Editor: Dianne P. O'Leary, oleary@cs.umd.edu

MONTE CARLO MINIMIZATION AND COUNTING: ONE, TWO, …, TOO MANY

By Isabel Beichl, Dianne P. O'Leary, and Francis Sullivan

Monte Carlo methods use sampling to produce approximate solutions to problems for which other methods aren't practical. In this homework assignment, we study three uses of Monte Carlo methods: for function minimization, discrete optimization, and counting.

In the long run, the stock market always goes up—or so we're told. Thus, a sure way to increase your wealth is to buy some shares of everything sold on the New York Stock Exchange. But wait a minute, you say—that's not practical, and it would cost way too much money. Isn't there some way to buy a *sample* of shares in a way that guarantees it goes up along with the market? Well, yes, actually, there is—you could buy mutual funds. Of course, there's a risk that the fund you buy won't do a good job of sampling and will go down instead of up, but that probably won't happen with a well-established fund that uses a good sampling algorithm, one that really *does* follow the market.

Monte Carlo methods embody this sampling philosophy in a set of remarkably versatile algorithms that prove themselves useful when other methods aren't practical for solving difficult numerical problems. When well-designed, they can tell you a lot about what's going on without forcing you to look at every possibility. We focus in this case study on three uses of Monte Carlo methods: for function minimization, for discrete optimization, and for counting.

Function Minimization

A strictly convex function $f(\mathbf{x})$ in the interval, $\mathbf{a} \le \mathbf{x} \le \mathbf{b}$, attains a minimum that we can find with a variety of methods, including the many versions of Newton's method, conjugate gradients, and (if derivatives aren't available) pattern search algorithms. For nonconvex functions, such as that in Figure 1, these algorithms find a *local minimizer* such as $x = 0.4$ but aren't guaranteed to find the *global minimizer* $x^* = 1.8$.

Minimization Using Monte Carlo Techniques

Monte Carlo methods provide a good means for generating starting points for nonconvex optimization problems. In its simplest form, a Monte Carlo method generates a random sample of points in the function's domain. We can then use our favorite minimization algorithm starting from each of these points and, among the minimizers found, report the best one. By increasing the number of Monte Carlo points, we increase the *probability* that we'll find the global minimizer. Algorithm 1 summarizes this method. Note that our "favorite" minimization algorithm can be as simple as reporting the starting point.

ALGORITHM 1. MONTE CARLO MINIMIZATION

We want to minimize the function $f(\mathbf{x})$ over the region $\mathbf{a} \le \mathbf{x} \le \mathbf{b}$.
for each random point $\mathbf{x}_i$ generated in the region,
  Use an algorithm to approximate a local minimizer of $f(\mathbf{x})$, starting at $\mathbf{x}_i$, restricting the search to the region defined by **a** and **b**.
  If the resulting minimizer gives a lower function value than all previous minimizers, remember it.
end

Just as more information helps in choosing a mutual fund, this Monte Carlo method can be improved somewhat by using extra information about the function $f(\mathbf{x})$ that we're minimizing. Suppose we know a *Lipschitz constant* L for our function, so that for all **x** and **y** in the domain,

$$|f(\mathbf{x}) - f(\mathbf{y})| \le L\,\|\mathbf{x} - \mathbf{y}\|.$$

To make the example specific, let's suppose $L = 1$ and that we know $f(1) = 2$ and $f(4) = 0$. Because we know $f(4) = 0$, we know the global minimizer gives a function value of at most zero. Thus, we're no longer interested in looking at intervals that will have function values greater than zero. Our Lipschitz relation tells us that f can't decrease faster than a straight line with slope –1. In Figure 2, the blue shaded area shows where the function f must lie. In the interval marked

with a red line, we know f isn't less than $f(4)$, so we can exclude this interval (–1, 3) and not search in it for the minimizer.

This idea forms the basis of a more complicated, but somewhat more efficient, algorithm, Algorithm 2.

Algorithm 2. Monte Carlo Minimization Using Lipschitz Information

We want to minimize the function $f(x)$ over the interval $[a, b]$. Initially, the list of excluded intervals is empty.

for each random point x_i
  Determine whether the point x_i is in an excluded interval or in some subinterval of $[a, b]$ outside the excluded intervals.
if x_i isn't in an excluded interval, **then**
  Use an algorithm to approximate a local minimizer of $f(x)$, starting at x_i, restricting the search to the subinterval. If the resulting minimizer gives a lower function value than all previous minimizers, remember it.
  Use the Lipschitz condition to generate a new excluded interval (see the example in Figure 2) and update the list.
  end
end

In Problem 1, we can experiment with our two algorithms for Monte Carlo minimization.

Problem 1.

Consider the function `myf.m` on the Web site (www.computer.org/cise/homework/), with domain $0 \le x \le 7$.

