1 Using data perturbations for sensitivity analysis

An easy-to-use exploratory test for numerical and measurement error stability for a given model is to introduce small random perturbations to the data, on the order of the measurement error of the instruments used to collect it, and recalculate the estimate. When the estimates produced using this technique vary greatly, the model estimation is necessarily unstable. And although the converse is not necessarily true, where a model is already known to be statistically appropriate, this type of sensitivity analysis will give the researcher greater confidence that the their results are robust to numerical and measurement error.

We have developed a package in R that makes perturbation-based sensitivity analysis simple to apply and to interpret. For most models this running a sensitivity analysis involves only two steps.

1. Specify the data, model, and model options for the unperturbed model, and optionally, the error functions for the perturbation.

2. Use `summary()` or `plot(summary())` to see the sensitivity of the parameter estimates to perturbations.

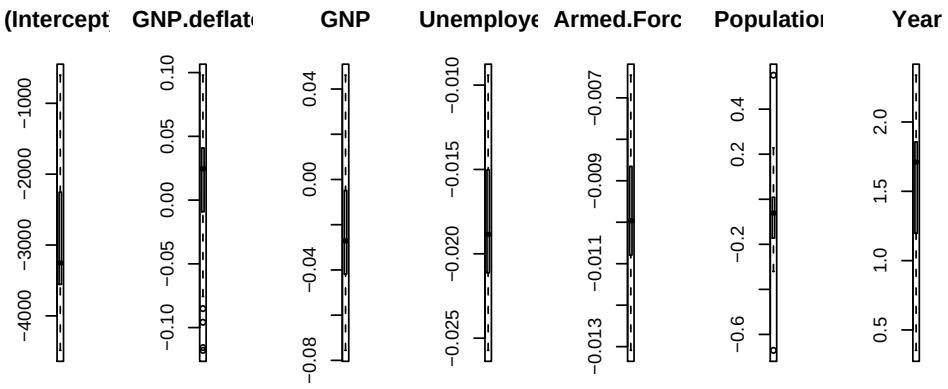
Perturb works automatically almost with any **R** model, such as `lm`, `glm`, and `nls`, that accepts `data` as an argument to supply data and that returns estimated coefficients through `coef()`.

The example below shows how to conduct a sensitivity analysis of the classic analysis by Longley (1964) using `sensitivity()` and default noise functions.

```
> plongley = sensitivity(longley, lm, Employed ~ .)
> print(summary(plongley), digits = 4)

[1] "Sensitivity of coefficients over  50  perturbations:"
             Perturb Est. (Orig. Est.) (Orig. Stderr)       2.5%      97.5%
(Intercept)    -2.880e+03   -3.482e+03      8.904e+02 -4.417e+03 -7.858e+02
GNP.deflator    1.312e-02    1.506e-02      8.491e-02 -1.112e-01  8.920e-02
GNP            -2.056e-02   -3.582e-02      3.349e-02 -6.567e-02  4.046e-02
Unemployed     -1.803e-02   -2.020e-02      4.884e-03 -2.517e-02 -9.615e-03
Armed.Forces   -9.763e-03   -1.033e-02      2.143e-03 -1.279e-02 -6.757e-03
Population     -7.140e-02   -5.110e-02      2.261e-01 -3.164e-01  2.227e-01
Year            1.519e+00    1.829e+00      4.555e-01  4.392e-01  2.307e+00
             [Unstable]
(Intercept)           *
GNP.deflator
GNP                   *
Unemployed            *
Armed.Forces
Population
Year                  *
```

The sensitivity results can also be expressed in plot format:

```
> plot(summary(plongley))
```



This is a rare example of a model that is very sensitive to noise. Even so, note that the small amounts of noise applied tremendously alter some of the estimated coefficients, but not others. In most practical cases, however, the substantive implications of your model will remain the same across the sensitivity analysis – in which case, you can publish them with greater confidence.

