, a method of gradually shifting the search from steepest descent to quadratic minimization. In this case, the fit fails to converge.

```
In[4]:= NonlinearFit[data, y[x], {x}, {alpha, beta}]

NonlinearFit::lmpnocon:
   Warning: the values of the parameters given to
     NonlinearFit do not appear to have converged.
     The returned value may not be at the minimum.

Out[4]= {alpha -> 7.7259, beta -> 3.33611}
```

Here the fit is restarted from the previous solution.

```
In[5]:= NonlinearFit[data, y[x], {x},
        Map[{#, # /. %}&, {alpha, beta}]]

NonlinearFit::lmpnocon:
   Warning: the values of the parameters given to
     NonlinearFit do not appear to have converged.
     The returned value may not be at the minimum.

Out[5]= {alpha -> 15.0313, beta -> 2.80452}
```

A third fit yields convergence.

```
In[6]:= lsSOLN = NonlinearFit[data, y[x], {x},
        Map[{#, # /. %}&, {alpha, beta}]]

Out[6]= {alpha -> 15.4933, beta -> 2.73862}
```

The χ^2 (`chisquare`) merit function is smallest for the third fit.

```
In[7]:= (chisquare = Apply[Plus,
           Map[(#[[2]] - y[#[[1]]])^2&, data]];
         Map[(chisquare /. #)&, {%4, %5, lsSOLN}])

                                  -6             -6
Out[7]= {0.00138138, 8.43401 10  , 6.85138 10  }
```

The steepest descent method is able to find a good solution from the default parameter starting values. The χ^2 for this solution is greater than that for the last two Levenberg-Marquardt fits.

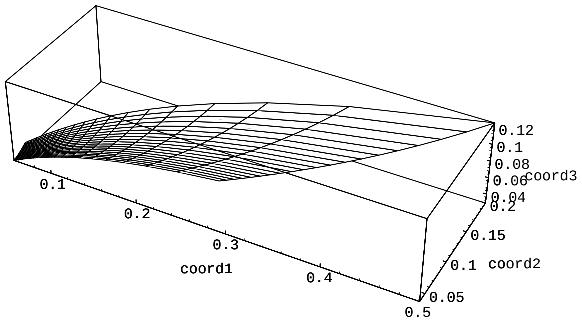
```
In[8]:= chisquare /. NonlinearFit[data,
        y[x], {x}, {alpha, beta},
        Method -> FindMinimum]

Out[8]= 0.000011016
```

The solution locus is the set of all points having the form $\{\alpha + 1.\beta, \alpha + 4.\beta, \alpha + 7.\beta\}$.

```
In[9]:= ({xval, yval} = Transpose[data];
        locus = Map[y[#]&, xval])

                    1                    1
Out[9]= {----------------, ----------------,
         alpha + 1. beta  alpha + 4. beta

          1
   ----------------}
   alpha + 7. beta
```

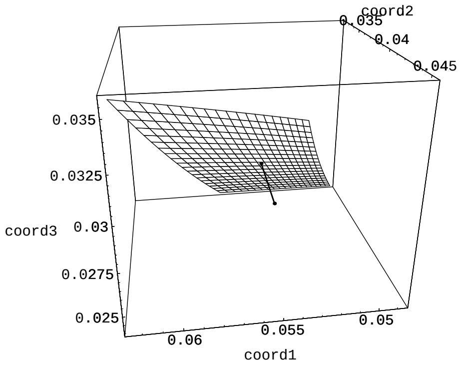
Plotting the solution locus shows how the "shape" of the locus and the "spacing" of lines of constant α and β varies between the starting values $\{1, 1\}$ and the final `lsSOLN` values.

```
In[10]:= ParametricPlot3D[Evaluate[locus],
          {alpha, 1., lsSOLN[[1,2]]}, {beta, 1., lsSOLN[[2,2]]},
          AxesLabel -> {"coord1", "coord2", "coord3   "},
          PlotRange -> Transpose[{locus /. lsSOLN,
          locus /. {alpha -> 1., beta -> 1.}}],
          Shading -> False]
```



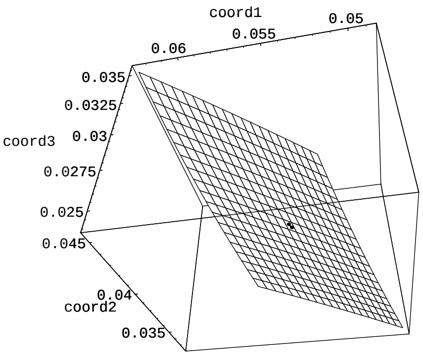
The least squares solution `lsSOLN` is that point on the solution locus nearest the observation point y. Adding the segment between y and `lsSOLN` to the graph and restricting the parameter ranges makes evaluating the locus behavior near `lsSOLN` somewhat easier.

```
In[11]:= (locusplot = ParametricPlot3D[Evaluate[locus],
      {alpha, 14., 17.}, {beta, 2., 3.5},
      AxesLabel -> {"coord1", "coord2", "coord3   "},
      Shading -> False,
      DisplayFunction -> Identity];
      Show[locusplot,
          Graphics3D[{Point[yval], Point[locus /. lsSOLN],
      Line[{yval, locus /. lsSOLN}]}],
             ViewPoint->{0.3, 2.0, 0.9},
      DisplayFunction -> $DisplayFunction])
```