(a) Generate 500 uniformly distributed points on the interval [0, 7] in Algorithm 1, using `fmincon` for the local minimizer. Make a graph illustrating the minimizer corresponding to each starting point.

(b) $L = 90.3$ is a Lipschitz constant for the function `myf.m`. Use Algorithm 2 on the interval [0, 7]. Compare the two methods' performance.

(c) (Extra) Try Monte Carlo minimization on your favorite function of n variables for $n > 1$.

Minimization Using Simulated Annealing

Suppose we have a box of particles that we have allowed to cool slowly. Naturally, we expect the system's *potential energy* to be small—for example, if you make ice in your freezer, the crystal structure that results has a lower potential energy than most of the alternatives. If we drop the temperature too fast, the system can easily get stuck in a configuration that has a higher potential energy; the crack lines we often see in ice cubes illustrate this. The physical process of slow cooling is called *annealing*, and, with luck, it can lead to something close to a perfect ice cube. The annealing process works because if the temperature is high, each particle has a lot of kinetic energy and can easily move to positions that increase the system's potential energy. This helps the system avoid getting

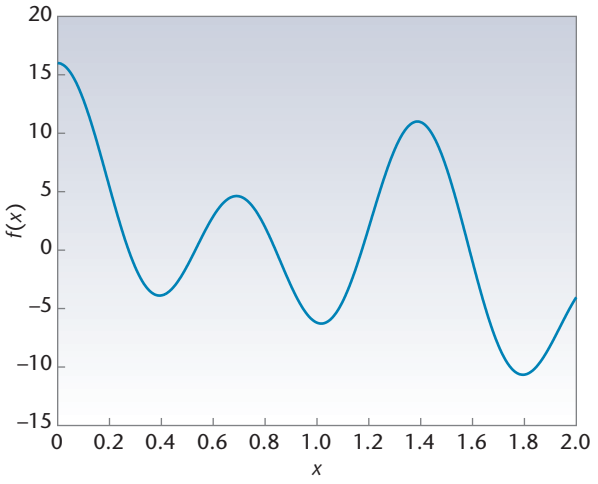

Figure 1. Nonconvex functions. A standard minimization algorithm might find the minimizer $x = 0.4$ if we start at $x = 0.5$. Monte Carlo minimization seeks the rightmost minimum, at $x = 1.8$, because it's the *global minimum*.

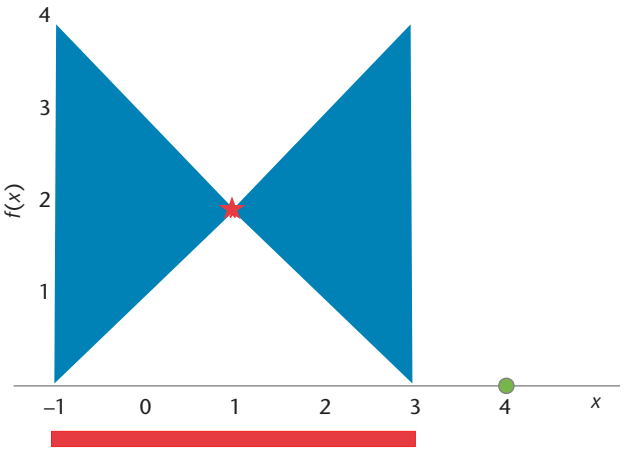

Figure 2. Lipschitz constant. If we know $f(1) = 2$ and are given a bound of 1 on the Lipschitz constant, then we know the function lies in the blue shaded region. Therefore, if we find that $f(4) = 0$, then we don't need to search the interval (–1, 3) for the minimizer of f.

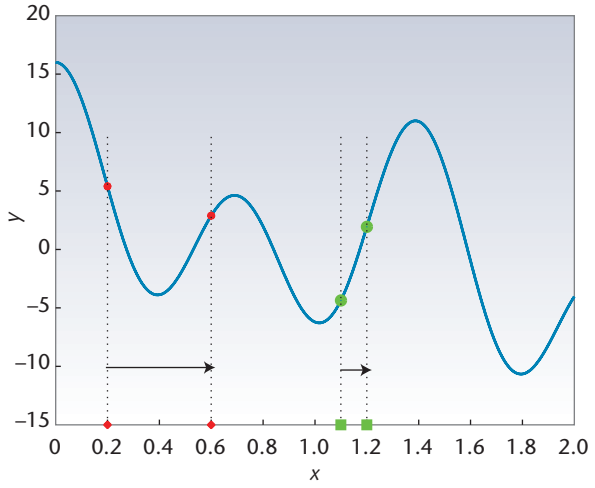

Figure 3. Simulated annealing. We sometimes allow moves that increase the function value, such as that illustrated by the shorter arrow, as well as moves that decrease the function value, such as the one from $x = 0.2$ to $x = 0.6$. At high temperature, uphill moves are likely to be accepted, whereas at low temperature, they're likely to be rejected. At high temperature, we might also allow larger moves, which can help us escape from valleys that don't contain the global minimum.

stuck in configurations that are local minimizers but not global ones. However, as the temperature decreases, it becomes less likely that a particle will make a move that gives an increase in energy, which lets the system do the fine-tuning needed to produce an optimal final configuration.

