Chapter 1

Optimization problem

Optimization problems are abundant in our daily lives, as we each have a set of goals and finite resources which should be optimally allocated. Of all these resources, time is always in demand, as there are natural bounds on our time resources. We are limited to 24 hours in a day, yet we need to allocate time for sleep, studies, work, rest, and multiple other tasks. We are always facing a question: should we sleep an extra hour to be rested before work or should we instead read an interesting book? Everyone has a different solution, but the problem is the same: how to optimize the distribution of the available resources to get a maximum outcome. In this chapter, we will cover several typical optimization problems and common methods to solve them. But it should be said up front: **there is no guaranteed way to find the global optimal point, i.e. the very best, at finite time in a general case.**

1.1 Introduction to optimization

Before we begin, we will start with a formal mathematical definition of the optimization problem.

The optimization problem

Find $\vec{x}$ that minimizes $E(\vec{x})$ subject to $g(\vec{x}) = 0$ and $h(\vec{x}) \leq 0$, where
$\vec{x}$ is the vector of independent variables;
$E(\vec{x})$ is the energy function, which sometimes is also called: objective or fitness or merit function;
$g(\vec{x})$ and $h(\vec{x})$ are constraining functions.

As you can see, we solve the optimization towards a minimum, i.e. the minimization problem in computer science. It is easy to see that maximization problems are the same as

minimization, once we change $E(\vec{x}) \to -E(\vec{x})$.

For a physicist, it is clear why the minimized function is called energy. We are looking for the minimum, or lowest point, which is the point on the potential energy landscape where a physical system tends to arrive (in the presence of the dissipative forces), i.e. nature constantly solves the minimization of energy problem.

The constraining functions are dealing with some additional dependencies among the components of the $\vec{x}$. If we have a budget of 100 dollars per day, we allocate certain amounts for food (x_1), books (x_2), movies (x_3), and clothes (x_4). Our final goal is to maximize overall happiness or, since we are doing minimization, to minimize unhappiness. Everyone has a different merit function which depends on the above parameters, although there is an obvious common constraint: we cannot spend more than \$100, so $h(\vec{x}) = (x_1+x_2+x_3+x_4)-100 \leq 0$. The constraining function can take the form of conditional statements or set a limit for only a few parameters. For example, we can say that we need to spend at least some amount of money on food. There are also unconstrained problems where any value of $\vec{x}$ is permitted.

1.2 One dimensional optimization

At first, we consider the one dimensional optimization problem, i.e. the ($\vec{x}$) has only one component. In this case, we can drop the vector notation and just use x for the independent variable as x is one dimensional. Suppose that dependence of our merit or energy function (E) on x looks as shown in the fig. 1.1. If we know the analytical expression for $E(x)$, we can find an expression for its derivative $f(x) = dE/dx$. Then we find the positions of extrema by solving $f(x) = 0$ and checking which of them belongs to the global minimum (recall that some of them might belong to local minima or even maxima). We just reduced the optimization problem to the root finding problem for which we have a variety of solution algorithms discussed in the chapter 8.

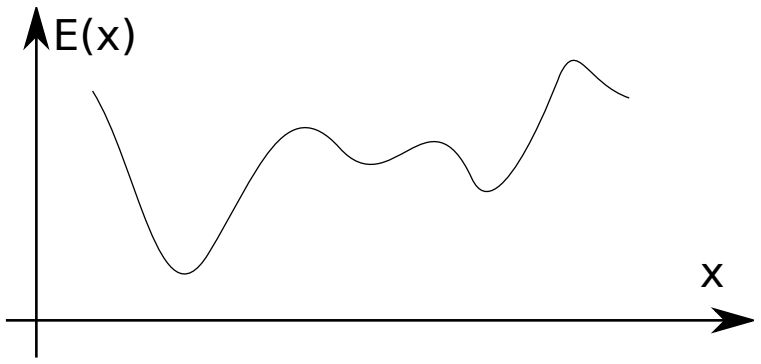

Figure 1.1: The example of the function to be minimized.

Interestingly enough, if we know how to solve the minimization problem, we can use it for the root finding problem, i.e. $f(x) = 0$. This is done by assigning the merit function to

be $E(x) = f(x)^2$. Since such $E(x) \geq 0$, the global minimum position (x_m) where $E = 0$ coincides with the root of $f(x)$.

Now, let's discuss the general requirements for the minimization algorithm. We need to somehow bracket the minimum (optimum) location, and then iteratively reduce the bracket size, until we find the precision of the optimum location (given by the bracket length) satisfactory. Let's assume that we are in *the close vicinity of a minimum*, i.e. there is no other extremum inside the bracket. If you think a bit about the problem, you will see that probing only one test point within the bracket does not provide enough information to correctly assign the new ends of the bracket. So, we need to calculate the function value for at least two inner points, then we can assign the new bracket by the following rule: the bracket ends should be the two closest points (among already checked: bracket ends and two inner points) surrounding the lowest currently known merit point. Then we can repeat the above bracket updating procedure until the required precision is reached.

Now, the key question becomes: how do we choose two inner points for the above general algorithm efficiently[1]? Quite often, the merit function is expensive to calculate either from the time required for calculation or sometimes literally (if you optimize a rocket engine, it is not cheap to build a new test engine). Therefore, we would like to reduce the number of the merit function calculations per bracket reduction.

1.2.1 The golden section optimum search algorithm

The golden section optimum search algorithm addresses the efficiency question by reusing one of the two previous test points and, consequently, requires only one additional function calculation per bracket update.

[1] Efficiently means optimally, so we are solving yet another optimization problem.

The golden section optimization algorithm

Assign a bracket interval (a, b) which surrounds a minimum (ideally the global minimum) closely enough, i.e. there is no other extrema in the interval (this is similar to the requirement in the item 4 below).

1. Calculate $h = (b - a)$.
2. Assign new probe points $x_1 = a + R \times h$ and $x_2 = b - R \times h$, where
$$R = \frac{3 - \sqrt{5}}{2} \approx 0.38197 \tag{1.1}$$
3. Calculate $E_1 = E(x_1)$, $E_2 = E(x_2)$, $E_a = E(a)$, and $E_b = E(b)$.
4. We require the bracket size to be small enough h: $E(x_1) \leq E(a)$ and $E(x_2) \leq E(b)$. **This is important!** In this case, we can shrink or update the bracket:
   - if $E_1 < E_2$ then $b = x_2$ and $E_b = E_2$ else $a = x_1$ and $E_a = E_1$;
   - recalculate $h = (b - a)$.
5. If the required precision is reached, i.e. $h < \varepsilon_x$, then stop (choose any point within the bracket for the final answer) otherwise do the steps below.
6. Reuse one of the old points either (x_1, E_1) or (x_2, E_2)
   - if $E_1 < E_2$
     then $x_2 = x_1$, $E_2 = E_1$, $x_1 = a + R \times h$, $E_1 = E(x_1)$
     else $x_1 = x_2$, $E_1 = E_2$, $x_2 = b - R \times h$, $E_2 = E(x_2)$
7. Repeat from the step 4.

