Lecture 11

Simulated Annealing

Contents

1 Introduction

Optimization is a common problem in science and engineering. It involves minimization or maximization of a function of one or more variables.

Suppose the "cost" function to be minimized is $E(\mathbf{s})$, where $\mathbf{s} = (s_1, s_2, ..., s_N)$ is a point in the multidimensional space.[1] We seek to find the value of $\mathbf{s}$ that minimizes $E(\mathbf{s})$.

Mathematically, this can be written as,

$$\arg\min_{\mathbf{s}} E(\mathbf{s}). \tag{1}$$

Most gradient-based minimization methods yield a local minimum that is close to the starting point.

Simulated annealing (SA) is a meta-heuristic MCMC method for global minimization designed to explore and overcome many local minima.

It allows tremendous flexibility in specifying constraints.

2 Algorithm

The underlying idea of SA is to treat the cost function to be minimized as the reciprocal of the PDF. It can then be explored using standard MCMC techniques.

- generate new trial point based on current point
- evaluate the cost function at the two locations
- accept new value (with Metropolis or MH criterion) if it lowers cost

A key idea in SA is a fictitious "**temperature**". This is the origin of the label "annealing" in SA.

SA is fashioned after crystal formation from molten materials by cooling. If the rate of cooling is gradual, molecules move freely at high temperature (explore space), and slide into low-energy crystal lattices (get stuck in minimas) at low temperatures as the material solidifies.[2]

In SA, at high temperatures, we explore parameter space jumping over energy barriers; at lower temperatures, exploration is restricted. Our cost function (eqn 1) is thus analogous to energy.

The **basic algorithm** is as follows:

(i) Start with an initial guess $\mathbf{s}_0$ and a high temperature T_0.

(ii) Consider a gradually decreasing sequence of temperatures T_i, $i = 0, 1, \cdots, M$.

(iii) At each temperature T_i, perform Metropolis MCMC:

- propose an update and evaluate function
- preferentially accept updates that improve solution

$$a(\mathbf{s} \to \mathbf{s}') = \min \left\{ 1, \exp\left(-\frac{\Delta E}{T_i} \right) \right\},$$

where $\Delta E = E(\mathbf{s}') - E(\mathbf{s})$.

[1]Fortunately, maximization of $E(\mathbf{s})$ is identical to minimization of $-E(\mathbf{s})$. Thus, numerically, we don't need separate algorithms for minimization and maximization. Without loss of generality, we shall consider a minimization problem here.

[2]Note that "annealing" is different from "quenching" in which the temperature is decreased very rapidly. Quenching freezes the structure in a high-energy state.

Initially, when $T \gg \Delta E$, $\exp(-\Delta E/T) \approx 1$. Thus, the acceptance probability $a(\mathbf{s} \to \mathbf{s}') \approx 1$, and most proposals are accepted. High temperatures allow us to climb over "hills" in the cost function.

If cooling is sufficiently slow, the global minimum will be reached. However, if we cool too slowly, it could take us forever! Our goal is to choreograph the annealing process so that we can solve the problem in a reasonable time period.

This means we have to choose some parameters smartly.

2.1 Parameter Settings

Typically, we have to make three important choices that differ from problem to problem.

(i) initial temperature

(ii) final temperature

(iii) rate of temperature decrease

These choices are in addition to the typical Metropolis MCMC choices, such as the form of the proposal function, starting points etc.[3] Let us consider these three parameters in turn.

2.1.1 Initial Temperature

The initial temperature T_0 is problem-specific. One rule of thumb suggests that we should target a T_0 that results in an average initial acceptance probability of $a_0(\Delta E > 0) \approx 0.8$.[4]

This rule of thumb ensures that we are accepting most moves initially improving our ability to dig out of local minima. The target of 80% calibrates the temperature to the specific problem, and ensures we do not start too hot and prolong the cooling process.

So how do we go about finding T_0 practically?

T_0 can be estimated by an initial exploratory search in which all *increases* in the cost function E are accepted. Set T_0 to something very large (say, the floating point maximum $\sim 10^{300}$), and calculate the average increase in the cost function,

$$\overline{\Delta E}^{+} = \frac{1}{n} \sum_{\Delta E > 0} \Delta E, \tag{2}$$

where n is the number of terms in the summation. Note that the summation is only over moves that increase E.