This motivates an algorithm: if we want to minimize some function other than energy, can we *simulate* the annealing process? Figure 3 illustrates the idea. We need some artificial definition of *temperature*, a means of generating a new configuration and deciding whether to keep it, and a *cooling schedule* for reducing the temperature. Algorithm 3 offers a way to implement this.

Algorithm 3. Simulated Annealing

Initialize the temperature T to some large number. Choose a number α between 0 and 1 and a small positive number ε.
while $T > \varepsilon$,
Generate a random move from the current $\mathbf{x}$ to a new $\hat{\mathbf{x}}$.
(Moves should be larger on average when the temperature is high.)
if $t \equiv f(\hat{\mathbf{x}}) - f(\mathbf{x}) < 0$, **then**
Our new point improves the function value and we accept the move, setting $\mathbf{x} = \hat{\mathbf{x}}$.
else
We accept the move with probability $e^{-t/T}$.
end
Decrease the temperature, replacing T by αT.
end

The simulated annealing algorithm has a long history. An idea in a paper Nicholas Metropolis coauthored in 1953 first inspired it, so the algorithm sometimes bears his name. The underlying idea for the Metropolis algorithm is called the Monte Carlo Markov Chain, or MCMC. Strictly speaking, simulated annealing isn't actually an MCMC method because it doesn't satisfy a technical condition called *detailed balance*, but it "feels" like MCMC.

Let's see how it works.

Problem 2.

Use the simulated annealing algorithm to minimize `myf.m`.

Although all three of our minimization algorithms are slow, they have two virtues: one, when compared to standard minimization algorithms, they give a better probability of finding a global rather than a local minimizer, and two, they don't require derivative values for f. In fact, they don't even require that f be differentiable.

Minimization of Discrete Functions

Many optimization problems are simple to state but difficult to solve—the traveling salesperson problem (TSP) is a good example. This person needs to visit n cities exactly once but also wants to minimize the total distance traveled and finish the trip back at the starting point. To solve the problem, we need to *permute* the list of cities to obtain an itinerary with the shortest total distance.

If, for definiteness, we specify the first city visited, then among $(n - 1)!$ permutations of the other cities, we want to choose the best. We have an enormous number of possibilities for moderate values of n, so it isn't practical to test each of them. Rather, we can find an approximate solution either by generating random permutations and choosing the best (a Monte Carlo algorithm) or by using simulated annealing.

One step of a simulated annealing algorithm for TSP might look like Algorithm 4.

Algorithm 4. Simulated Annealing for TSP

Start with an initial ordering of cities and an initial temperature T.
Randomly choose two cities.
if interchanging those cities decreases the length of the circuit **then**
Interchange them!

else

Interchange them with a probability that depends on the amount of increase and the current temperature.

end

Decrease the temperature.

The art of the method is in determining the probability function and the temperature sequence appropriate to the specific problem.

PROBLEM 3.

(a) Matlab provides a naive Monte Carlo solution algorithm for the TSP. Given an ordering of the cities, it randomly generates a pair of cities and interchanges them if this interchange lowers the total distance. Experiment with the demonstration program `travel.m` and display the program using `type travel` to see how this works.

(b) Write a program to solve a TSP using simulated annealing, and compare your algorithm with that used in part (a).

Monte Carlo Methods for Counting

Counting is fundamental to our civilization: it's something we learn as young children, so what could be so difficult about counting that would lead us to use Monte Carlo methods? Well it all depends on how many things we're asked to count and our ability to keep track of them! George Gamow recognized this in the title of his famous book, *One*, *Two*, *Three*…*Infinity*.

Choosing a random sample of reasonable size and using it to estimate the exact counts of a much larger population has a long history in science, but it's even more widely used in commerce and public policy: Monte Carlo samples are the basis for opinion polls, census methodologies, traffic volume surveys, and many other information estimates.

Consider Figure 4 and suppose it's the lattice of a crystal containing two types of molecules: *dimers*, which fill two adjacent sites, and *monomers*, which occupy only one site. Define $C(k)$ to be the number of distinct arrangements of k dimers. (Clearly $C(0) = 1$ and $C(k)$ is nonzero only if k is an integer between 0 and half the number of sites in the lattice.) This function $C(k)$ is related to the *partition function*, which is studied in statistical physics and quantum mechanics and from which we can derive many physical properties. In it, each and every arrangement of d dimers would have its own