It is easy to see that the new bracket size $h' = (1 - R) \times h$. In other words, the bracket shrinks by the factor $1 - R = \varphi = (\sqrt{5} - 1)/2 \approx 0.61803$, which is the golden ratio[2]. This gives the name to the whole algorithm.

Derivation of the R coefficient

Let's derive the expression for the R coefficient. Look at the above algorithm; at the first step we have

$$x_1 = a + R \times h \tag{1.2}$$
$$x_2 = b - R \times h \tag{1.3}$$

[2] The golden ratio dates to the time of ancient Greeks and naturally comes up in the solutions of several geometrical and mathematical problems. Some even argue that it has some aesthetic properties for geometrical constructs. For example, the rectangle with the ratio of the short side to long side equal to φ is apparently pleasing to eyes. Maybe this is why more and more computer screens have sides of ratio 9 to 16 which is close to φ.

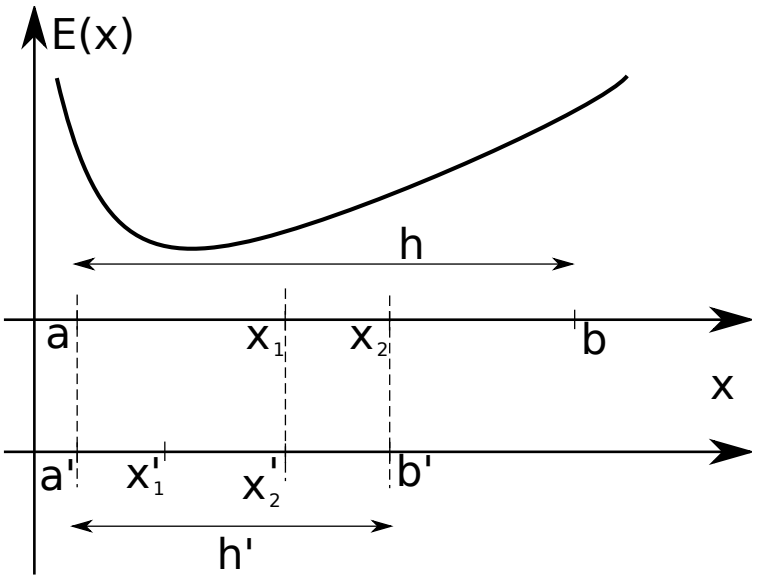

Figure 1.2: The golden section minimum search method illustration.

As depicted in fig. 1.2, if $E(x_1) < E(x_2)$, $a' = a$, and $b' = x_2$ then we assign the next probing points x_1' and x_2' according to

$$x_1' = a' + R \times h' = a' + R \times (b' - a') \tag{1.4}$$

$$x_2' = b' - R \times h' = b' - R \times (b' - a') = x_2 - R \times (x_2 - a) \tag{1.5}$$

We would like to reuse one of the previous evaluations of E, so we require that $x_1 = x_2'$. Using this, we plug eq. (1.2) into the right hand side of eq. (1.5) and obtain

$$a + R \times h = b - R \times h - R \times (b - R \times h - a) \tag{1.6}$$

Recalling that $h = b - a$, we can cancel it out and obtain the quadratic equation

$$R^2 - 3R + 1 = 0$$

with two possible roots

$$R = \frac{3 \pm \sqrt{5}}{2}$$

We need to choose the solution with the 'minus' sign since $R \times h$ should be less than h to land the two probe points inside the bracket.

1.2.2 MATLAB's built-in function for the one dimension optimization

MATLAB has a built-in function `fminbnd` to perform the one dimension optimization, which uses the modified golden section search algorithm for its implementation. The `fminbnd`

function takes three mandatory arguments: the handle to the merit function, the left end of the bracket, and the right end of the bracket. We can also supply some additional options that, for example, set the required precision for the minimum location or the number of the permitted function evaluations.

1.2.3 One dimensional optimization examples

Maximum of the black body radiation

In physics, an object is called '*black body*' if it does not reflect electro-magnetic radiation. Surprisingly, a black body could radiate quite a lot of energy and thus appear bright when it is hot enough. In this regard, our Sun is actually an almost perfect black body and so is an incandescent bulb when it is on.

According to Plank's law, the spectrum of the power radiated per area of the black body per wavelength into the solid angle has the following dependence on wavelength of electro-magnetic radiation (λ) and temperature (T)

$$I(\lambda, T) = \frac{2hc^2}{\lambda^5} \frac{1}{e^{\frac{hc}{\lambda kT}} - 1} \tag{1.7}$$

where

h is the Planck constant 6.626×10^{-34} J×s,

c is the speed of light 2.998×10^{8} m/s,

k is the Boltzmann constant 1.380×10^{-23} J/K,

T is the body temperature in K,

λ the is wavelength in m.

For an incandescent bulb with a typical filament's temperature of 1500 K, the black body radiation spectrum looks as depicted in fig. 1.3. We calculate it with the help of the function, which implements eq. (1.7), listed below

Listing 1.1: black_body_radiation.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/black_body_radiation.m)

```
function I_lambda=black_body_radiation(lambda,T)
% black body radiation spectrum
% lambda - wavelength of EM wave
% T - temperature of a black body
h=6.626e-34;  % the Plank constant
c=2.998e8;    % the speed of light
k=1.380e-23;  % the Boltzmann constant
```

```
I_lambda = 2*h*c^2 ./ (lambda.^5)  ./ (exp(h*c./(lambda*k*T))-1);
end
```

It is easy to see that most of the radiation is emitted above the 1000 nm wavelength, where the human eye has no ability to register the light. Consequently, incandescent bulbs are not very efficient at providing light, since most of the energy becomes heat (i.e. infrared radiation). It's no wonder there is a big effort to replace incandescent bulbs with modern fluorescent or LED bulbs, which provide much more efficient lighting.