Viewing the solution locus from the direction of y reveals that in the neighborhood of `lsSOLN`, the locus appears planar and the parameter lines approximately straight, parallel, and equispaced.

```
In[12]:= Show[locusplot,
         Graphics3D[{Point[yval], Point[locus /. lsSOLN],
            Line[{yval, locus /. lsSOLN}]}],
         ViewPoint-> yval +
      2(#/Sqrt[#.#])&[yval-(locus/.lsSOLN)],
         DisplayFunction -> $DisplayFunction]
```



Bates and Watts (1980) developed curvature diagnostics quantifying the degree to which a solution locus is planar and has equispaced parameter lines in the neighborhood of a least squares solution. The idea is to consider an arbitrary straight line in parameter space and the behavior of the associated curve on the solution locus. Constant velocity along the line in parameter space yields an acceleration along the curve on the locus. The p tangential components of the acceleration measure the equispacedness of parameter lines on the surface and the $n - p$ normal components of the acceleration measure the planarity of the locus. The resulting measures, termed maximum relative *parameter-effects* and *instrinsic* curvatures are invariant under changes of scale in the response so they can be used to compare least squares solutions across data sets. It turns out that in practice parameter-effects curvature almost always exceeds intrinsic curvature, suggesting that parameterization has a larger effect on the reliability of a least squares solution to a nonlinear fit than the form of the model.

The algorithm for computing these measures is easily written in *Mathematica* and takes advantage of calculus and linear algebra functions. The curvatures may be used to test the "planar" and "uniform coordinates" assumptions using the *F*-ratio distribution provided in the `Statistics`NormalDistribution`` package.

```
BeginPackage["CurvatureDiagnostics`", "OrthogonalComplement`"]

CurvatureDiagnostics::usage =
"CurvatureDiagnostics[data, model, vars, params, (opts)] computes the
maximum relative instrinsic and parameter effects curvatures associated with
the least squares solution of a nonlinear fit."

IntrinsicCurvature::usage =
"IntrinsicCurvature is used to identify the maximum relative intrinsic curvature
associated with the least squares solution of a nonlinear fit.  This
measure is intrinsic to the model and is unaffected by reparameterization."

ParameterCurvature::usage =
"ParameterCurvature is used to identify the maximum relative parameter effects
curvature associated with the least squares solution of a nonlinear fit.
This measure is affected by reparameterization of the model."

Sigma::usage =
"Sigma is an estimate of the standard deviation of the measurement error."

Begin["`Private`"]

CurvatureDiagnostics[data:_?(VectorQ[N[#], NumberQ]&),
        model_, vars_, params_, soln_, opts___] :=
  CurvatureDiagnostics[Transpose[Append[(Evaluate[Table[#,
        {Length[vars]}]]&) [Range[Length[data]]], data]],
        model, vars, params, soln, opts]

CurvatureDiagnostics[data:_?(MatrixQ[N[#], NumberQ]&),
        model_, vars:{__Symbol}, params:{__Symbol}, soln:{__Rule}, opts___] :=
  Module[{result =
          iCurvatureDiagnostics[data, model, vars, params, soln, opts]},
        result /; result =!= $Failed]

iCurvatureDiagnostics[data_, model_, vars_, params_, soln_, opts___] :=
  Module[{p = Length[params], n = Length[data], tol, maxiter, sigma,
          datasubs, radius, d1 = Map[D[model, #]&, params],
          V1, V2, q, rinv, accel, PEcurve, INcurve},
    {tol, maxiter, sigma} = {Tolerance, MaxIterations, Sigma} /.
        Options[CurvatureDiagnostics] /. {opts};
    datasubs[x_] := Thread[Rule[vars, Drop[x, -1]]];
    If[sigma === Automatic,
        sigma = Sqrt[Apply[Plus,
          Map[(Last[#] - model /. soln /. datasubs[#])&, data]^2]/(n-p)]];
    radius = sigma Sqrt[p];
    V1 = radius^(-1) Map[(d1 /. soln /. datasubs[#])&, data];
    V2 = radius^(-1) Map[((Map[D[d1, #]&, params]) /. soln /.
              datasubs[#])&, data];
    If[Head[qrd = QRDecomposition[V1]] === List,
        {q, rinv} = {qrd[[1]], Inverse[qrd[[2]]]},        (* {p x n, p x p} *)
        Message[CurvatureDiagnostics::qrdfail]; Return[$Failed]];
    q = Join[q, OrthogonalComplement[q]];                 (* n x n *)
    accel = q.Map[(Transpose[rinv].#.rinv)&, V2];         (* n x p x p *)
    INcurve = maxcurve[Drop[accel, p], tol, maxiter];
    If[INcurve === Null,
        Message[CurvatureDiagnostics::infail, tol, maxiter]; Return[$Failed]];
    PEcurve = maxcurve[Take[accel, p], tol, maxiter];
    If[PEcurve === Null,
        Message[CurvatureDiagnostics::pefail, tol, maxiter]; Return[$Failed]];
    {IntrinsicCurvature -> INcurve, ParameterCurvature -> PEcurve}
  ]
```

```
(* maxcurve returns the maximum curvature of an acceleration matrix when
   convergence occurs or Null if the maximum # of iterations is exceeded. *)

maxcurve[A_, tol_, maxiter_] :=
  Module[{dir, grad, Ad, dAd, thresh = 1. - tol},
    (* A is a m x p x p tensor, where p = # of parameters, m = d.o.f. *)

    (* initialize parameter space direction vector *)
    dir = Append[Table[0, {Length[ A[[1,1]] ] - 1}], 1];

    Do[  (* set grad to [dirT A dir]T [A dir] and normalize *)
         Ad = Map[(#.dir)&, A];      (* m x p matrix *)
         dAd = Map[(dir.#)&, Ad];    (* m vector *)
         grad = dAd . Ad;             (* p vector *)
         grad /= Sqrt[grad.grad];    (* normalize *)

         (* at convergence, dir points along grad *)
         If[ Abs[grad.dir] <= thresh,
           dir = 3 grad + dir;       (* update dir *)
           dir /= Sqrt[dir.dir],     (* normalize *)
           Return[Sqrt[dAd.dAd]] ],

         {maxiter}]
  ]

Options[CurvatureDiagnostics] =
        {Tolerance -> N[10^(-6)], MaxIterations -> 20, Sigma -> Automatic}

CurvatureDiagnostics::qrdfail = "QRDecomposition failed."
CurvatureDiagnostics::pefail =
"Maximum parameter effects curvature search failed to converge to `` tolerance
 within `` iterations."
CurvatureDiagnostics::infail =
"Maximum intrinsic curvature search failed to converge to `` tolerance
 within `` iterations."

(*  Ref: "Relative Curvature Measures of Nonlinearity",
            Bates and Watts, J.R.Stat.Soc.B (1980), 42, No.1, pp.1-25. *)

End[]
EndPackage[]
```