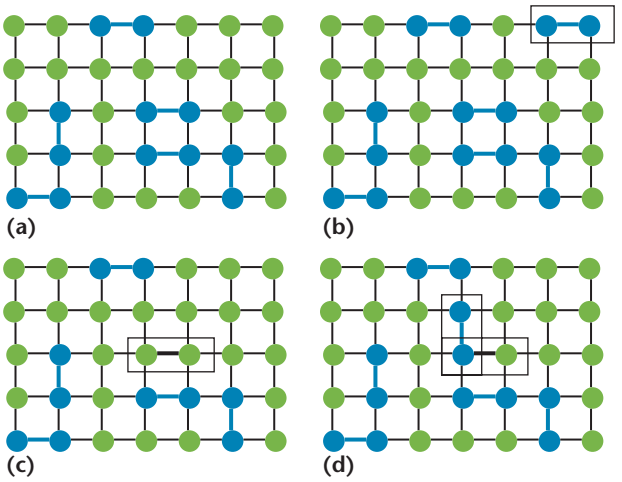

Figure 4. Crystal lattice. (a) A lattice of six dimers (blue) and 23 monomers (green), (b) the result of adding a dimer in row 1, (c) the result of deleting a dimer in row 3, and (d) the result of swapping a horizontally oriented dimer in row 3 for a vertical one.

associated energy $E(d)$ and would occur with probability $\exp(-E(d)/kT)$, where k is a physical constant and T is temperature. The partition function is $Z(T) = \sum\exp(-E(d)/kT)$, where the sum is over all arrangements. For the monomer–dimer problem, all arrangements of k dimers have the same energy, so we group them together and use $C(k)$ to count them.

For very small lattices, we can actually list and count all the arrangements. For larger lattices, though, the $C(k)$ aren't known exactly, and counting all the possibilities is too expensive. Instead, we can use Monte Carlo methods to estimate them. The basic idea is to obtain a random arrangement of dimers, assume that monomers occupy all the other sites, determine the number of dimers k, and add one to the count of occurrences of k dimers. We repeat this process for as long as we can.

One difficulty with this idea is in randomly generating acceptable arrangements of dimers in which the dimers don't overlap. To avoid generating mostly unacceptable arrangements, algorithms usually start with an acceptable arrangement and make a small change to it. The trouble with this is that the two arrangements are highly correlated, so our sample is no longer uniformly random. (It's a bit like deciding to take a survey of world opinion by talking to the first 100 people you meet on a single street corner in Washington, DC, at noon on a Wednesday.) To fix this deficiency for the dimer problem, we make many changes—to the point that the two arrangements are no longer correlated—before we record a count.

However, this still doesn't quite guarantee that we'll get good samples well spread over the set of arrangements. This is because many more arrangements have the lattice half-

TOOLS

For more background information on probability, consult a standard text.[1] Kirkpatrick, Gelatt, and Vecchi discuss simulated annealing in more detail in a *Science* article.[2]

Michael Heath[3] and Donald Knuth[4] give information on the generation of (pseudo-) random samples, as do http://random.mat.sbg.ac.at/literature/ and http://random.mat.sbg.ac.at/links/rando.html.

To test random number generators, consult Knuth's book or http://stat.fsu.edu/~geo/diehard.html.

Finally, you can find more information on random counting algorithms and the estimation of partition functions in papers by Claire Kenyon, Dana Randall, and Alistair Sinclair,[5] Isabel Beichl and Francis Sullivan,[6] Russel Caflisch,[7] and Beichl, Dianne O'Leary, and Sullivan.[8]

References

1. A.M. Mood and F.A. Graybill, *Introduction to the Theory of Statistics*, McGraw Hill, 1963.
2. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, 1983, pp. 671–680.
3. M. Heath, *Scientific Computing: An Introductory Survey*, 2nd ed., McGraw Hill, 2002.
4. D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, 1997.
5. C. Kenyon, D. Randall, and A. Sinclair, "Approximating the Number of Monomer–Dimer Coverings of a Lattice," *J. Statistical Physics*, vol. 83, nos. 3-4, 1996, pp. 637–659.
6. I. Beichl and F. Sullivan, "The Importance of Importance Sampling," *Computing in Science & Eng.*, vol. 1, no. 2, 1999, pp. 71–73.
7. R.E. Caflisch, "Monte Carlo and Quasi-Monte Carlo Methods," *Acta Numerica*, vol. 7, 1998, pp. 1–49.
8. I. Beichl, D.P. O'Leary, and F. Sullivan, "Approximating the Number of Monomer-Dimer Coverings in Periodic Lattices," *Physical Rev. E*, 2001; 64:016701.1–6.

covered with dimers than those that have zero or one dimer, so we'll probably see many copies of the zero-dimer arrangement before we generate enough arrangements to obtain a good estimate of the number of half-covered arrangements. Therefore, we must run the algorithm for a long time and deal with possible repetitions. We might find, for example, that we had zero dimers in five of our trials and three dimers in 52; because we know the lattice has only one arrangement of zero dimers, our best guess at the number of arrangements of three dimers is 52/5 = 10.4.