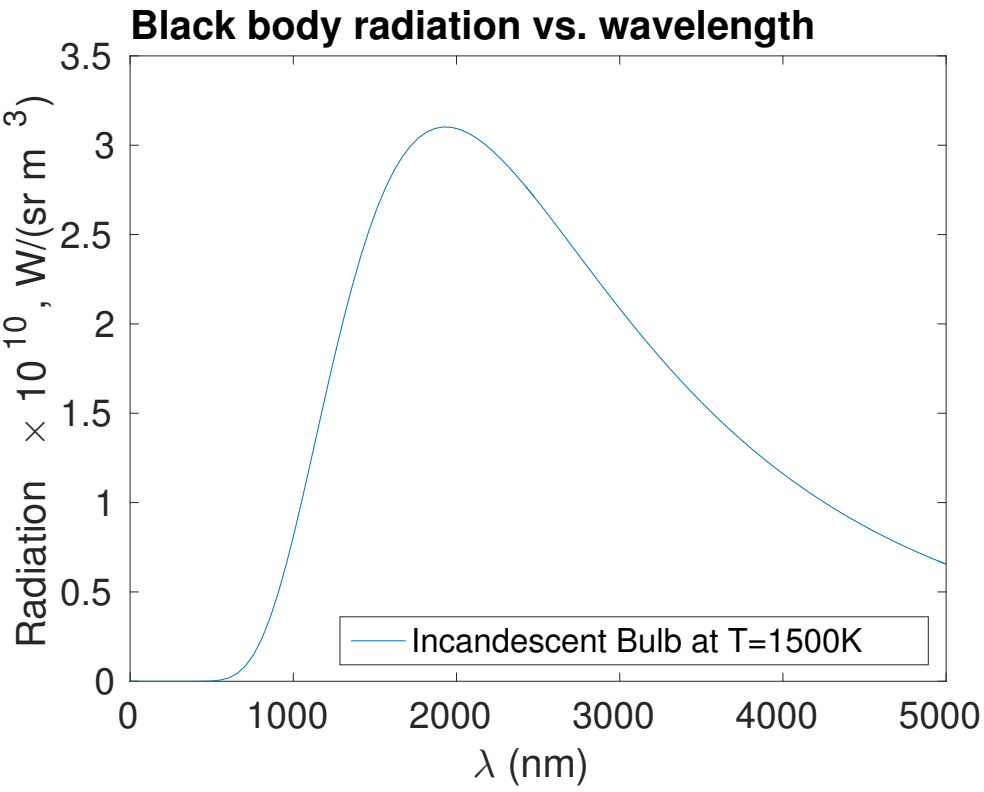

Figure 1.3: The black body radiation spectrum for an incandescent bulb with the filament's temperature T=1500 K

Suppose we would like to know the wavelength of the Sun's maximum radiation. MATLAB knows how to find a minimum of the function, so we create the merit function `f`, which is `black_body_radiation` reflected (inverted) with respect to 'x' axis. The Sun's photosphere temperature is 5778 K, so we set the T accordingly.

```
T=5778;
f = @(x) - black_body_radiation(x,T);
```

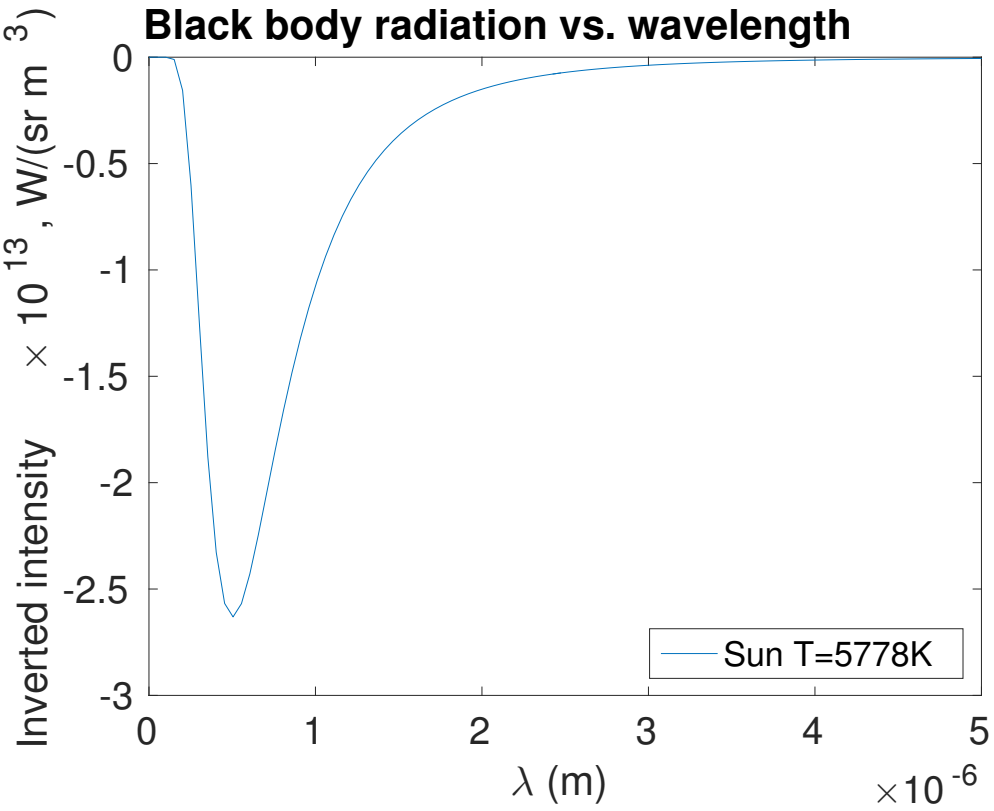

Figure 1.4: The inverted radiation spectrum of the Sun or any black body with the temperature T=5778 K.

The resulting plot of the merit function, i.e. the inverted black body radiation spectrum of the Sun, is depicted in fig. 1.4. As we can see, the minimum is located somewhere in the $(10^{-9}, 10^{-6})$ interval. We need to adjust the default 'x' tolerances, since the typical 'x' value, as it shown in fig. 1.4 is in the order of 10^{-6} which is MATLAB's default precision. We use `optimset` command below to tune the precision. Now, we are ready to search the minimum location with the following commands.

```
fminbnd(f, 1e-9, 1e-6, optimset('TolX',1e-12))
ans = 5.0176e-07
```

As you can see, the answer is `5.0176e-07`, measured in meters, so the maximum of the Sun radiation is at roughly 502 nm, which corresponds to green light[3]. No wonder that human eye is the most sensitive to green light, since it is the dominating wavelength in a naturally lit environment.