The `CurvatureDiagnostics`` package.

This loads the `CurvatureDiagnostics`` package.

```
In[13]:= <<CurvatureDiagnostics`
```

The intrinsic and parameter-effects curvatures are returned in the form of rules.

```
In[14]:= curvatures = CurvatureDiagnostics[data, y[x], {x},
         {alpha, beta}, lsSOLN]

Out[14]= {IntrinsicCurvature -> 0.0553821,
  ParameterCurvature -> 0.166518}
```

Load the definition of `FRatioDistribution`.

```
In[15]:= <<Statistics`NormalDistribution`
```

This gives the 5% quantile of the F-ratio distribution with $n - p = 1$ numerator degrees of freedom and $p = 2$ denominator degrees of freedom, denoted $F_{1,2,.05}$. Each maximum relative curvature statistic is a ratio of the estimate of standard deviation in the measurement error (here having $n - p$ degrees of freedom) to an estimate of the minimum curvature radius of the p-dimensional solution locus. Thus both the intrinsic and parameter effects curvatures may be tested against the square root of $F_{1,2,.05}$.

```
In[16]:= fstat = Quantile[FRatioDistribution[1, 2], .05]

Out[16]= 0.00501253
```

We find that the assumption of planarity may be accepted and the assumption of uniform coordinates rejected at the 5% significance level.

```
In[17]:= {IntrinsicCurvature < Sqrt[fstat],
         ParameterCurvature < Sqrt[fstat]} /. curvatures

Out[17]= {True, False}
```

Note that some practitioners prefer to compare the maximum relative curvatures against `Sqrt[fstat/2]`, using the more conservative approach that the relevant curvature should be small compared to `Sqrt[fstat]` for either the assumption of planarity or uniform coordinates to be valid. Using this approach, both assumptions would be rejected in this example. Presumably a least squares parameter estimate based on a larger data set would have improved linearity properties.

■ 2.4 Robust Estimation

Classical statistical procedures assume that errors are distributed normally or at a minimum according to a known distribution. For example, the sample mean `Mean[`*data*`]` is a maximum likelihood estimator (MLE) of location when *data* are distributed normally, while the sample median `Median[`*data*`]` is an MLE of location when *data* are distributed according to a double exponential or Laplacian distribution. These functions are available in ``Statistics`DescriptiveStatistics` `` from the standard package library.

`Median[`*data*`]` may also be looked at as a robust estimator of location when the underlying distribution of *data* is known only to be symmetric. Here a "robust" estimator is taken to mean one that performs tolerably well over a range of distributions. For distributions that are close to a nominal symmetric distribution, but have some symmetric contamination, the median may be too conservative in terms of taking the best possible advantage of the available data. In general, whether the mean or the median is a better location estimator of a symmetrically contaminated normal distribution depends on the level of contamination. The advantages and disadvantages of location estimators under different distributions are easily explored using *Mathematica*.

This extends the definition `Random` to various continuous distributions such as `NormalDistribution` and `LaplaceDistribution`. It also loads the definitions of `Mean`, `Median`, and `Variance` from the ``Statistics`DescriptiveStatistics` `` package.