The KRS counting algorithm (named for its creators, Claire Kenyon, Dana Randall, and Alistair Sinclair) is based on this idea and uses three types of changes, as Figure 4 illustrates. The resulting method is Algorithm 5. We'll use the KRS algorithm to estimate $C(k)$ for some simple lattices.

ALGORITHM 5. KRS ALGORITHM

Start with a lattice filled with monomers and choose a probability α.

repeat

Choose an adjacent pair of sites.

If both sites have monomers, add this dimer with probability α.

If the sites are occupied by a single dimer, delete it with probability α.

If one site is occupied with a dimer, swap that dimer for this one with probability α.

If it has been long enough since our last recording, add the resulting configuration to the record.

until enough steps are completed

PROBLEM 4.

(a) Compute the partition function coefficients for a 2×2 lattice by explicit counting. Repeat for a 3×2 lattice. Consider the 4×4 case, and see why it becomes difficult to explicitly count the number of arrangements.

(b) Implement the KRS algorithm, and use it to estimate the partition function for a 4×4 lattice. Try various choices of probabilities and updating intervals. Repeat for a lattice as large as possible (perhaps 10×10).

Using Monte Carlo for minimization and counting is always expensive, but when carefully implemented, the resulting estimates can be very reliable. The "Tools" sidebar offers pointers for further reading.

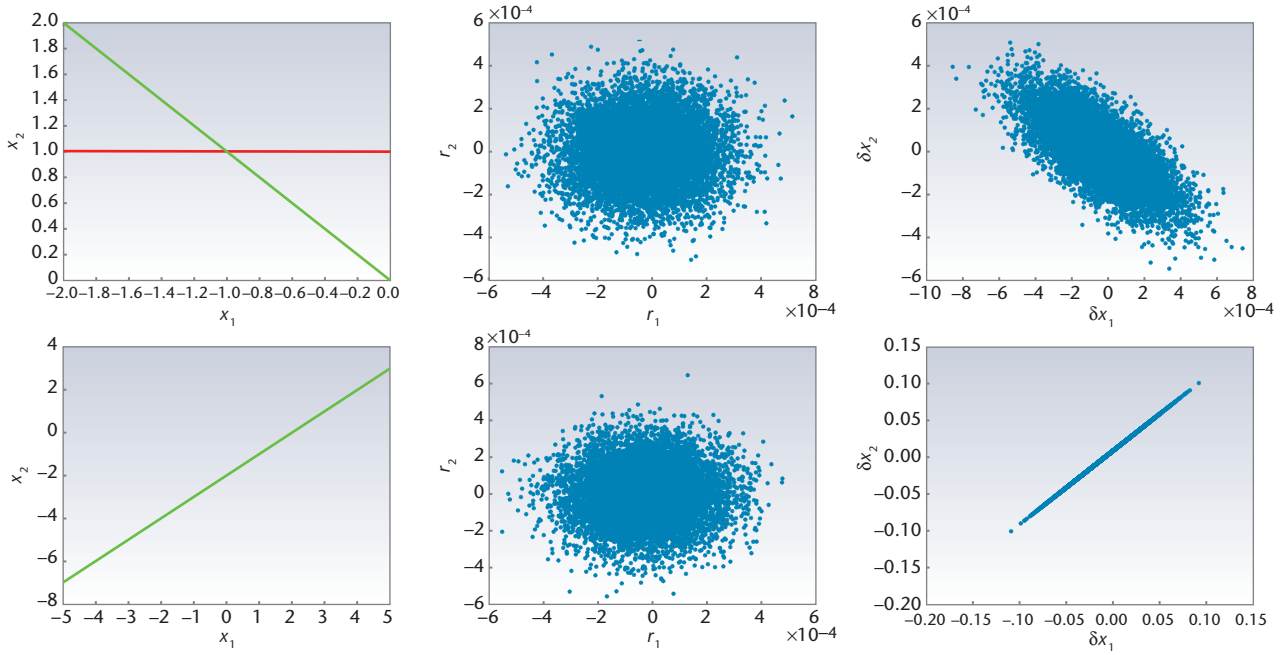

Figure A. Results of Problem 2. The top example is typical of well-conditioned problems, whereas the bottom example is ill-conditioned, so the solution is quite sensitive to noise in the data. The graphs at left plot the linear equations, those in the middle plot residuals to perturbed systems, and those at the right plot the corresponding solution vectors.

Partial Solution to Last Issue's Homework Assignment

SENSITIVITY ANALYSIS: WHEN A LITTLE MEANS A LOT

By Dianne P. O'Leary

In the last issue's installment of Your Homework Assignment, we investigated the use of several tools for *sensitivity analysis*.

PROBLEM 1.

Suppose x_1 and x_2 are the roots of the quadratic equation $x^2 + bx + c = 0$.

(a) Use implicit differentiation to compute dx/db.