[3] Strangely enough, the Sun appears to be yellowish to a human eye. This is due to a particular response of the light sensitive elements in the eye and in the brain that reconstruct the perceived color. One side effect is that there are white, blue, yellow, and red stars (which are all the black bodies listed in the order of the decreasing temperature) but there are no green stars in the sky.

1.3 Multidimensional optimization

We will not talk about algorithms of multidimensional optimization for smooth functions here[4]. If you are interested in them, have a look at specialized numerical methods books, for example [6]. Instead, we will piggyback on the MATLAB's `fminsearch` function.

1.3.1 Examples of multidimensional optimization

The inversed sinc function

Let's find a minimum of the inverse two-dimension sinc function

$$f1(x,y) = -\sin(r)/r, \text{ where } r = \sqrt{x^2+y^2} \tag{1.8}$$

The plot of this function is shown in fig. 1.5.

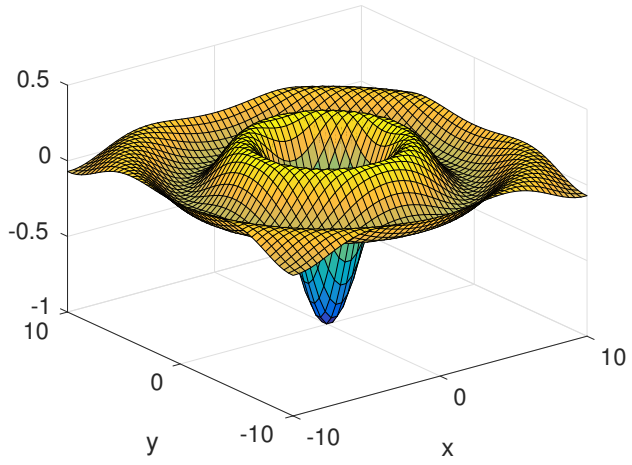

Figure 1.5: The two-dimension sinc function plot.

To do the optimization with `fminsearch`, we need to implement eq. (1.8) as a function of only one vector argument as shown below

Listing 1.2: `fsample_sinc.m` (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/fsample_sinc.m)

```
function ret=fsample_sinc(v)
        x=v(1); y=v(2);
        r=sqrt(x^2+y^2);
        ret= -sin(r)/r;
end
```

[4] It is the author's opinion that programming such algorithms does not bring much educational value. MATLAB has good enough and ready to use implementations.

The components of the input vector `v` are representing 'x' and 'y' coordinates. It is up to us to assign 'x' as the first component and 'y' as the second one; we can do it the other way as well.

To call `fminsearch` we need two arguments: the first is the handle to the function to be optimized (i.e. `@fsample_sinc`) and the second is a starting point for the minimum search algorithm (in this example, we will use `[0.5, 0.5]`). Once we have made the above decision, we are ready to search for the minimum:

```
>> x0vec=[0.5, 0.5];
>> [x_opt,z_opt]=fminsearch(@fsample_sinc, x0vec)
  x_opt = [0.2852e-4,    0.1043e-4]
  z_opt = -1.0000
```

As we can see, the minimum is located at `x_opt=[0.2852e-4, 0.1043e-4]` which is very close to the true global minimum at `[0,0]`. The value of the function in the found optimum location is `z_opt = -1.0000` which matches (within shown precision) the global minimum value -1.

It is easy to miss the global minimum if we choose a bad starting point as in the example below

```
>> x0vec=[5, 5];
>> [x_opt,z_opt]=fminsearch(@fsample_sinc, x0vec)
  x_opt = [ 5.6560    5.2621 ]
  z_opt = -0.1284
```

Here, we find a local minimum, but not the global minimum. Recall that no algorithm can find the global minimum in a general case, especially when it starts far away from the optimum.

3D optimization

Let's find the minimum of the function

$$f2(x, y, z) = 2x^2 + y^2 + 2z^2 + 2xy + 1 - 2z + 2xz. \tag{1.9}$$

We do it by implementing $f2$ as show below

Listing 1.3: `f2.m` (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/f2.m)

```
function fval = f2( v )
x = v(1);
y = v(2);
```

```
z = v(3);
fval = 2*x^2+y^2+2*z^2+2*x*y+1-2*z+2*x*z;
end
```

Yet again, it is up to us which component of the input vector we use as 'x', 'y', and 'z'. To find the minimum, we choose an arbitrary starting point `[1,2,3]` and execute

```
>> [v_opt, f2_opt]=fminsearch(@f2, [1,2,3])
v_opt = -1.0000    1.0000    1.0000
f2_opt = 4.8280e-10
```

At first glance, it may not be clear how to check the calculated minimum position `v_opt = -1.0000 1.0000 1.0000`, but we can rewrite the eq. (1.9)

$$f2(x,y,z) = (x+y)^2 + (x+z)^2 + (z-1)^2 \tag{1.10}$$

Since every term above is quadratic, the minimum is reached when each term is equal to zero. So the global minimum is at $[x,y,z]$=`[-1,1,1]`. For the same reason, $f2$ cannot be less than zero; so `f2_opt = 4.8280e-10`, which is very close to zero, is appropriate.

Joining two functions smoothly

Suppose we have a function which has the following form[5]

$$\Psi(x) = \begin{cases} \Psi_{in}(x) = \sin(kx) & : \quad 0 \le x \le L \\ \Psi_{out}(x) = Be^{-\alpha x} & : \quad x > L \end{cases}$$

We would like to make our function smooth, i.e. both the function and its first derivative are continuous everywhere. The only problem point is located at $x = L$ where one continuous and smooth function meets another. The following equations are in charge of the smooth link conditions

$$\Psi_{in}(L) = \Psi_{out}(L) \tag{1.11}$$
$$\Psi'_{in}(L) = \Psi'_{out}(L) \tag{1.12}$$

[5] You are probably interested where this function comes from. This is the solution of the Quantum Mechanics problem about a particle in one-dimensional potential well described by the following potential

$$U(x) = \begin{cases} \infty & : \quad x < 0 \\ 0 & : \quad 0 \le x \le L \\ U_o & : \quad x > L \end{cases}$$

where $k = \frac{\sqrt{2m(E-U_o)}}{\hbar}$, $\alpha = \frac{\sqrt{2m(U_o-E)}}{\hbar}$, m is the mass of the particle, E is its total energy, and $\hbar = h/(2\pi)$ is the reduced Planck constant. Since the potential is infinite at $x < 0$, $\Psi(x) = 0$ in this region.