If error functions are not specified, a default set of error function will be selected based on measurement types of the variable: continuous, ordered, or unordered. Continuous variables, by default are subject to a small amount of mean-zero component-wise uniformly distributed noise, which is typical of instrumentation-driven measurement error. Ordered factors are assigned a small probability of having observations reclassified to the neighboring classification, and unordered factors have a small probability of being reassigned to another legal value.

Alternatively, one can specify the error functions to use yourself, or use one of many supplied by *accuracy*. The *accuracy* package comes with a wide range of noise functions for continuous distributions, and random reclassification of factors. [1]

Your choice of error functions should be chosen to reflect measurement error model for the specific data you are using. In numerical analysis, uniform noise is often used since this is what would be expected from simple rounding error. Normal random noise is commonly used in statistics, under the assumption that measurement error is the sum of multiple independent error processes. In addition, when normal perturbations are used, the result can be interpreted, for many models, as equivalent to the results of running a slightly perturbed *model* on unperturbed data. In some cases, like discrete or ratio variables, other forms of noise are necessary to preserve the structure of the problem. (see for example, Altman, Gill, McDonald 2005). The magnitude of the noise is also under the control of the researcher. Most use a magnitude equivalent to the researchers estimate of the underlying measurement error in the data. Noise is usually adjusted to the size of each component, since this better preserves the structure of the problem, however in some cases the underlying measurement error model may imply norm-wise scaling of the noise. For more information on noise distributions and measurement error models see , e.g., Belsley 1991, Chaitin-Chatelin & Traviesas-Caasan (2004b), Caroll et. al (1995), Cheng & Van Ness (1999), Fuller (1987).

If multiple plausible measurement error models can be hypothesized, we recommend that `sensitivity` be run multiple times with different noise specifications, However, in our experience with social science analyses, the choice of error model does not tend to effect, in practice, the substantive conclusions from the sensitivity analysis.

[1] The `perturb` package for collinearity diagnosis by Hendrickx, et. al (2004) (which was developed for `R` after the `accuracy` package) provides additional methods for randomly reclassifying factors that via its `reclassify()` function. This function can be used in conjunction with `accuracy`. Hendrickx, et. al also provide a number of collinearity diagnostics, including one based on data perturbations.

Some researchers omit perturbations to outcome variables, since, in terms of statistical theory, mean-zero measurement error on outcome variables (as opposed to explanatory variables) contribute only to increased variance in estimates, not bias. While this attitude is well-justified in the context of statistical theory, it is not similarly justified in the computational realm. If the estimation of a model is computationally unstable, errors in the outcome variable may have large and unpredictable biases on the model estimate. Hence, the conservative default in our package is to subject all variables to perturbation, although options are available to completely control the form and magnitude of all perturbations.

Consider this example, which shows a sensitivity analysis of the anorexia analysis described in Venables and Ripley (2002). In this case, we leave the dependent variable unperturbed, by assigning it the *identity* error function.

```
> data(anorexia, package = "MASS")
> panorexia = sensitivity(anorexia, glm, Postwt ~ Prewt + Treat +
+     offset(Prewt), family = gaussian, ptb.R = 100, ptb.ran.gen = c(PTBi,
+     PTBus, PTBus), ptb.s = c(1, 0.005, 0.005))
> print(summary(panorexia), digits = 4)

[1] "Sensitivity of coefficients over  100  perturbations:"
            Perturb Est. (Orig. Est.) (Orig. Stderr)    2.5%   97.5% [Unstable]
(Intercept)       49.811      49.7711        13.3910 49.0986 50.5302
Prewt             -0.566      -0.5655         0.1612 -0.5746 -0.5573
TreatCont         -4.097      -4.0971         1.8935 -4.1657 -4.0410
TreatFT            4.562       4.5631         2.1333  4.4886  4.6490
```

Finally, if a model in `R` does not take a `data` argument or does not return coefficients through the `coef` method, it is usually only a matter of a few minutes to write a small wrapper that calls the original model with appropriate data, and that provides a `coef` method for retrieving the results. (Alternatively, you might to choose to run such models in `Zelig`, as described in the next section.)