(b) We know that the roots are

$$x_{1,2} = \frac{1}{2}\left(-b \pm \sqrt{b^2 - 4c}\right).$$

Differentiate this expression with respect to b, and show that the answer is equivalent to the one you obtained in (a).

(c) Find values of b and c for which the roots are very sensitive to small changes in b and values for which they aren't sensitive.

Answer:

(a) $2xdx + bdx + xdb = 0$, so

$$\frac{dx}{db} = -\frac{x}{2x + b}.$$

(b) Differentiating, we obtain

$$\frac{dx}{db} = -\frac{1}{2} \pm \frac{1}{2}\frac{b}{\sqrt{b^2 - 4c}}.$$

Substituting the roots $x_{1,2}$ into the expression obtained in (a) gives the same result as this.

(c) The roots will be most sensitive when the derivative is large, which occurs when the discriminant $\sqrt{b^2 - 4c}$ is almost zero, and the two roots almost coincide. In contrast, a root is insensitive when the derivative is close to zero. In this case, the root itself might be close to zero, so although the absolute change will be small, the relative change could be large.

PROBLEM 2.

Consider the linear system $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ with

$$\boldsymbol{A} = \begin{bmatrix} \delta & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

where $\delta = 0.002$.

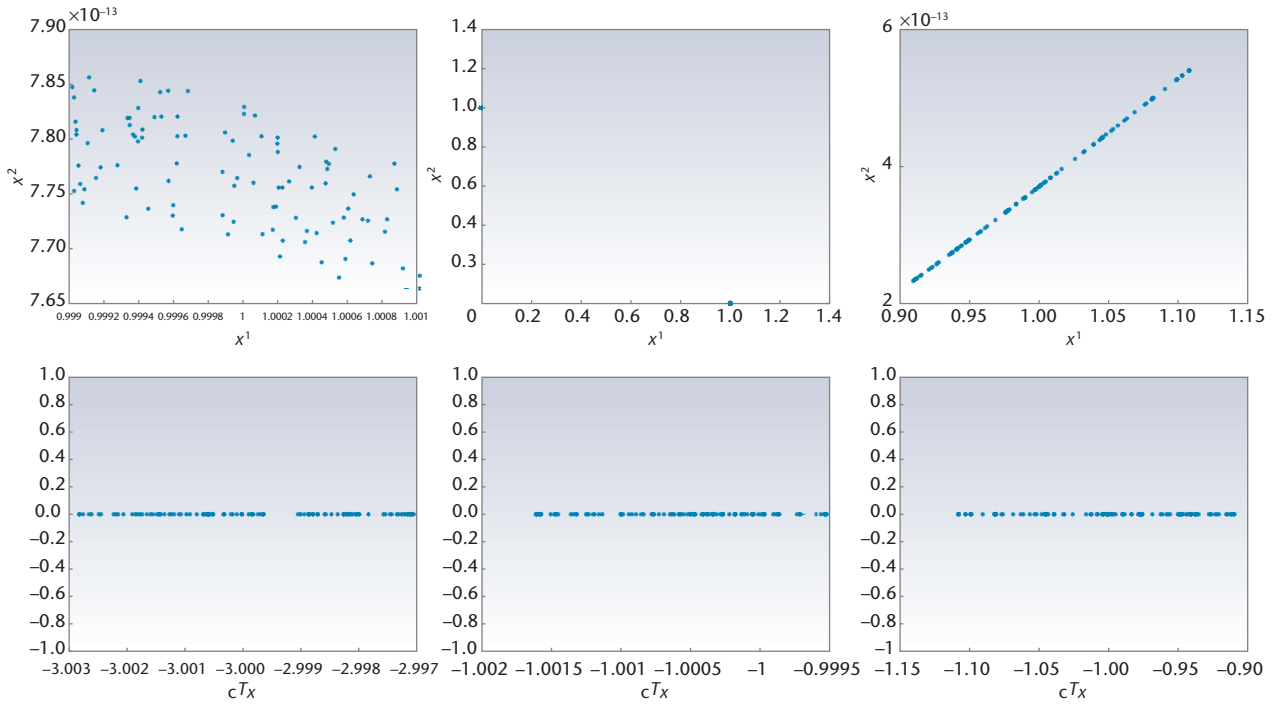

Figure B. Results of Problem 3. The graphs on the top plot the perturbed solutions to the three problems, whereas those on the bottom plot the optimal function values, which are increasingly sensitive to small changes in the data. Note the vastly different vertical scales in the graphs on the top.

(a) Plot the two equations defined by this system, and compute the condition number of A (`cond(A)`).