After substitution of the Ψ expression, we obtain

$$\sin(kL) = Be^{-\alpha L} \tag{1.13}$$

$$k\cos(kL) = -\alpha Be^{-\alpha L} \tag{1.14}$$

Suppose that we somehow know k. What should be the values of α and B? We can solve the above system of nonlinear equations to get α and B but this is a tedious task. Besides, this chapter is about optimization. So, we will use our new skills to solve the problem. We rearrange the equations as

$$\sin(kL) - Be^{-\alpha L} = 0 \tag{1.15}$$

$$k\cos(kL) + \alpha Be^{-\alpha L} = 0 \tag{1.16}$$

then we square and add them together

$$\left(\sin(kL) - Be^{-\alpha L}\right)^2 + \left(k\cos(kL) + \alpha Be^{-\alpha L}\right)^2 = 0. \tag{1.17}$$

So far, we have not done anything out of the ordinary. Now, we call the right hand side of the above equation as the merit of our problem

$$M(\alpha, B) = \left(\sin(kL) - Be^{-\alpha L}\right)^2 + \left(k\cos(kL) + \alpha Be^{-\alpha L}\right)^2. \tag{1.18}$$

The global minimum of the merit function is the point in α and B space where eqs. (1.15) and (1.16) are satisfied. The listing of the merit function is shown below

Listing 1.4: merit_psi.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/merit_psi.m)

```
function [m] = merit_psi(v, k , L)
% merit for the potential well problem
alpha=v(1);
B=v(2);

m=(sin(k*L) - B*exp(-alpha*L))^2 + (k*cos(k*L) + alpha*B*exp(-alpha*L))^2;

end
```

All we need to do is to assign the k and L values and make the `fminsearch` compatible merit function (i.e. the one which accepts the problem parameters vector). All of this is done by executing the code below

```
>> k=2+pi; L=1;
>> merit=@(v) merit_psi(v, k, L);
>> v0=fminsearch( @merit, [.11,1] )
v0 = 2.3531   -9.5640
```

The resulting values are $\alpha = 2.3531$ and $B = -9.5640$. The plot of the Ψ function with these values is shown in fig. 1.6. As you can see, the transition between inner and outer parts of the Ψ function is smooth, as required.

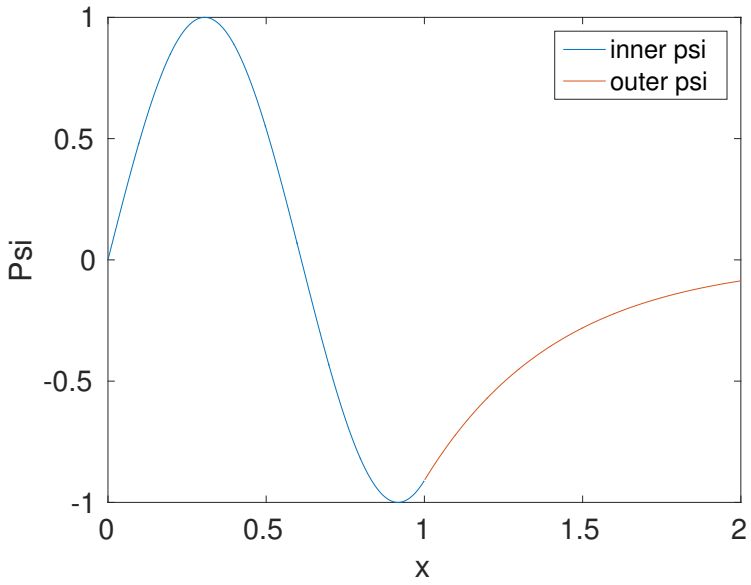

Figure 1.6: The plot of the smoothly connected inner and outer parts of the Ψ function

Hanging weights problem

Consider the masses m_1 and m_2, which are connected by rods with length L_1, L_2, and L_3 to suspension points hanging in Earth's gravitational field (see fig. 1.7). The suspension points are separated horizontally by L_{tot} and vertically by H_{tot} distances. Our goal is to find the angles θ_1, θ_2, and θ_3 of these weights arrangement in equilibrium. This is a typical Physics 101 problem. To solve it, we need to set and solve several equations regarding forces and torques acting on the system. This is not a trivial task. You might wonder what this problem has to do with optimization. You will soon see that this problem can be replaced with the minimization problem and solved quite elegantly (i.e. with fewer equations to track). The downside is that the solution will be numerical, i.e. we would have to redo the calculations if some parameters change.

Recall that the other name of the merit function is energy, for a quite important reason: a real life system seeks the minimum of the potential energy due to forces of nature. So, we need to minimize the potential energy subject to the length constraints. The latter requirement is important since the potential energy minimization alone will push our weights to the lowest position, but they are connected by the links, so this needs to be taken into account. See the code below for the resulting merit function of this problem. Note that we already put the particular values for the masses, lengths of each rod, and suspension point separation.

Listing 1.5: `EconstrainedSuspendedWeights.m` (available at `http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/EconstrainedSuspendedWeights.m`)