For example, the `mle` function for maximum-likelihood estimation does not have an explicit `data` option. Instead, it normally receives data implicitly through the log-likelihood function, `ll`, passed into it. To adapt it for use in `sensitivity` we simply construct a another function that accepts data and a log-likelihood function separately, constructs a temporary log-likelihood function with the data passed in the environment, and then calls `mle` with the temporary function:

```
> mleD <- function(data, lld, ...) {
+     f = formals(lld)
+     f[1] = NULL
+     ll <- function() {
+         cl = as.list(match.call())
+         cl[1] = NULL
+         cl$data = as.name("data")
+         do.call(lld, cl)
+     }
+     formals(ll) = f
+     mle(ll, ...)
+ }
```

Finally, construct the log-likelihood function to accept data. As in this example, which is based on the documented example in the `Stats4` package:

```
> library(stats4)
> dat = as.data.frame(cbind(0:10, c(26, 17, 13, 12, 20, 5, 9, 8,
+     5, 4, 8)))
> llD <- function(data, ymax = 15, xhalf = 6) -sum(stats::dpois(data[[2]],
+     lambda = ymax/(1 + data[[1]]/xhalf), log = TRUE))
> print(summary(sensitivity(dat, mleD, llD)), digits = 4)

[1] "Sensitivity of coefficients over  50  perturbations:"
      Perturb Est. (Orig. Est.)    min   2.5%  97.5%    max
ymax        25.081       24.993 21.862 24.993 26.097 29.989
xhalf        3.044        3.057  2.355  2.741  3.057  3.879
```

1.1 Sensitivity analysis using `Zelig`

`Zelig` (Imai, et. al 2005) is an easy-to-use R package that can estimate and help interpret the results of a large range of statistical models. `Zelig` provides a uniform interface to these models the `Accuracy` package utilizes to enable sensitivity analyses. In addition, `Accuracy` can also be used to perform sensitivity analyses of the robust alternatives, simulated predicted values, expected values,

first differences, and risk ratios that `Zelig` produces for all the models it supports. [2] So, using these packages together is an easy way to analyze the sensitivity of *predicted values* to measurememnt error.

To illustrate, we replicate Longley's analysis (above), using `zelig()` (instead of `lm()`) to run the OLS model, and the convenience function `sensitivityZelig()` to run the sensitivity analysis:

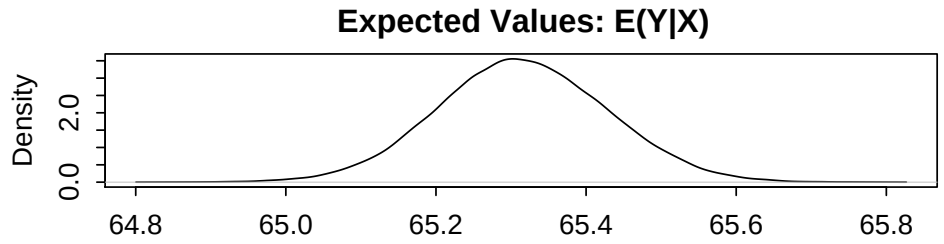
```
> if (require("Zelig", quietly = T, warn.conflicts = F)) {
+     zelig.out = zelig(Employed ~ GNP.deflator + GNP + Unemployed +
+         Armed.Forces + Population + Year, "ls", longley)
+     perturb.zelig.out = sensitivityZelig(zelig.out)
+ }

##
##  Zelig (Version 2.8-4, built: 2007-06-01)
##  Please refer to http://gking.harvard.edu/zelig for full documentation
##  or help.zelig() for help with commands and models supported by Zelig.
##
```

Just as above, `summary()` and `plot(summary())` can be used summarize the sensitivity of the model coefficients. In addition, we can use the `Zelig` methods `setx` and `sim` to simulate various quantities of interest. And when `summary()` and `plot()` are used, they will display a *sensitivity analysis* of the predicted values.