Using $a_0 = 0.8 = \exp(-\overline{\Delta E}^{+}/T_0)$, for production run set,

$$T_0 = -\frac{\overline{\Delta E}^{+}}{\ln a_0} \tag{3}$$

[3]In some sense, the optimization problem is simpler than the general sampling problem in that our goal is not to produce a set of samples $\mathbf{s}$ that follow the distribution $\pi(\mathbf{s}) = \exp(-\Delta E(s)/T$. All we care about is the maxima of $\pi(\mathbf{s})$, which corresponds to the minima of $E(\mathbf{s})$.

[4]Kirkpatrick *et al.*, Optimization by Simulated Annealing, *Science*, **220**(4598), 671-680, **1983**.

2.1.2 Final Temperature

The final temperature cam be set during runtime using some sort of convergence criterion.

We stop when the simulation ceases to make further progress. "Lack of progress" can be operationalized as,

- the cost function stops decreasing further
- acceptance ratio falls below a certain threshold
- some combination of the two
- Or a certain (fixed) total number of steps at a specified cooling rate.

2.1.3 Cooling Schedule

After every L trials (corresponding to $\approx 10 - 100$ MCS), the temperature may be decreased *exponentially*,

$$T_{k+1} = \alpha T_k, \tag{4}$$

with $\alpha \approx 1$. Often, $\alpha = 0.95 - 0.999$, so that the cooling is gradual.

It may also be decreased *linearly*. For example, after every L trials we decrease the temperature by a fixed amount,

$$T_{k+1} = T_k - \Delta T. \tag{5}$$

In either case, α and ΔT control the rate of cooling.

The guiding principle is simple: we want to pick the slowest cooling rate that allows us to perform the calculation in a reasonable time.

2.1.4 Reseeding

What is reseeding?

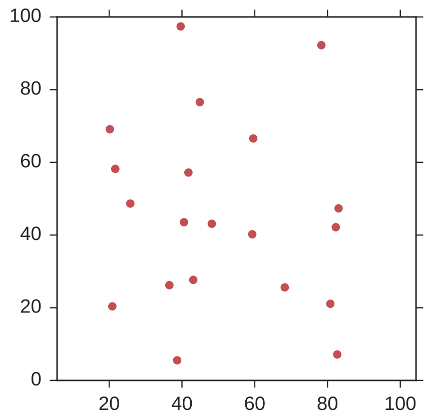
It is the act of going back to the best solution found so far, and resuming the calculation. In many practical simulations, reseeding is necessary.

In our algorithm, we shall choose to reseed with a small probability.

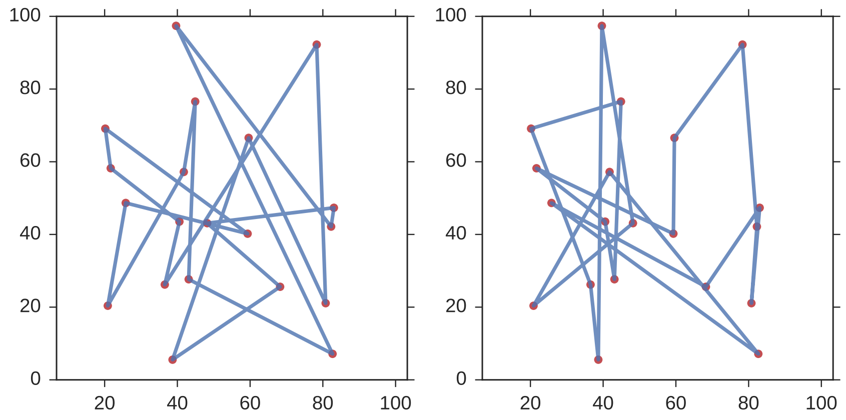
3 Example: Traveling Salesman

We shall now consider a specific example to demonstrate simulated annealing. This example is the famous traveling salesman problem.

Suppose there are $Q = 20$ cities on a 100 by 100 grid. You are a salesman, who wants to pay a sales visit to all the Q cities, and return to the one you started from.



You want to come up with an itinerary that minimizes the distance you have to travel. For example, consider two possible routes connecting these cities as shown below.



The total distance traveled is 856.6 (left) and 813.4 (right).

Let's formalize things a little bit, and state the problem mathematically.