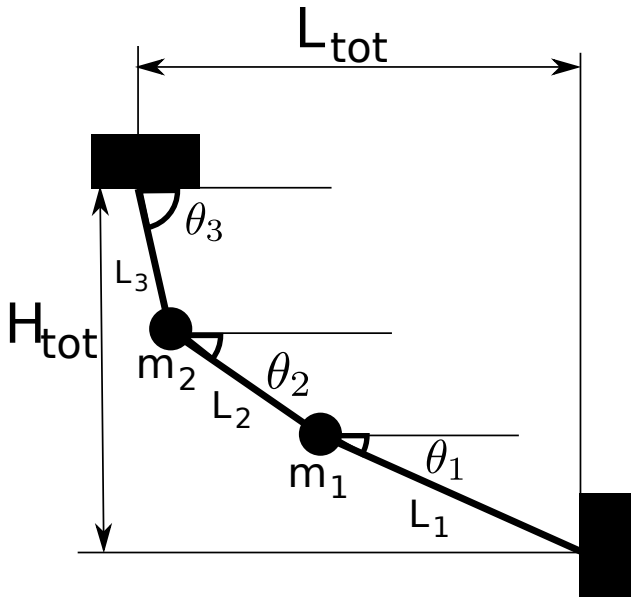

Figure 1.7: The suspended weights arrangement.

```
function [merit, LengthMismatchPenalty, HeightMismatchPenalty] =
    EconstrainedSuspendedWeights( v )
% reassign input vector elements to the meaningful variables
theta1=v(1); theta2=v(2); theta3=v(3); % theta angles

g=9.8; % acceleration due to gravity
m1=2; m2=2; % masses of the weights
L1=3; L2=2;  L3=3; % lengths of each rod
Ltot=4; Htot=0; % suspension points separations
% fudge coefficients to make merit of the potential energy comparable to
% length mismatch
mu1=1000; mu2=1000;

Upot=g*( (m1+m2)*L1*sin(theta1)+m2*L2*sin(theta2) ); %potential energy
HeightMismatchPenalty=(Htot-(L1*sin(theta1)+L2*sin(theta2)+L3*sin(theta3)))
    ^2;
LengthMismatchPenalty=(Ltot-(L1*cos(theta1)+L2*cos(theta2)+L3*cos(theta3)))
    ^2;

merit=Upot+mu1*LengthMismatchPenalty+mu2*HeightMismatchPenalty;
end
```

The above code needs some walkthrough. Why is length mismatch called a penalty? If we have a length mismatch for some test point, we need to let the solver know that we

are breaking some constraints, i.e. we need to penalize such a probe point. Since we are looking for the minimum, we add something positive to the resulting merit value at this point. It is usually good idea to add a square of mismatch: it is always positive and smooth. The `mu1` and `mu2` coefficients emphasize the importance of the particular contribution to the resulting merit. Their assignment usually requires some tuning to make all contributions equally important.

For the problem with the chosen above parameters, i.e.

```
m1=2; m2=2;
L1=3; L2=2;  L3=3;
Ltot=4; Htot=0;
```

We can notice the symmetry: the masses are the same and the outer rods are the same. So, we can be sure that the inner rod (L_2) should be horizontal, i.e. $\theta_2 = 0$; additionally, θ_1 should be equal to $-\theta_3$ due to the same symmetry. We can even find their values precisely: $\theta_1 = -1.231$ and $\theta_3 = 1.231$. Let's see if the minimization algorithm gives the correct answer.

```
>> theta = fminsearch( @EconstrainedSuspendedWeights, [-1,0,-1], optimset('
   TolX',1e-6))
theta = -1.2321   -0.0044    1.2311
```

You can see that the answer is quite close to the theoretically predicted values. So, we declare our approach successful.

1.4 Combinatorial optimization

There is a subclass of problems where the parameters vector or its components can take only discrete values. For example, you can only buy hot dog buns in set of 8. So, when you are optimizing your spendings for a party, you would have to account for 0 or 8 or 16 ... as a possible number of buns.

As a result of this discretization, the optimization algorithms and function, which we have covered before, are of little use. They assume that any component can take any value and that the merit function will be fine with it. There is a way around this. We can create constraining functions which take care of it but this is generally not a trivial task.

Instead, we have to find a method to search through discrete sets of all possible input values, i.e. try all possible combinations of $\vec{x}$ components. Hence, the name of the optimum search is combinatorial optimization.

Unfortunately, there is no way to design a general optimum searching algorithm which can solve any combinatorial problem. So every combinatorial problem requires a specific solution, but the general idea is the following: probe every possible combination of the inputs and

select the best. Usually, the hardest part is to devise a method to go over all possible combinations (ideally without repeating the already probed one).

We will cover two problems which should give you a general idea about how to approach problems of this type.

1.4.1 Backpack problem

Suppose you have a backpack with the given size (volume) and a set of objects with given volumes and monetary (or sentimental) values. Our job is to find the subset of items that can be packed into the backpack and has the maximum combined value. For simplicity, we will assume that every item occurs only once.

The mathematical formulation of the above problem is the following: maximize the merit function

$$E(\vec{x}) = \sum \text{value}_i x_i = \overrightarrow{\text{values}} \cdot \vec{x}$$

subject to the following constraint

$$\sum \text{volume}_i x_i = \overrightarrow{\text{volumes}} \cdot \vec{x} \leq \text{BackpackSize}$$

Where x_i can be 0 or 1, i.e. it reflects whether we pack i_{th} item or not.

We will try a brute force approach, i.e. we will check every possible combination of the items[6]. For each item there are two possible outcomes (pack or do not pack). If we have N items the number of all possible combinations is 2^N. So, both the size of all possible combinations (i.e. the problem space) and the solving time grow **exponentially**. On the bright side, we will find the **global** optimum.

The hardest part of the problem is to find a way to generate all possible combinations of objects to leave or take. We note that the vector $\vec{x}$ is a combination of zeros and ones. For example, the vector might be $\vec{x} = [0, 1, 0, 1, \cdots, 1, 1, 0, 1, 1]$. A combination of zeros and ones resembles a binary number. The set of all zeros corresponds to the smallest possible positive integer number, i.e. 0, and the set of all ones corresponds to the largest possible binary number constructed with N ones: $2^N - 1$. There is a simple recipe for generating all possible integer numbers from 0 to $2^N - 1$: start from 0 and just keep adding 1 to get the next one. The tricky part is to do it according to binary arithmetic, more precisely, to implement the proper tracking of the digit overflow mechanism[7]. All modern computers use

[6] There are better ways. We can be more selective about how we select items to pack. For example, we can presort all items in the ascending order and put them one by one; if the current item does not fit, there is no reason to probe even larger items. This will save computational time. Another way is to use the simulated annealing algorithm (see section 1.5) to find a good enough solution.

[7] In the decimal system, we cannot add 1 to the largest decimal symbol (9) without using an extra digit, i.e. $9 + 1 = 10$. Similarly, in the binary system, where the largest symbol is 1, $1 + 1 = 10_2$.

the binary system under the hood, but, strangely enough, we would have to put some effort into the proper implementation of it[8].

The pseudo code for probing all $\vec{x}$ combinations for N objects would be the following:

Pseudo code for the backpack problem

1. Start with $\vec{x} = [0, 0, 0, 0, \cdots, 0, 0]$ consisting of N zeros.
2. Every new $\vec{x}$ will be generated by adding 1 to the previous $\vec{x}$ according to binary addition rules.
   - For example, $x_{next} = [1, 0, 1, \cdots, 1, 1, 0, 1, 1] + 1 = [1, 0, 1, \cdots, 1, 1, 1, 0, 0]$.
3. For every new $\vec{x}$, check if the items fit into the backpack and if the new packed value is larger than the previously found maximally packed value.
4. We are done once we have tried all 2^N combinations of $\vec{x}$.