For example, this code generates predictions of the distribution of the explanatory variable, 'Employed', around the point where 'Year' equals 1955, and the other variables are at their means, and creates a profile plot of the predicted distribution of the explanatory variable:

[2] `Zelig` also integrates nonparametric matching methods as an optional preprocessing step. Thus `Accuracy` supports sensitivity analysis of models subject to such pre-processing as well.

```
> if (require("Zelig", quietly = T, warn.conflicts = F)) {
+     setx.out = setx(perturb.zelig.out, Year = 1955)
+     sim.perturb.zelig.out = psim(perturb.zelig.out, setx.out)
+     print(summary(sim.perturb.zelig.out))
+ }

**** 50  COMBINED perturbation simulations

  Model: ls
  Number of simulations: 1000

Values of X
     (Intercept) GNP.deflator   GNP Unemployed Armed.Forces Population Year
1947           1        101.7 387.7      319.3        260.7      117.4 1954

Expected Values: E(Y|X)
      mean     sd 2.5% 97.5%
1947 65.32 0.1137 65.1 65.54

> if (require("Zelig", quietly = T, warn.conflicts = F)) {
+     plot(sim.perturb.zelig.out)
+ }

**** 50  COMBINED perturbation simulations
```



1.2 True random numbers through entropy collection

'Random' numbers aren't. The numbers provided by routines such as `runif()` are not genuinely random. Instead, they are *pseudo-random number generators* (PRNGs), deterministic processes that create a sequence of numbers. Pseudo-random number generators start with a single "seed" value (specified by the user or left at defaults) and generate a repeating sequence with a certain fixed length, or period p. This sequence is statistically similar, in limited respects, to random draws from a uniform distribution.

The earliest PRNGs, still in use in some places, and used in early versions of R, is the Linear Congruential Generator (LCG), which is defined as:

$$\begin{aligned} LCG(a, m, s, c) \equiv & \\ & x_0 = s, \\ & x_n = (ax_{n-1} + c) \bmod m. \end{aligned} \tag{1}$$

(All parameters are integers, and in practice x is usually divided by m to yield numbers between zero and one.)

This function generates a sequence of numbers between $[0, m-1]$ which appears to be, using some tests, uniformly distributed in that range. Other PNRG's are more complex, but share with the LCG the fundamental properties of determinism and periodicity. See (Gentle 1998) for an extensive treatment of modern PRNG's and theory.

`R` provides several high quality PRNG's natively, and packages such as `gsl`, `rstream` and `rpsrng` which can be used to generate quasi-random number streams, and concurrent PRNG streams. Regardless of the particular PRNG algorithm used, however, a PRNG cannot perfectly mimic a random sequence. And, in fact, there is no complete theory to describe the domains for which PRNG and true random sequences can be considered interchangeable. In addition, the theory on which PRNG's are based assumes that the seed itself is *truly* random.

The `runifT()` routine is different from other random number generators in R. It delivers true random numbers based on entropy collected from external physical sources of randomness.

Two sources of randomness are currently supported. On Unix and Linux system, the kernel gathers environmental noise from device drivers and other sources into a system entropy pool. This pool can be accessed through the '/dev/random' pseudo-device. Alternatively, the "Hotbits" web server, run by FourmiLab provides random bytes based on radioactive decay.

Using either source, these routines will retrieve random bits in chunks, and keep them in a local pool. This pool will be used as necessary to satisfy calls to `runifT()` and `resetSeed()`, and will be

automatically refreshed from the external sources when empty. If external sources are unavailable, the pool is refreshed using standard PRNG's.