Let $\mathbf{c}_i$ denote the co-ordinates of city i, for $i = 1, 2, \cdots, Q$. Let d_{ij} denote the (Euclidean) distance between two cities i and j:

$$d_{ij} = ||\mathbf{c}_i - \mathbf{c}_j||$$

Let a path be some permutation of the vector $[1, 2, ..., Q]$. For the two paths considered in the figure above,

$$p_{\text{left}} = [1, 12, 13, ..., 19, 8, 1]$$

$$p_{\text{right}} = [4, 15, 6, ..., 20, 13, 4]$$

Note that there are $Q + 1$ elements in a path, since the last element involves returning to the first city.[5] The total distance traversed by a given path is,

$$E = \sum_{j=1}^{Q} d_{p_j, p_{j+1}} \tag{6}$$

In this example, the cost function is the total distance given by eqn 6. We can code this up as a Python function as follows:[6]

[5]A path $p = [p_1, p_2, \cdots, p_Q, p_1]$, i.e. $p_{Q+1} = p_1$.

[6]Python arrays are C-style; indexing starts from 0.

```python
def E(p, c):
    """given a sequence p, and city coordinates c,
       returns the total distance traversed"""
    d = 0.
    for j in range(len(p)-1):
        d += np.linalg.norm((c[p[j],:] - c[p[j+1],:]))
    return d
```

3.1 Proposal Design

We want to find a path p that minimizes the total distance traversed E. Let's think a little bit about setting up efficient proposal moves.

A trial move involves taking the current path p and modifying it to suggest a new path p'. One way to build p' from p may be to simply **swap a pair of random selected elements**.

Furthermore, since the path is cyclic, without any loss of generality, we can simply swap an **internal** pair of elements. This is the strategy we will adopt.

```python
def proposal(oldp):
    """Given oldp, pick a pair of unique internal
       nodes i and j to swap"""

    N = len(oldp)                  # N = Q + 1
    i = np.random.randint(1, N-1)  # pick an internal node
    j = i                          # initialize j; just a placeholder

    # randomly pick a unique partner node
    while j == i or oldp[i] == oldp[j]:
        j = np.random.randint(1, N-1)

    return i, j
```

Let's test this using an example with $Q = 8$ cities.

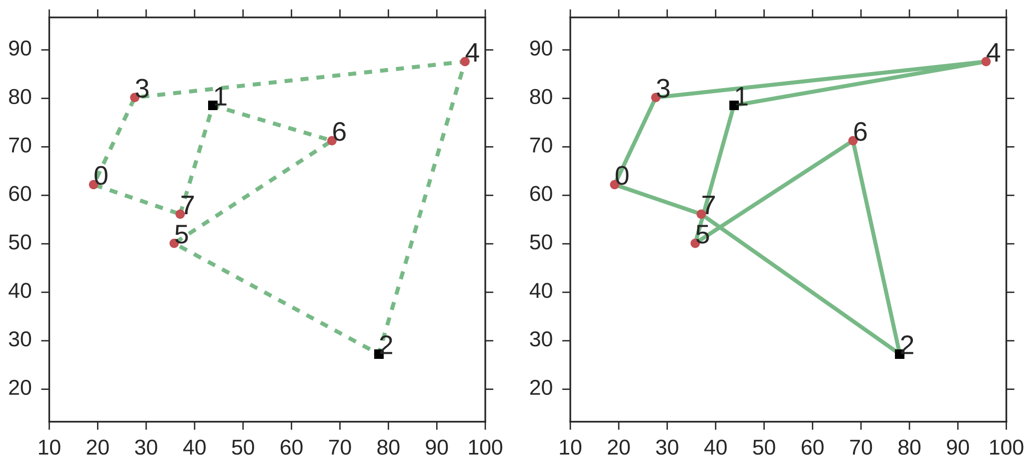
```python
# Location of the cities
Q = 8; c = np.random.uniform(0., 100., size=(Q,2))

# Select a cyclic path
p = np.random.permutation(Q); p = np.append(p, p[0])
[7 1 6 5 2 4 3 0 7]

# Pick locations to swap (here 5th and 2nd elements)
i,j = proposal(p)
4, 1

# Generate new path pn (copy old path p and swap)
pn = p.copy(); pn[i] = p[j]; pn[j] = p[i]
[7 2 6 5 1 4 3 0 7]
```

The picture below visualizes the proposed swap.