MATLAB's realization of this algorithm is listed below

Listing 1.6: backpack_binary.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/backpack_binary.m)

```
function [items_to_take, max_packed_value, max_packed_volume] = ...
    backpack_binary( backpack_size, volumes, values )
% Returns the list of items which fit in backpack and have maximum total
   value
% backpack_size - the total volume of the backpack
% volumes       - the vector of items volumes
% values        - the vector of items values

% We need to generate vector x which holds designation: take or do not take
% for each item.
% For example x=[1,0,0,1,1] means take only 1st, 4th, and 5th items.
% To generate all possible cases, go over all possible combos of 1 and 0
% It is easy to see the similarity to the binary number presentation.
% We will start with x=[0, 0, 0, ... ,1]
% and add 1 to the last element according to the binary arithmetic rules
% until we reach x=[1, 1, 1, ... ,1] and
% then x=[0, 0, 0, ... , 0], which is the overfilled [111..1] +1.
% This routine will sample all possible combinations.

% nested function does the analog to the binary 1 addition
function xout=add_one(x)
    xout = x;
    for i=N:-1:1
```

[8] There are MATLAB functions that deal with the conversion to and from binary numbers.

```
        xout(i)=x(i)+1;
        if (xout(i) == 1 )
            % We added 1 to 0. There is no overfill, and we can stop here.
            break;
        else
            % We added 1 to 1. According to the binary arithmetic,
            % it is equal to 10.
            % We need to move the overfilled 1 to the next digit.
            xout(i)=0;
        end
    end
end

% initialization
N=length(values);   % the number of items
xbest=zeros(1,N);   % we start with empty backpack, as the current best
max_packed_value=0; % the empty backpack has zero value

x=zeros(1, N); x(end)=1; % assigning 00000..001 the very first choice set

while ( any(x~=0) ) % while the combination is not [000..000]
    items_volume = sum(volumes .* x);
    items_value  = sum(values  .* x);
    if ( (items_volume <= backpack_size) && (items_value > max_packed_value)
        )
        xbest=x;
        max_packed_value=items_value;
        max_packed_volume=items_volume;
    end
    x=add_one(x);
end

indexes=1:N;
items_to_take=indexes(xbest==1); % converting x in the human notation
end
```

The interesting part of the above code is the `add_one` subfunction which does binary addition. Another feature is the use of `indexes` at the next to last line. This returns a human readable list of the objects to pack, instead of $\vec{x}$ consisting of zeros and ones. The rest is just bookkeeping.

We can test the backpack algorithm with a list of the five items of various values and volumes

```
>> backpack_size=7;
```

```
>> volumes=[  2,  5,  1,  3, 3];
>> values =[ 10, 12, 23, 45, 4];
>> [items_to_take, max_packed_value] = ...
            backpack_binary( backpack_size, volumes, values)

  items_to_take = [1 3 4]
  max_packed_value = 78
```

As you can see the algorithm suggests to take the first, third, and fourth items to maximize the total packed value. There is no better solution than this, as we can see by solving this problem ourselves.

The algorithm searches through all combinations of 5 objects almost instantaneously. To go over the list of 20 items, my computer takes 24 seconds. It would take almost 1000 times longer to sort through 30 items, i.e. more than 6 hours. It is unpractical to use this algorithm to sort through even a slightly longer list of objects. This is the price for the ability to find the global optimum via probing all 2^N combinations.

Words of wisdom

Use of brute force algorithms is never a good idea: they are fast to implement and slow to use.

1.4.2 Traveling salesman problem

Suppose that a salesman has a list of N cities with given coordinates (x and y) to visit. The salesman starts in the city labeled 1 and needs to be in the N_{th} city at the end of a route (see fig. 1.8). We need to find the shortest route so that the salesman visits every city and does it only once.

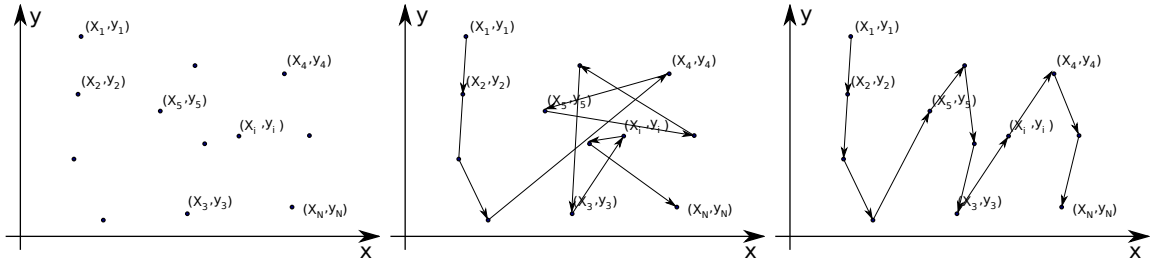

Figure 1.8: The traveling salesman problem illustration. The N cities arrangement is shown at the left; a possible sub optimal route shown in the middle; a shorter route is shown at the right.

This problem has many connections to the real world. Every time you ask your navigator to

find a route from one place to another, the navigator unit has to solve a very similar problem. However, the navigator has to select intermediate locations and then find the shortest route. If you choose the destination too far, the navigator may even complain that it has not enough resources to do the planning and may suggest to choose an intermediate destination. Below you will see why planning a long route with too many places to visit is a hard problem for a computer (at least if a brute force approach taken).

Let's estimate the problem size of our traveling salesman problem, i.e. how many possible combinations exist. If we have N cities in total, the salesman can go from the first city to $N-2$ destinations. We subtract two because the first and last cities are predefined by the problem. For the third city to visit, we have $N-3$ choices. For the fourth city, we have $N-4$ cities. The sequence goes on until we have no choices. So the total number of choices is given by

$$(N-2) \times (N-3) \times (N-4) \times \cdots \times 2 \times 1 = (N-2)! \tag{1.19}$$

This grows even faster than exponential dependence. Recall Stirling's approximation: $N! \sim \sqrt{2\pi N}(N/e)^N$. If we spend only a nanosecond to test every route consisting of 22 cities, it would take about 77 years to go over all possible combinations since $20! \approx 2.4 \times 10^{18}$. Now you see why choosing a route is quite a hard problem for a navigator[9].