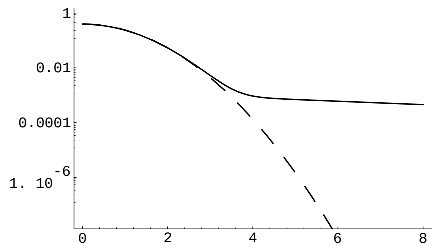
```
In[1]:= <<Statistics`ContinuousDistributions`
```

Load the functions `LogPlot` and `LogLinearListPlot`. `LogPlot` is useful for plotting the tails of exponentially decaying density functions. `LogLinearListPlot` is useful for plotting an estimate's variance as a function of exponentially increasing sample size.

```
In[2]:= <<Graphics`Graphics`
```

Create a set of data from a $\{0,1\}$ normal contaminated by a 2% contribution from a $\{0,7\}$ Laplace distribution. This gives a distribution with heavier than normal tails, as shown by the plot. The log of the tail of an uncontaminated normal is plotted with a dashed line.

```
In[3]:= (contaminated02 =
          Table[If[Random[] > .02,
             Random[NormalDistribution[0, 1]],
             Random[LaplaceDistribution[0, 7]]], {1000}];
          LogPlot[{.98 PDF[NormalDistribution[0, 1], x] +
              .02 PDF[LaplaceDistribution[0, 7], x],
               PDF[NormalDistribution[0, 1], x]},
              {x, 0, 8}, PlotStyle ->
              {{GrayLevel[0]}, {Dashing[{.05, .05}]}}])
```



An estimator can be evaluated by considering its efficiency, *i.e.*, the sample size needed to achieve a given variance. For example, `varMean02` gives the variance of the mean for a set of increasing sample sizes.

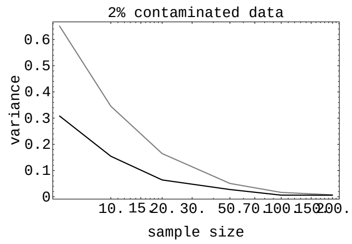
```
In[4]:= (samplesize = {5, 10, 20, 50, 100, 200};
         varMean02 = Map[Variance[Map[Mean,
         Partition[contaminated02, #]]]&,
         samplesize])

Out[4]= {0.651094, 0.345354, 0.164854, 0.0509621,
  0.01665, 0.00733262}
```

Similarly, `varMedian02` gives the variance of the median for the same set of increasing sample sizes.

```
In[5]:= varMedian02 = Map[Variance[Map[Median,
         Partition[contaminated02, #]]]&,
         samplesize]

Out[5]= {0.308122, 0.154554, 0.0642531, 0.0277091,
  0.00628985, 0.00604011}
```

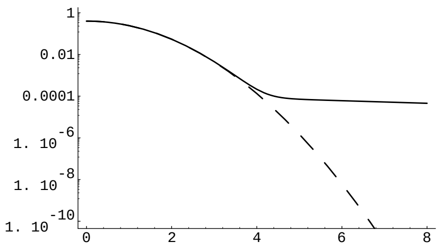
Plotting the variance of the two estimates as a function of sample size indicates that for this moderate level of contamination (2%), the median (plotted in black) outperforms the mean.

```
In[6]:= (varMean02plot =
         LogLinearListPlot[Transpose[{samplesize, varMean02}],
         PlotJoined -> True, DisplayFunction -> Identity,
         PlotStyle -> GrayLevel[.5]];
         varMedian02plot =
         LogLinearListPlot[Transpose[{samplesize, varMedian02}],
         PlotJoined -> True, DisplayFunction -> Identity,
         PlotStyle -> GrayLevel[0]];
         Show[varMean02plot, varMedian02plot,
         PlotLabel -> "2% contaminated data",
         Frame->True, FrameLabel -> {"sample size",
         "variance", None, None},
         DisplayFunction -> $DisplayFunction])
```



Consider a 0.2% level of contamination. The tails of the resulting distribution are not as heavy.

```
In[7]:= (contaminated002 =
          Table[If[Random[] > .002,
          Random[NormalDistribution[0, 1]],
                 Random[LaplaceDistribution[0, 7]]], {1000}];
          LogPlot[{.998 PDF[NormalDistribution[0, 1], x] +
              .002 PDF[LaplaceDistribution[0, 7], x],
               PDF[NormalDistribution[0, 1], x]},
              {x, 0, 8}, PlotStyle ->
              {{GrayLevel[0]}, {Dashing[{.05, .05}]}}])
```

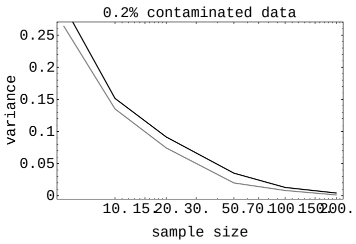


This gives the variance of the mean and median with increasing sample size for the 0.2% contaminated normal data.

```
In[8]:=  {varMean002 = Map[Variance[Map[Mean,
         Partition[contaminated002, #]]]&,
         samplesize],
          varMedian002 = Map[Variance[Map[Median,
         Partition[contaminated002, #]]]&,
         samplesize]}

Out[8]= {{0.264051, 0.135188, 0.0743762, 0.0198436,
   0.0081154, 0.00112864},
  {0.295685, 0.151543, 0.0914846, 0.0350854,
   0.0128447, 0.00411574}}
```