p = [7 $\underline{1}$ 6 5 $\underline{2}$ 4 3 0 7]   pn = [7 $\underline{2}$ 6 5 $\underline{1}$ 4 3 0 7]

3.1.1 Observations and Efficiency

As you can see, most of the connections don't change. We don't have to recompute the total E starting from scratch, since most of the distances in the sum are unchanged.

$$E = \sum_{j=1}^{Q} d_{p_j, p_{j+1}}$$

In this case, before the swap, `p = [7 1 6 5 2 4 3 0 7]`:

$$E_{\text{old}} = {\color{blue} d_{71} + d_{16}} + d_{65} + {\color{blue} d_{52} + d_{24}} + d_{43} + d_{30} + d_{07}.$$

After the proposed swap, `pn = [7 2 6 5 1 4 3 0 7]`

$$E_{\text{new}} = {\color{red} d_{72} + d_{26}} + d_{65} + {\color{red} d_{51} + d_{14}} + d_{43} + d_{30} + d_{07}.$$

In general, only 4 distance pairs are altered, **independent of** Q. This is especially useful when Q is large. There are Q terms in the summation (eqn 6); $Q - 4$ terms are not affected by the swap.

Dwelling on the problem, we recognize that distances between cities are computed over and over again. This might be another source of efficiency.

For example, we could pre-compute distances and build a lookup table of inter-city distances.

```python
def CreateDistanceTable(c):
    """Given city coordinates c, create Q by Q matrix K,
       with pairwise distances"""
    Q = len(c)
    K = np.zeros((Q, Q))

    for i in range(Q):
        for j in range(Q):
            K[i,j] = np.linalg.norm((c[i,:] - c[j,:]))
    return K
```

With these two optimizations, we can compute $\Delta E = E_{new} - E_{old}$ relatively cheaply

```
def dE(p, i, j, K):
    """change in cost function due to a swap (i,j) of path p
       K is the matrix (lookup table)"""
    ip = j; jp = i

    Enew = K[p[i-1], p[ip]] + K[p[ip], p[i+1]] +
           K[p[j-1],p[jp]]  + K[p[jp],p[j+1]]
    Eold = K[p[i-1], p[i]]  + K[p[i], p[i+1]]  +
           K[p[j-1],p[j]]   + K[p[j],p[j+1]]
    return Enew - Eold
```

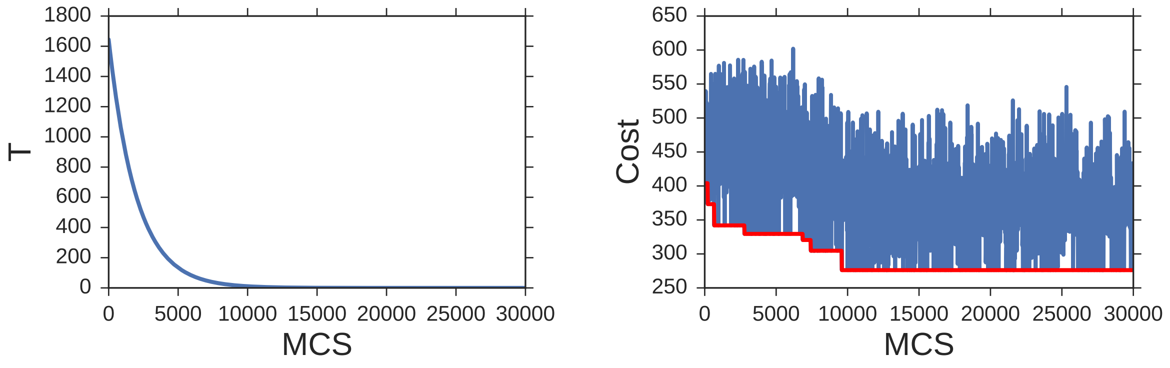
In addition to the current path, we will also store the best path explored until now p^*.

Now that we've done the most important legwork, let's do the other Monte-Carlo routines: set initial temperature, acceptance, and driver routines. These are included in sec. A.1

We are now ready to test the code for (say) $Q = 10$ cities.