Entropy collection is relatively slow compared to PRNGS. So, these routines are most efficient for generating either small numbers of very-high-quality random numbers (e.g. for cryptography) or for seeding (and regularly reseeding) PRNG's. The function `resetSeed()` sets the seed for the standard PRNGś using true random bits. The `runifS()` automates this process further, by reseeding runif() with random values, periodically to improve the random properties of the resulting sequence:

```
> birthday <- function(x, n = 2^20) {
+     spacings = diff(trunc((x * .Machine$integer.max)%%n))
+     tab = table(spacings)
+     tab = tab[which(tab > 1)]
+     chisq.test(sample(tab, 200, replace = T))
+ }
> resetSeed()

  [1]         403         510  -222047061 -1879145125  1999631555  1118884790
  [7]  1716650372  -359304945  -507983749  1146161482  1874669144   901019456
 [13] -1268531138   -11334818 -1035756596 -1459732929   484367461  1626982928
 [19]  -355112847   399089832  1996549344  1252172215  -474337806  1960265124
 [25]  1445993260  -526140316   146150757   -41736198   743551686   732975172
 [31]  1870058014    98043700  1884900185 -1482336184 -1381356215  -495894649
 [37] -1590961630 -1094685022  2027925340  -269028185   561760442  1857312899
 [43] -1149290016  -434487859   234537540  1496176061  1479903066   889188435
 [49]  -815626172  -717352227 -1609983340  -702472787  1359096018   736905435
 [55]   440677651 -1773406783   -11443522  -744887575  1006793284 -1075086730
 [61] -1497426157 -1966211142   -16755315   277416594   929760520  -895335158
 [67]   -73298802  1146888329  1897945435  -400309409  1453057505   286749905
 [73] -1403750336   810656530  1408950444 -1082045901 -1913186023  1861633732
 [79]  -277615904  1828129434   938360278  1251297183  1739628075 -1923289459
 [85]     4504211   344664612   737924520   516011272  2001861111  1052720308
 [91]  -285985745   602622587  1040766822  -262936833  1186705385   -48189235
 [97]  1639296794  1651132128    -7372494 -1513671875   275778939  1368411992
[103]   622606550  -961381527   -98420519  1504524008  1190613942  -844268026
[109]  -948431587 -1849669293  -835883768  2046420796   440916308 -1863214416
[115] -1269069343  -850004080  -180469100  -501430431 -1648529714  1192860118
[121]  1389709476 -1134590567 -1581505480  1030588508  -414901654  1947034995
[127]   729400943  -787804024  1356929310  -787006935 -1999554279   949949545
[133]  1550998540   453540229  1088960891   465139301   498830991  1959683941
[139]  1293928253  -632671003  1870225839 -1003667951 -1620599852   964051009
[145]  -150767997  1641567244   -12263429 -1890926533  1176383116   886724823
[151]  -241476367 -1813043289  1150240006  1452372953   958082155   116408758
[157]  1358825695 -2109992666  1264515621  1369088239  1809070015 -1205069283
[163]  -398305435  1063893731 -1781922767 -1172724692  1475082589   528898608
[169] -1162507493  -879449354   341091632 -2070164558   382702975  -662102551
[175]  -550020107  -781269942  1663185579   -99400690  -534946004   809665889
[181] -1227912269  -198330907 -1154858992 -2108880013  -773156911 -1130272799
```

1.3 Tests for global optimality

The estimation of many statistical models rests on finding the global optimum to a user-specified non-linear function. R provides a number of tools for such estimations, including `nlm()`, `nls()`, `mle()`, `optim()` and `constrOptim()`.

All of these functions rely on local search algorithms, and the results they return may depend on the starting point of the search. Maximum likelihood functions, non-linear-regression models, and the like, are not guaranteed to be globally convex in general. And even where convexity is guaranteed by statistical theory, inaccuracies in statistical computation can sometimes induce false local optima (discontinuities that may cause local search algorithms to converge, or at least stop). A poor or unlucky choice of starting values may cause a search algorithm to converge at a local optimum, which may be far from the real global optimum of the function. Inferences based on the values of the parameter at the local optimum will be incorrect.

Knowing when a function has reached its true maximum is something of an art. While the plausibility of the solution in substantive terms is often used as a check, relying solely on the expected answer as a diagnostic might bias researchers toward Type I errors. Diagnostic tests are therefore useful to provide evidence that computed solution is the true solution.

A number of strategies related to the choice of starting values have been formalized as tests or global optimality. In this package we implement two. The 'Starr' test and the 'Dehaan' test. [3] [4]

The intuition behind the Starr test statistic is to run the optimization from different starting points to observe 'basins of attraction', and then to estimate the number of *unobserved* basins of attraction from the number of observed basins of attraction. The greater the number of observed basins of attraction, the lower the probability that a global optimum has been located. This idea has been attributed to Turing (1948), and the test statistics was developed by Starr (1979):

$$V_2 = \frac{S}{r} + \frac{2D}{r\,(r-1)}. \tag{2}$$

Here V_2 is the probability a convergence point has not been observed, and r is the number of randomly chosen starting points. S is the number of convergence points that were produced from one (or a Single) starting value and D is the number of convergence points that were produced from two (or Double) different starting values.