(b) Compute the solution $\mathbf{x}^*$ to $A\mathbf{x} = \mathbf{b}$, and also compute the solution to the nearby systems

$$(A + E^{(i)})\mathbf{x}^{(i)} = \mathbf{b}$$

for $i = 1, \ldots, 1{,}000$, where the elements of $E^{(i)}$ are normally distributed with mean 0 and standard deviation $\tau = .0001$. (You can do this by setting each `E = tau*randn(2,2)`.) Plot $\mathbf{x}^{(i)} - \mathbf{x}^*$. This plot reveals the forward error in using $(A + E^{(i)})$ as an approximation to A. In a separate figure, plot the residuals $\mathbf{b} - A\mathbf{x}^{(i)}$ (the backward error) for each computed solution.

(c) Repeat (a) and (b) with the linear system $A\mathbf{x} = \mathbf{b}$ with

$$A = \begin{bmatrix} 1+\delta & \delta-1 \\ \delta-1 & 1+\delta \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 2 \\ -2 \end{bmatrix}.$$

(d) Discuss your results. Why do the forward error plots for the two problems look so different? How does the condition number relate to what you see in the forward error plots? What do the backward error plots tell you?

Answer:

The solution is given in `exlinsys.m` (found on the Web site; http://computer.org/cise/homework), and Figure A shows the results. From the graphs on the left, we see that if we "wiggle" the first linear system's coefficients a little bit, then the intersection of the two lines doesn't change much; in contrast, because the two equations for the second linear system almost coincide, small changes in the coefficients can move the intersection point a great deal.

The middle graphs show that despite this sensitivity, the solutions to the perturbed systems satisfy the original systems quite well—to within 5×10^{-4} residuals. This means that the *backward error* in the solution is small; we've solved a nearby problem $\mathbf{x} = \mathbf{b} + \mathbf{r}$ in which the norm of $\mathbf{r}$ is small. This is characteristic of Gauss elimination, even on ill-conditioned problems.

The graphs on the right are quite different, though. The changes in the solutions $\mathbf{x}$ for the first system are all of the same order as the residuals, but for the second system, they're nearly 500 times as big as the perturbation. Note that for the well-conditioned problem, the solutions give a rather circular cloud of points, whereas the ill-conditioned problem has a direction, corresponding to the right singular vector for the small singular value, in which large perturbations occur.

The condition number of the matrix captures this behavior; it's about 2.65 for the first matrix and 500 for the second, so we expect changes in the right-hand side for the second problem might produce a relative change 500 times as big in the solution $\mathbf{x}$.

PROBLEM 3.

Investigate the sensitivity of the *linear programming problem*

$$\min_{\mathbf{x}} \mathbf{c}^T x$$

subject to

$$\boldsymbol{A}\mathbf{x} \le b, \quad x_1 \ge 0, \quad x_2 \ge 0.$$

(a) Let $\boldsymbol{A} = [1, 1]$, $b = 1$, and $\mathbf{c}^T = [-3, -1]$. Solve the linear program (using, for instance, Matlab's `linprog`) and use the Lagrange multipliers (also called *dual variables*) to evaluate the sensitivity of $\mathbf{c}^T\mathbf{x}$ to small changes in the constraints. Illustrate this sensitivity using a Monte Carlo experiment, solving 100 problems with $\boldsymbol{A}$ replaced by $\boldsymbol{A} + \mathbf{E}^{(i)}$, where the elements of $\boldsymbol{E}^{(i)}$ are uniformly distributed on the interval $[-\tau, \tau]$, with $\tau = 0.001$. (You can do this by setting each `E = tau*(rand(1,2)-.5)`.) Plot all the solutions in one figure and all the function values $\mathbf{c}^T\mathbf{x}$ in another.

Repeat for two more examples:

(b) $\boldsymbol{A} = [1, 1]$, $b = 1$, $\mathbf{c}^T = [-1.0005, -0.9995]$.

(c) $\boldsymbol{A} = [0.01, 5]$, $b = 0.01$, $\mathbf{c}^T = [-1, 0]$.

Explain how the Lagrange multiplier for the constraint $\boldsymbol{A}\mathbf{x} \le \mathbf{b}$ gives insight into the sensitivity observed in the corresponding Monte Carlo experiment.

Answer:

The solution is given in `exlinpro.m`, and Figure B shows the results. For the first example, the Lagrange multiplier predicts that the change in $\mathbf{c}^T\mathbf{x}$ should be roughly three times the size of the perturbation, which the Monte Carlo experiments confirm. The Lagrange multipliers for the other two examples (1.001 and 100, respectively) are also good predictors of the change in the function value. But note that something odd happens in the second example. Although the *function value* isn't very sensitive to perturbations, the *solution vector* is quite sensitive: it's sometimes close to [0, 1] and sometimes close to [1, 0]! The solution to a (nondegenerate) linear programming problem must occur at a vertex of the feasible set. In our unperturbed problem, we have three vertices: [0, 1], [1, 0], and [0, 0]. Because the gradient of $\mathbf{c}^T\mathbf{x}$ is almost parallel to the constraint $\boldsymbol{A}\mathbf{x} \le \mathbf{b}$, we sometimes find the solution at the first vertex and sometimes at the second.