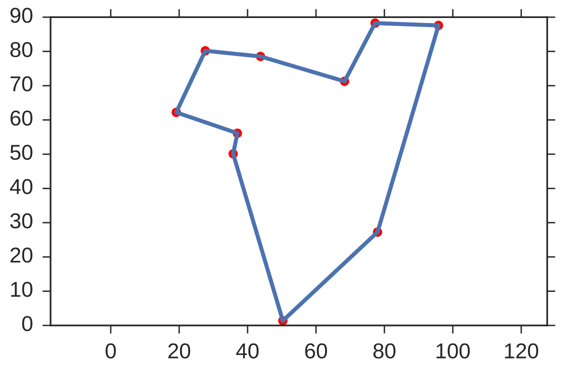
```
Q = 10
c = np.random.uniform(0., 100., 2*Q)
c = c.reshape((Q,2))
pstr, cstr = driver(c, 0.995, 30000)
```

The cooling schedule and the cost function are presented in the picture below.



The overall acceptance ratio is found to be 0.49. The red line in the subfigure on the right tracks the best solution p^*, and the blue lines show the "current" cost.

In this case, the best solution is shown in the next figure.



4 Perspective

SA is guaranteed to converge to the global minimum if T_0 is large enough, and the cooling rate is slow enough. This is reassuring, but doesn't tell us much about how to pick algorithm parameters to converge in a reasonable period of time.

In adaptive simulated annealing, the algorithm parameters that control temperature schedule and random step selection are automatically adjusted according to algorithm progress.

This potentially makes the algorithm more efficient and less sensitive to user defined parameters than vanilla SA.

In any event, it is advisable to try SA using multiple starting points to ensure that the global minimum has been sampled.

4.1 Resources

A valuable free resource on TSP is open-access paper on TSP by Rajesh Matai, Surya Prakash Singh and Murari Lal Mittal.

Implementations of SA are available in most compiled and interpreted languages. Matlab has a sophisticated SA optimizer in its Global Optimization Toolbox. Scipy optimize has a annealing-type algorithm scipy.optimize.basinhopping), which should be used in place of the deprecated `scipy.optimize.anneal`. Lester Ingber wrote C code for adaptive SA. A C++ implementation called CISMM is available on Github. SIMANN is a Fortran 77 code for adaptive sampling written by Bill Goffe that is archived on NETLIB.

5 Problems

5.1 Thought Questions

(i) Suppose you are minimizing a 2D function $f(x_1, x_2)$. Where would you incorporate constraints (such as $x_1, x_2 > 0$, or $x_1 < x_2^2$ or $-1 \leq x_1 \leq 1$ etc.) into the simulated annealing algorithm?

5.2 Numerical Problems

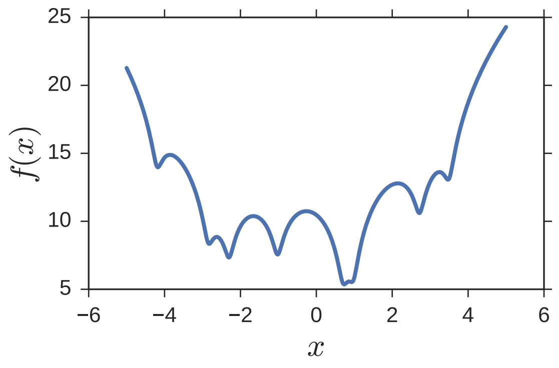
(i) Multiple Minima

Consider the function,[7]

$$f(x) = \sum_{i=1}^{8} \log\left(0.01 + (\alpha_i - x)^2\right),$$

with $\boldsymbol{\alpha} = \{-4.2, -2.85, -2.30, -1.02, 0.70, 0.98, 2.72, 3.50\}$.

[7] Brooks and Morgan, **1995**



(ii) Maximization

Find the maxima of the function,[8]

$$f(x) = (\cos 50x + \sin 20x)^2$$

on $x \in [0, 1]$.

(iii) 2D Minimization[9]

Use SA to find the minima of the 2D function,

$$f(x, y) = x^2 + 2y^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.7, \quad -1 \le x, y \le 1.$$

The function is symmetric, and has a number of minima and saddle-points. The true minima is at (0, 0).