[3] In addition to these tests, the `R` user may also wish to investigate the `bhat` package, which can generate diagnostic profile likelihood plots.

[4] If this indicats that the optimum has not been reached, the user may consider using heuristics designed for non-smooth optimization problems, such as the simulated annealing option for `optim()`, or the optimizers provided by the `gafit`, `genalg`, `rgenoud` modules.

Finch, Mendell, and Thode (1989) demonstrate the value of the statistic by analyzing a one parameter equation on a $[0, 1]$ interval for $r = 100$. While the proposed statistic given by the above equation is compelling, their example is similar to an exhaustive grid search on the $[0, 1]$ interval. (Starr's result is further generalizable for triples and higher order observed clumping of starting values into their basins of attraction, but Finch, Mendell, and Thode assert that counting the number of singles and doubles is usually sufficient.)

The statistic may be infeasible to compute for an unbounded parameter space with high dimensionality. However, the intuition behind the statistic can still be soundly applied in these cases. If multiple local optima are identified over the course of a search for good starting values, a researcher should not simply stop once an apparent best fit has been found, especially if there are a number of local optima which have basins of attraction that were identified only once or twice. Our implementation of the Staff test provides a ready-to-use-interface that can be easily incorporated into a search of the parameter space for good optimization starting values.

For computationally intensive problems, another test, by Veall (1990), drawing upon a result presented by de Haan (1981), may be more practical. The de Haan/Veall test relies on sampling the optimization function itself rather than identifying basins of attraction. A confidence interval for the value of the likelihood function's global optimum is generated from the points sampled from the likelihood surface. This procedure is much faster than the Starr test because the likelihood function is calculated only once for each trial. As with starting value searches, researchers are advised to increase the bounds of the search area and the number of trials if the function to be evaluated has a high degree of dimensionality or a high number of local optimum have been identified.

Veall suggests that by using a random search and applying extreme asymptotic theory, a confidence interval for the candidate solution can be formulated. The method, according to Veall (1990: 1460) is to randomly choose a large number, n, of values for the parameter vector using a uniform density over the entire parameter space. Call the largest value of the evaluated likelihood function L_1 and the second largest value L_2. The $1 - p$ confidence interval for the candidate solution, $L^{'}$, is $[L_1, L^p]$ where:

$$L^p = L_1 + \frac{L_1 - L_2}{p^{-1/\alpha} - 1} \tag{3}$$

and $\alpha = k/2$, where k is some function that depends on n such that $k(n) \to 0$, as $k(n), n \to \infty$ (a likely candidate is $k = \sqrt{n}$).

As Veall (1990: 1461) notes, the bounds on the search of the parameter space must be large enough to capture the global maximum and n must be large enough to apply asymptotic theory. In Monte Carlo simulations, Veall suggests that 500 trials are sufficient for rejecting that a local

optimum is not the *a priori* identified global optimum.

Examples of applying both the dehaan and starr tests are below:

```
> data("BOD")
> stval = expand.grid(A = seq(10, 100, 10), lrc = seq(0.5, 0.8,
+     0.1))
> llfun <- function(A, lrc) -sum((BOD$demand - A * (1 - exp(-exp(lrc) *
+     BOD$Time)))^2)
> lls = NULL
> for (i in 1:nrow(stval)) {
+     lls = rbind(lls, llfun(stval[i, 1], stval[i, 2]))
+ }
> fm1 <- nls(demand ~ A * (1 - exp(-exp(lrc) * Time)), data = BOD,
+     start = c(A = 20, lrc = log(0.35)))
> ss = -sum(resid(fm1)^2)
> dehaan(lls, ss)

[1] TRUE

> llb = NULL
> for (i in 1:nrow(stval)) {
+     llb = rbind(llb, coef(nls(demand ~ A * (1 - exp(-exp(lrc) *
+         Time)), data = BOD, start = c(A = stval[i, 1], lrc = stval[i,
+         2]))))
+ }
> starr(llb)

[1] 0
```