Therefore, in optimization problems, even if the function value is relatively stable, we might encounter situations in which the solution parameters have very large changes.

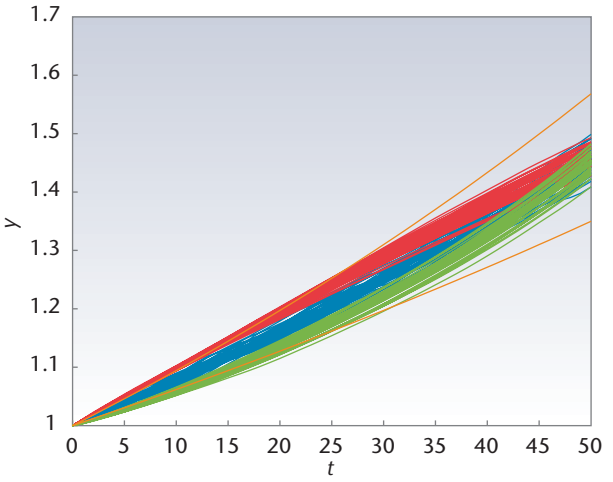

Figure C. Results of Problem 4. The black curves result from setting $a = 0.006$ and $a = 0.009$. The blue curves have random rates chosen for each year, and the red curves are the results of trials with the random rates ordered from largest to smallest. For the green curves, the rates were ordered smallest to largest.

PROBLEM 4.

Consider the very simple differential equation for $y(t)$:

$$y' = ay,$$

where $y(0) = 1$ and $a(t)$ is given. Let's investigate the equation's sensitivity to our knowledge of a.

To make the problem concrete, we can divide the US population growth rate into two pieces: a rate of 0.006 due to births and deaths, and a rate of 0.003 due to migration. Determine how much the population will increase over the next 50 years if this rate stays constant, and how much it will increase if we set migration to zero. Then perform Monte Carlo experiments, assuming that the birth/death rate is normally distributed with mean 0.006 and standard deviation 0.001, and the migration rate is uniformly distributed between 0 and 0.003. Also experiment with what happens if years of high growth rate come early, followed by years of low growth rate, and vice versa. Plot and discuss the results.

Answer:

The results are computed by `exode.m` and shown in Figure C. The Monte Carlo results predict that growth is likely to be between 1.4 and 1.5. The two black curves produced under the assumption that the rates are known exactly give very pessimistic upper and lower bounds on the growth: 1.35 and 1.57, respectively. This is typical of *forward error bounds*. Notice that the solution is the product of exponentials,

$$y(50) = \prod_{\tau=1}^{\tau=49} e^{a(\tau)},$$

where $a(\tau)$ is the growth rate in year τ. Because exponentials commute, the final population is invariant with respect to the ordering of the rates, but the intermediate population (and thus the demand for social services and other resources) is quite different under the two assumptions.

PROBLEM 5.

Using the two linear systems from Problem 2, perform a Monte Carlo experiment that computes the solution to the nearby systems

$$A\mathbf{x}^{(i)} = \mathbf{b} + \mathbf{e}$$

for $i = 1, \ldots, 1{,}000$, where the elements of $\mathbf{e}$ are normally distributed with mean 0 and standard deviation $\tau = 0.0001$. Compute 95 percent confidence intervals on each component of the solution to the two linear systems, and see how many of the components of the Monte Carlo samples lie within the confidence limits.

Answer:

The solution is given in `exlinsys.m`. The confidence intervals for the first example are $x_1 \in [-1.0228, -1.0017]$, $x_2 \in [1.0018, 1.0022]$ and for the second example are $x_1 \in [0.965, 1.035]$, $x_2 \in [-1.035, -0.965]$. Those for the second example are 20 times larger than for the first because they're related to the size of A^{-1}, but in both cases roughly 95 percent of the samples lie within the intervals, as expected.

Remember that these intervals should be calculated using a Cholesky factorization or the backslash operator. Using `inv` or raising a matrix to the –1 power is slower when n is large and generally less accurate.

Disclaimer

Mention of commercial products does not imply endorsement by NIST.

Isabel Beichl is a mathematician in the Information Technology Laboratory at the National Institute of Standards and Technology. Contact her at isabel.beichl@nist.gov.

Dianne P. O'Leary is a professor of computer science and a faculty member in the Institute for Advanced Computer Studies and the Applied Mathematics Program at the University of Maryland. She has a BS in mathematics from Purdue University and a PhD in computer science from Stanford. O'Leary is a member of SIAM, the ACM, and AWM. Contact her at oleary@cs.umd.edu; www.cs.umd.edu/users/oleary/.

Francis Sullivan is the director of the IDA Center for Computing Sciences in Bowie, Maryland. From 2000 through 2004, he served as *CiSE* magazine's editor in chief. Contact him at fran@super.org.