A Appendix

A.1 Python Code for Traveling Salesman

All the relevant code for solving the problem are included below. We start with the driver, which takes in the location of the cities, a cooling schedule parameter (α), and the total number of steps. The parameter L together with α control the rate of cooling together.

```python
def driver(c, alpha=0.995, nsteps=10000, L=10):

    # initialization
    Q    = len(c)
    L    = 10    # change temperature


        # setup initial temperature
    T0   = EstimateT0(c, nsteps=1000, a0=0.8)
    Temp = T0

    # starting position
    p    = np.random.permutation(Q);
```

[8]Robert and Casella
[9]from Bohachevsky et al., Technometrics, 28, 209-217, **1986**.

```
    p    = np.append(p,p[0])
    cost = E(p, c)


    # lookup table
    K    = CreateDistanceTable(c)

    # best solution
    pstr = p.copy()
    cstr = cost
    numSucc = 0

    # write a log-files of samples
    f = open('log.dat','w')

    # main loop
    for iMCS in range(nsteps):

        i, j = proposal(p)
        acc, p, dEng = acceptMetro(p, i, j, K, Temp)

        if acc:
            cost = cost + dEng
            numSucc += 1

        if iMCS % L == 0:  # time to change Temp
            cost = E(p,c)
            if cost < cstr: # update best?
                cstr = cost; pstr = p
            if np.random.rand() < 0.05: # reseed?
                p    = pstr; cost = cstr

            f.write("{0:d} {1:8.3e} {2:8.2f} {3:8.2f}\n"
                    .format(iMCS, Temp, cost, cstr))
            Temp = Temp * alpha
    f.close()

    print("acceptance ratio =", float(numSucc)/nsteps)
    return pstr, cstr
```

The MCMC-like proposal and acceptance functions are at the heart of the algorithm.

```
def proposal(oldp):
    """Given oldp, pick a pair of unique internal
       nodes i and j to swap"""

    N = len(oldp)                   # N = Q + 1
    i = np.random.randint(1, N-1) # pick an internal node
    j = i                           # initialize j; just a placeholder

    # randomly pick a unique partner node
    while j == i or oldp[i] == oldp[j]:
        j = np.random.randint(1, N-1)

    return i, j

def acceptMetro(pold, i, j, K, Temp):
    """accept or reject proposed swap?"""

    accept = False
    dEnerg = dE(pold, i, j, K)
    ratio  = -dEnerg/Temp

    if ratio > 0.:
        accept = True
    elif np.random.rand() < np.exp(ratio):
        accept = True

    # keeping track of dEnergy for efficiency
    if accept:
        pnew = pold.copy()
        pnew[i] = pold[j]; pnew[j] = pold[i]

        return accept, pnew, dEnerg
    else:
        return accept, pold, dEnerg
```

The routine to estimate the initial temperature T_0 is next. Currently, the driver routine is not optimized since the distance table is created in two places (`EstimateT0` and `driver`). This can be easily refactored for efficiency (exercise).

```
def EstimateT0(c, nsteps=1000, a0=0.8):

    Q    = len(c)
    Temp = 1.0e300
    p    = np.random.permutation(Q); p = np.append(p,p[0])
    K    = CreateDistanceTable(c)

    cost = E(p, c)

    dEp  = 0.
    num  = 0

    for iMCS in range(nsteps):

        i, j = proposal(p)
        acc, p, dEng = acceptMetro(p, i, j, K, Temp)

        if dEng > 0.:
            dEp += dEngmay
            num += 1

    dEp = dEp/num

    # factor 10 for safety
    T0 = 10.0 * -dEp/np.log(a0)

    return T0
```

The functions to return energy (complete calculation) and change in energy due to a proposal are next.

```
def E(p, c):
    """given a sequence p, and city coordinates c,
       returns the total distance traversed"""
    d = 0.
    for j in range(len(p)-1):
        d += np.linalg.norm((c[p[j],:] - c[p[j+1],:]))
    return d

def dE(p, i, j, K):
    """change in cost function due to a swap (i,j) of path p
       K is the matrix (lookup table)"""
    ip = j; jp = i

    Enew = K[p[i-1], p[ip]] + K[p[ip], p[i+1]] +
           K[p[j-1],p[jp]]  + K[p[jp],p[j+1]]
    Eold = K[p[i-1], p[i]]  + K[p[i], p[i+1]]  +
           K[p[j-1],p[j]]   + K[p[j],p[j+1]]
    return Enew - Eold
```

And finally, the routine to create the table of distances.

```
def CreateDistanceTable(c):
    """Given city coordinates c, create Q by Q matrix K,
       with pairwise distances"""
    Q = len(c)
    K = np.zeros((Q, Q))

    for i in range(Q):
        for j in range(Q):
            K[i,j] = np.linalg.norm((c[i,:] - c[j,:]))
    return K
```