The mean takes into account the value of every data point, whereas the median makes use of only one or two data points located in the middle of the sorted sample. When the contamination is very low, the greater efficiency of the mean (plotted in gray) outweighs the fact that the mean more heavily weights outliers than does the median.

```
In[9]:= (varMean002plot =
         LogLinearListPlot[Transpose[{samplesize, varMean002}],
         PlotJoined -> True, DisplayFunction -> Identity,
         PlotStyle -> GrayLevel[.5]];
         varMedian002plot =
         LogLinearListPlot[Transpose[{samplesize, varMedian002}],
         PlotJoined -> True, DisplayFunction -> Identity,
         PlotStyle -> GrayLevel[0]];
         Show[varMean002plot, varMedian002plot,
         PlotLabel -> "0.2% contaminated data",
         Frame->True, FrameLabel -> {"sample size",
         "variance", None, None},
         DisplayFunction -> $DisplayFunction])
```



Bivariate Median

The definition of the median for univariate data is unambiguous since it is based on sorting a list of numbers. However, in the case of multivariate data, there are several definitions of the median, because there is no unique ordering in dimensions greater than one. Using built-in and package functions, *Mathematica* can be used to define several different bivariate medians.

Begin by defining a function to generate a bivariate normal. The definition follows the paradigm established by the existing statistical distribution packages wherein the built-in function `Random` is extended to accept an object describing a distribution and the rule for `Random` is attached to the symbol associated with the distribution (*i.e.*, `BivariateNormalDistribution`).

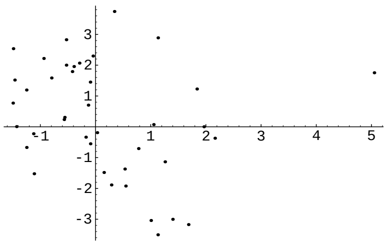
```
In[10]:= BivariateNormalDistribution/:
         Random[BivariateNormalDistribution[
          mu_, sigma_, rho_]] :=
          Module[{z = Random[NormalDistribution[0, 1]],
           r = {Sqrt[1-#], Sqrt[#]}& [Abs[rho]]},
            mu +
            sigma {r. {Random[NormalDistribution[0, 1]], z},
            r. {Random[NormalDistribution[0, 1]], Sign[rho] z}}
         ]
```

Define an outlier distribution that has "doughnut" shaped support, centered at $\mu = \{\mu_1, \mu_2\}$ and having radii $r = \{r_1, r_2\}$.

```
In[11]:= DoughnutDistribution/:
         Random[DoughnutDistribution[mu_, r_]] :=
         mu + {Re[#], Im[#]}& [N[Random[Real, r] *
         Exp[I 2 Pi Random[]]]]
```

This creates and plots a set of bivariate normal data contaminated by 5% outliers.

```
In[12]:=  (data2D = Table[If[Random[] > .05,
            Random[BivariateNormalDistribution[
            {0, 0}, {1, 2}, -.4]],
            Random[DoughnutDistribution[
            {0, 0}, {5, 6}]]], {40}];
         data2Dplot = ListPlot[data2D])
```



For comparison purposes, compute the mean and the componentwise median.

```
In[13]:= {mean2D, median2Dc} =
         {Map[Mean, #], Map[Median, #]} & [Transpose[data2D]]

Out[13]= {{0.154658, 0.251211},
  {-0.0934565, 0.153808}}
```

Oja's estimate of the k-dimensional median (1983) minimizes the sum of the volumes of the k-dimensional simplices the estimate forms with all possible combinations of k members from the sample. When $k = 2$, the volume is the area of a triangle.

```
In[14]:= area[x1_, x2_, x3_] := Abs[Det[Prepend[
         Transpose[{x1, x2, x3}], {1, 1, 1}]]]/2
```

This prints in shortened form the indices of the unique data pairs. Each data pair forms a triangle with the estimate.

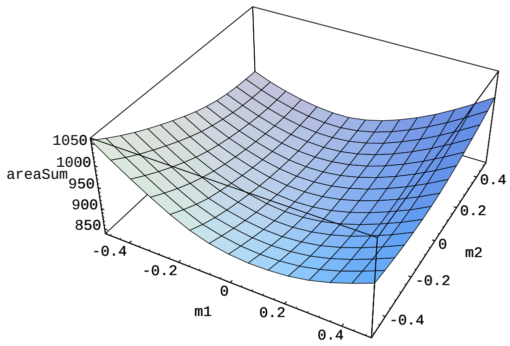
```
In[15]:= ((pairs = Flatten[Table[{i, j},
         {i, 39}, {j, i+1, 40}], 1]) // Short)

Out[15]//Short=
 {{1, 2}, {1, 3}, {1, 4}, <<776>>, {39, 40}}
```

Print in shortened form the sum of the areas in terms of the unknown estimate `{m1, m2}`. The `{m1, m2}` minimizing this expression will be called the "algebraic" median.

```
In[16]:= ((areaSum = Apply[Plus,
          Map[area[#[[1]], #[[2]], {m1, m2}]&,
          Map[data2D[[#]]&, pairs]] ]) // Short)

Out[16]//Short= Abs[6.50851 + <<2>>]
                -------------------- + <<779>>
                         2
```