1.4 A generalized Cholesky method

The generalized inverse is a commonly used technique in statistical analysis, but the generalized Cholesky has not before been used for statistical purposes, to our knowledge. When the inverse of the negative Hessian does not exist, we suggest two separate procedures to choose from. One is to create a *pseudo-variance matrix* and use it, in place of the inverse, in an importance resampling scheme. In brief, applying a generalized inverse (when necessary, to avoid singularity) and

generalized Cholesky decomposition (when necessary, to guarantee positive definiteness) together often produce a pseudo-variance matrix for the mode that is a reasonable summary of the curvature of the posterior distribution. This method is developed and analyzed in detail in (Gill and King, 2004), here we provide a brief sketch.

The Gill/Murray Cholesky factorization of a singular matrix C, adds a diagonal matrix E such that the standard Cholesky procedure is defined. Unfortunately it often increments C by an amount much larger than necessary providing a pseudo-Cholesky result that is further away from the intended result. Schnabel and Eskow (1990) improve on the C+E procedure of Gill and Murray by applying the Gerschgorin Circle Theorem to reduce the infinity norm of the E matrix. The strategy is to calculate delta values that reduce the *overall* difference between the singular matrix and the incremented matrix. This improves the Gill/Murray approach of incrementing diagonal values of a singular matrix sufficiently that Cholesky steps can be performed.

This technique is complex to describe but simple to use:

```
> S <- matrix(c(2, 0, 2.5, 0, 2, 0, 2.5, 0, 3), ncol = 3)
> sechol(S)

      [,1]  [,2]     [,3]
[1,] 1.414 0.000 1.767767
[2,] 0.000 1.414 0.000000
[3,] 0.000 0.000 0.004262
attr(,"delta")
[1] 1.817e-05

> t(T)

     [,1]
[1,] TRUE
```

2 References

Altman M, Gill J, McDonald MP (2003). *Numerical Issues in Statistical Computing for the Social Scientist.* John Wiley & Sons, New York.

Belsley DA (1991). *Conditioning diagnostics, collinearity and weak data in regression.* John Wiley

& Sons, New York.

Chaitin-Chatelin F, Traviesas-Caasan E (2004b). "Qualitative Computing.", In Bo Einarsson (ed.), *Accuracy and Reliability in Scientific Computing.* SIAM Press, Philadelphia.

Cheng C, Van Ness JW (1999). *Statistical Regression with Measurement Error.* Arnold, London.

de Haan, L (1981). "Estimation of the Minimum of a Function Using Order Statistics." *Journal of the American Statistical Association*, **76**, 467-9.

Fuller WA (1987). *Measurement Error Models.* John Wiley & Sons, New York.

Gill J & King G (2004). "What to do When Your Hessian is Not Invertible: Alternatives to Model Respecification in Nonlinear Estimation." *Sociological Methods and Research*, **32**(1), 54-87.

Hendrickx J, Belzer B, te Grotenhuis M, Lammers J (2004). "Collinearity Involving Ordered and Unordered Categorical Variables." Presented at "RC33 conference in Amsterdam, August 17-20". URL `http://www.xs4all.nl/~jhckx/perturb/`.

Imai K, King G, Lau O (2005). "Zelig: Everyone's Statistical Software." R package version 2.4-5. `http://gking.harvard.edu/zelig`

Longley, JW (1967). "An Appraisal of Computer Programs for the Electronic Computer from the Point o f View of the User." *Journal of the American Statistical Association*, **62**, 819-41.

Schnabel RB, Eskow E (1990). "A New Modified Cholesky Factorization." *SIAM Journal of Scientific Statistical Computing*, **11**, 1136-58.

Veall MR (1990). "Testing for a Global Maximum in an Econometric Context." *Econometrica*, **58**, 1459-65.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S. Fourth Edition.* Springer, New York.