The `areaSum` expression can be plotted in the neighborhood of the mean and the componentwise median.

```
In[17]:= Plot3D[areaSum, {m1, -.5, .5}, {m2, -.5, .5},
          AxesLabel -> {"m1", "m2", "areaSum"}]
```



Start the search for the algebraic median between the the mean and the componentwise median. It is necessary to use two starting values for `m1` and `m2` because *Mathematica* cannot find the symbolic derivative of an expression involving `Abs`.

```
In[18]:= FindMinimum[areaSum,
          {m1, {mean2D[[1]], median2Dc[[1]]}},
          {m2, {mean2D[[2]], median2Dc[[2]]}}]

Out[18]= {843.309, {m1 -> -0.0849581,
   m2 -> 0.313485}}
```

Define `median2Da` to be the solution of the minimization.

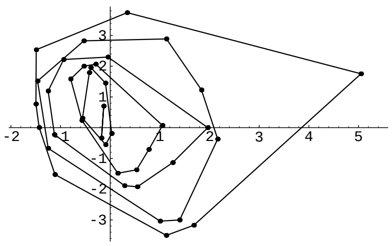
```
In[19]:= median2Da = {m1, m2} /. %[[2]]

Out[19]= {-0.0849581, 0.313485}
```

The idea of "peeling" the convex layers of a point set and taking the mean of the innermost layer to be the multidimensional median is usually attributed to Tukey. This loads a package for finding the bivariate median by the peeling method. The package makes use of `ConvexHull` in `` DiscreteMath`ComputationalGeometry` ``.

```
In[20]:= <<BiMedian`
```

`Layer` gives the indices of the points lying on the convex hull layers, from outermost to innermost. `PlanarGraphPlot` is a function loaded by `` DiscreteMath`ComputationalGeometry` ``.

```
In[21]:= Show[Map[PlanarGraphPlot[data2D, #,
          DisplayFunction -> Identity,
          LabelPoints -> False]&,
          Layer[data2D]],
          DisplayFunction -> $DisplayFunction,
          AspectRatio -> 1/GoldenRatio,
          Axes -> Automatic]
```

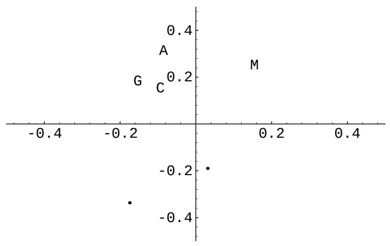


BiMedian makes use of Layer to find the innermost layer. Call this median the "geometric" median.

```
In[22]:= median2Dg = BiMedian[data2D]

Out[22]= {-0.152686, 0.182853}
```

The mean (M), componentwise median (C), algebraic median (A), and geometric median (G) are displayed with sample data located near the origin, the "correct" location of the distribution. The algebraic and geometric medians are both affine invariant location estimates; this is not the case with the componentwise median.

```
In[23]:= Show[data2Dplot, Graphics[{Text[M, mean2D],
           Text[C, median2Dc], Text[A, median2Da],
           Text[G, median2Dg]}],
         PlotRange -> {{-.5, .5}, {-.5, .5}}]
```




```
BeginPackage["BiMedian`", "DiscreteMath`ComputationalGeometry`"]

BiMedian::usage =
"BiMedian[list] gives the median of the points in the list, defined as the
mean of the points lying on the innermost convex layer of the point set."

Layer::usage =
"Layer[list] organizes the indices of points in the list according to the
convex layers of the point set, ordered from outermost to innermost."

Begin["Private`"]

BiMedian[coord_] :=
        Module[{layer = Layer[coord], innermost},
          (innermost = Last[layer];
           Apply[Plus, coord[[innermost]]] / Length[innermost]
          ) /; FreeQ[layer, Layer]]

Layer[coord_] :=
        Module[{result = iLayer[coord]},
                result /; result =!= $Failed]

iLayer[coord_] :=
        Module[{n = Length[coord], output = {}, i, singlelayer, vertices},
                vertices = Range[n];
                If[n <= 3, Return[{vertices}]];
                For[i = 1, Length[vertices] > 3, i++,
                        singlelayer = ConvexHull[coord[[vertices]]];
                        If[!FreeQ[singlelayer, ConvexHull],
                                Message[Layer::conv, i];
                                Return[$Failed]];
                        singlelayer = vertices[[singlelayer]];
                        output = Join[output, {singlelayer}];
                        vertices = Complement[vertices, singlelayer];
                ];
                If[Length[vertices] > 0,
                        output = Join[output, {vertices}]];
                output]

Layer::conv = "ConvexHull failed at layer `1`."

End[]
EndPackage[]
```

The BiMedian` package.

■ 2.5 Spatial Data Analysis

Spatial data usually include both spatial locations and data observed at those locations; the spatial locations may or may not be considered random. Informational databases provide sources of spatial data of the "non-random location" variety, such as statistics associated with cities or regions. An example of this can be found in the package `Miscellaneous`CityData``.

This loads the `Miscellaneous`CityData`` package, one of the packages in *Mathematica*'s standard package library.

```
In[1]:= <<Miscellaneous`CityData`
```

This prints in shortened form a list of the cities in the database that possess a `CityPosition` field.

```
In[2]:= Short[CityData[CityPosition]]

Out[2]//Short=
 {Abu Dhabi, Addis Ababa, Aden, <<116>>, Zurich}
```

The latitude and longitude of Boston is given in {*degree*, *minute*} format.

```
In[3]:= CityPosition["Boston"]

Out[3]= {{42, 20}, {-71, -5}}
```

The `MoreCityData`` package follows the `Miscellaneous`CityData`` database model and includes additional United States cities as well as fields for various city statistics such as `PerCapitaHighways`, the per capita expenditure on highways in the twelve months ending on June 30, 1986.

```
In[4]:= (<<MoreCityData`;
         PerCapitaHighways["Boston"])

Out[4]= 42.5672
```

Consider the cities for which the `PerCapitaHighways` and `CityPosition` fields are defined.

```
In[5]:= (testcities = Intersection[CityData[PerCapitaHighways],
         CityData[CityPosition]]) // Short

Out[5]//Short=
 {Akron, Albuquerque, Anaheim, <<64>>, Wichita}
```

Convert the {*latitude*,*longitude*} position data into $\{x,y\}$ coordinates.

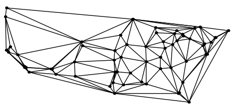
```
In[6]:= (locations = Map[Reverse[N[CityPosition[#] . {1,
        1/60}]]&,
         testcities]) // Short

Out[6]//Short=
 {{-81.5167, 41.0833}, <<66>>, {-97.3333, 37.7}}
```

This loads the definition of `DelaunayTriangulation`, a useful function for visualizing spatial data.

```
In[7]:= <<DiscreteMath`ComputationalGeometry`
```

The Delaunay triangulation of the city locations does not necessarily have much in common with the graph established by major highway arteries.

```
In[8]:= (tri = DelaunayTriangulation[locations];
         PlanarGraphPlot[locations, tri,
         LabelPoints -> False])
```



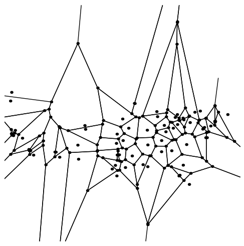
With some imagination, one might hypothesize that there would be some interaction between neighboring locations of the per capita highways expenditure variate, described by the first order autoregressive model

$$y_i = \alpha + \rho \sum w_{ij} y_j + \epsilon_i$$

where $i = 1, \ldots, n$, the sum is over the neighbors of the i^{th} location, α and ρ are parameters, and $\{w_{ij}\}$ is a set of nonnegative weights representing the degree of variate interaction. By convention, w_{ii} is set to zero. Assume that the errors are independently distributed according to a normal $\{0, \sigma^2\}$ distribution. A natural choice is to base the interaction weight for a pair of neighboring cities on the intercity distance and the length of the Voronoi polygon face shared by the cities.

Each Voronoi polygon is centered on a city and adjacent cities share common faces. One can model the interaction as being proportional to the length of the shared face and inversely proportional to the intercity distance.

```
In[9]:= ({vertices, vor} = VoronoiDiagram[locations, tri];
        DiagramPlot[locations, vertices, vor,
        LabelPoints -> False, TrimPoints -> 8])
```



The maximum likelihood estimators for ρ and α are given by

$$\rho = \text{argmin} \left\{ \prod_{i=1}^{n} (1 - \rho\lambda_i) \right\}^{-2/n} (y^T M y - 2\rho y^T M W y + \rho^2 y^T W^T M W y)$$

$$\alpha = n^{-1} 1_n^T (I - \rho W)\, y$$

where W is the matrix of interaction weights, the λ_i are the eigenvalues of W, M is $(n-1)/n$ on the diagonal and $-1/n$ otherwise, and 1_n denotes the length-n vector of ones. (For details see "Estimation Methods for Models of Spatial Interaction", K. Ord, *Journal of the American Statistical Association* (1975), Vol. 70, No. 349.) The following package derives the weight matrix using the Delaunay triangulation and the Voronoi diagram information. This matrix is very sparse.

```
BeginPackage["InteractionWeights`"]

InteractionWeights::usage =
"InteractionWeights[coords, tri, vertices, vor] computes a matrix of
interaction weights for the point coordinates coords, where tri is the
triangulation vertex adjacency list, vertices is the list of polygon vertices,
and vor is the polygon vertex adjacency list.  A weight is nonzero only for
neighboring points in the triangulation and is given by the arctangent of the
ratio of the length of the shared polygon face to the distance between the
points."
```

```
Begin["`Private`"]

InteractionWeights[coords_, tri_, vertices_, vor_] :=
  Module[{weights = Table[0, {#}, {#}] & [Length[coords]],
          locdata = Transpose[Append[Transpose[tri], Transpose[vor][[2]] ]]},
        Scan[Module[{loc1=#[[1]], neighbors=#[[2]], polygon1=#[[3]]},
          Scan[Module[{loc2=#, polygon2, facevertices},
                If[loc2 <= loc1, Return[]]; (* symmetric weights *)
                polygon2 = Select[locdata, (#[[1]] == loc2)&][[1,3]];
                facevertices = vertices[[Intersection[polygon1, polygon2]]];
                weights[[loc1, loc2]] =
                   If[ !Apply[SameQ, Append[Map[Head, facevertices], List]],
                       (* the shared face has infinite length *)
                       Pi/2,
                       (* the shared face has finite length *)
                       ArcTan[Divide @@ Map[(#.# & [Subtract @@ #])&,
                                {facevertices, coords[[{loc1, loc2}]]}]]
                   ];
                weights[[loc2, loc1]] = weights[[loc1, loc2]]
               ]&, neighbors]
        ]&, locdata];
        weights
  ]

End[]
EndPackage[]
```

The `InteractionWeights`` package.

This loads the function for creating the interaction weights matrix.

```
In[10]:= <<InteractionWeights`
```

The interaction weights are based on the city locations, the intercity triangulation (`tri`), the vertices of the city polygons, and the adjacencies of polygon vertices (`vor`).

```
In[11]:= (w = N[InteractionWeights[locations, tri, vertices,
         vor]]) // (Short[#, 7]&)

Out[11]//Short=
 {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1.53643, 0, 0.818571, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 1.23084, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, <<67>>}
```

Normalize the rows of interaction weights so that the influences of neighboring cities sum to one for any particular city. Symmetrical interaction does not seem to be a sensible assumption for cities. When weights are normalized in this way, $\alpha = (1-\rho)\,\bar{y}$ where $\bar{y}$ is the average value of y.

```
In[12]:= (w = Map[(# / Apply[Plus, #])&, w]) // (Short[#, 7]&)

Out[12]//Short=
 {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0.428472, 0, 0.228278, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0.34325, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, <<67>>}
```

The eigenvalues of the weight matrix are needed to construct the loglikelihood function that is to be minimized over ρ.

```
In[13]:= (evals = Sort[Eigenvalues[w]]) // Short

Out[13]//Short=
 {-0.992833, -0.980984, -0.949973, <<64>>, 1.}
```

Take the observations to be the per capita highway expenditure statistics of the test cities.

```
In[14]:= (y = Map[PerCapitaHighways, testcities]) // Short

Out[14]//Short=
 {101.784, 73.5736, 40.6965, <<64>>, 37.1443}
```

Define the log of the likelihood function.

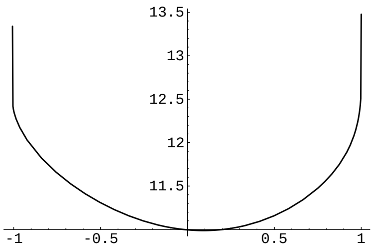
```
In[15]:= (n = Length[y];
         m = IdentityMatrix[n] - 1/n;
         wy = w.y;
         (loglike = Log[y.m.y - 2 rho y.m.wy + rho^2 wy.m.wy] -
          2/n Apply[Plus, Log[1 - rho evals]]) // Short[#, 5]&)

Out[15]//Short=
 -(Log[1 - 1. rho] + Log[1 - 0.999602 rho] +
      Log[1 - 0.99278 rho] +
      Log[1 - 0.991758 rho] + <<62>> +
      Log[1 + 0.980984 rho] + Log[1 + 0.992833 rho])
    / 34 + <<1>>
```

ρ must lie between $1/\lambda_{min}$ and $1/\lambda_{max}$ for the eigenvalue terms in `loglike` to be real.

```
In[16]:= {rhoMin, rhoMax} =
        {1/evals[[1]] + $MachineEpsilon,
         1/evals[[n]] - $MachineEpsilon}

Out[16]= {-1.00722, 1.}
```

This shows the loglikelihood function between the bounds on ρ.

```
In[17]:= Plot[loglike, {rho, rhoMin, rhoMax}]
```



Minimize the loglikelihood function.

```
In[18]:= FindMinimum[loglike, {rho, (rhoMin + rhoMax)/2,
         rhoMin, rhoMax}]

Out[18]= {10.9881, {rho -> 0.0915997}}
```

Compute the estimates for ρ, α, and σ.

```
In[19]:= {rhohat, alphahat, sigmahat} =
          {rho, Apply[Plus, y - rho wy]/n,
           N[Sqrt[(y - rho wy).m.(y - rho wy)/n]]} /. %[[2]]

Out[19]= {0.0915997, 59.7668, 29.4441}
```

The model is very poor; the estimate of σ using the interaction model (`sigmahat`) is approximately the same as the estimate of the standard deviation of y (`sigmay`).

```
In[20]:= sigmay = (Sqrt[#.#/n]) & [y - Apply[Plus, y]/n]

Out[20]= 29.6156
```

Load `BinCounts` for histogramming.

```
In[21]:= <<Statistics`DataManipulation`
```

These are the frequencies of the residuals for seven bins.

```
In[22]:= (residuals = y - rhohat wy - alphahat;
         freq = BinCounts[residuals, {-60., 80., 20.}])

Out[22]= {3, 12, 25, 11, 8, 6, 2}
```

Load `BarChart`.

```
In[23]:= <<Graphics`Graphics`
```

The freqencies versus bin midpoints may be plotted using `BarChart` to display a histogram of the residuals. It is useful to examine the residuals even in the case of a good model.

```
In[24]:= BarChart[Transpose[{freq,
          {-50., -30., -10., 10., 30., 50., 70.}}]]
```