Let's not be discouraged by the above numbers. We will be able to select the shortest route from 10 cities within a minute even if we go over all combinations. As before, the hard part is to find a way to go through all possible permitted combinations of the cities. We do it by noticing that a complete route involves all cities, so another route could be achieved by swapping positions of any two cities in the route assignment, i.e. by a permutation. So, we need to find a way to go over all possible permutations.

Permutation generating algorithm

Luckily, there are permutation generating algorithms available. MATLAB has one of them implemented as the `perms` function. Unfortunately, it is not suitable for our needs since it generates and stores a list of **all** permutations. This consumes all available computer memory even for a modest $N \approx 15$. Instead, we will use the method which goes back to 14th century India (see "Generating all permutations" in [4]). The pseudo code of this algorithm is shown below

[9] As with the backpack problem, there are better algorithms. Some are smart about ruling out sub optimal routes without even testing them; they still find the global minimum but with fewer tests. For example, the Held-Karp algorithm does it in $O(2^N N^2)$ steps [1], though this one requires quite a lot of memory to work. Other algorithms find a *good enough* route. For example, the simulated annealing algorithm, which we will see very soon in section 1.5.

Next lexicographic permutation generating algorithm

1. Start with the set sorted in the ascending order, i.e. $p = [1, 2, 3, 4, \cdots, N-2, N-1, N]$
2. Find the largest index k such that $p(k) < p(k+1)$.
   - If no such index exists, the permutation is the last permutation.
3. Find the largest index l such that $p(k) < p(l)$.
   - There is at least one $l = k+1$
4. Swap $p(k)$ with $p(l)$.
5. Reverse the sequence from $p(k+1)$ up to and including the final element $p(end)$.
6. We have a new permutation. If we need another, repeat from the step 2.

To generate a new permutation, it only needs to know the previous one, so the algorithm memory footprint is negligible. The name lexicographic comes from the requirement of the items to be sortable (i.e. we can compare their values). In the past, they used letters since there is the particular order (i.e. ranking) of them in the alphabet. We do not have to use letters, since numbers naturally possess this property. See MATLAB's implementation of this algorithm below

Listing 1.7: permutation.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/permutation.m)

```
function pnew=permutation(p)
        % Generates a new permutation from the old one
        % in such a way that new one will be lexicographically larger.
        %
        % If one wants all possible permutations, she
        % must prearrange elements of the permutation vector p
        % in ascending order for the first input, and then
    % feed the output of this function to itself.
        %
        % Elements of the input vector allowed to be not unique.
        %
        % See "The Art of Computer Programming, Volume 4:
        % Generating All Tuples and Permutations" by Donald Knuth
        % for the discussion of the algorithm.
        %
        % This implementation is optimized for MATLAB. It avoids cycles
        % which are costly during execution.

        N=length(p);
        indxs=1:N; % indexes of permutation elements

        % looking for the largest k where p(k) < p(k+1)
```

```
        k_candidates=indxs( p(1:N-1) < p(2:N) );
        if ( isempty(k_candidates) )
                % No such k is found thus nothing to permute.
                pnew= p;
        % We must check at the caller for this special case pnew==p
        % as condition to stop.
        % All possible permutations are probed by this point.
        return;
        end
        k=k_candidates(end); % note special operator 'end' the last element
           of array

        % Assign the largest l such that p(k) < p(l).
        % Since we are  here at least one solution is possible: l= k+1
        indxs=indxs(k+1:end); % we need to truncate the list of possible
           indexes
        l_candidates=indxs( p(k) < p (k+1:end) );
        l=l_candidates(end);

        tmp=p(l); p(l)=p(k); p(k)=tmp; % swap p(k) and p(l)

        %reverse the sequence between p(k+1) and p(end)
        p(k+1:end)=p( end:-1:k+1 );
        pnew=p;
end
```

The important thing to mention about the above code is that once the last permutation is reached (all items will be sorted in descending order), it will output the same combination that was the input. It is up to us to check for this condition to stop the search.

Combinatorial solution of the traveling salesman problem

Once we have a permutation generating algorithm, the rest is straight forward bookkeeping to find the shortest route. MATLAB's solution of the problem is shown below

Listing 1.8: traveler_comb.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/traveler_comb.m)

```
function [best_route, shortest_distance]=traveler_comb(x,y);
% x - cities x coordinates
% y - cities y coordinates

% helper function
```

```
function dist=route_distance(route)
    dx=diff( x(route) );
    dy=diff( y(route) );
    dist = sum( sqrt(dx.^2 + dy.^2) );
end

% initialization
N=length(x); % number of cities
init_sequence=1:N;

p=init_sequence(2:N-1); % since we start at the 1st city and finish in the
   last
pold=p*0; % pold MUST not be equal to p

route=[1,p,N]; % any route is better than none
best_route=route;
shortest_distance=route_distance(route);
% show the initial route with the first and the last cities marked with 'x'
plot( x(1), y(1), 'x', x(N), y(N), 'x',  x(2:N-1), y(2:N-1), 'o', x(route),
   y(route), '-');

while ( any(pold ~=p) ) % as long as the new permutation is different from
   the old one
    % Notice the 'any' operator above.
    pold=p;
    p=permutation(pold);
    route=[1,p,N];
    dist=route_distance(route);
    if (dist < shortest_distance)
        shortest_distance=dist;
        best_route=route;
        % Uncomment the following lines to see the currently best route
        %plot( x(1), y(1), 'x', x(N), y(N), 'x',  x(2:N-1), y(2:N-1), 'o', x
           (route), y(route), '-');
        %drawnow; % forces the figure update
    end
end
% plot all the cities and the best route
plot( x(1), y(1), 'x', x(N), y(N), 'x',  x(2:N-1), y(2:N-1), 'o', x(
   best_route), y(best_route), '-');

end
```

The author would like to attract the reader's attention to the helper function `route_distance` which calculates the route distance, as the name suggests. Here we piggyback on MATLAB's ability to generate an array with elements output in the order specified by the array of indexes (`route` in this case). The code also can plot the currently found best route as it executes.

Let's see how it copes with selection of the shortest route connecting 12 cities. We assign their coordinates rather randomly for the test shown below

```
>> x = [8.5 3.5 9.5 4.5 2.5 3.5 6.5 5.5 4.5 8.5 6.5 5.5];
>> y = [9.5 3.5 7.5 7.5 4.5 6.5 1.5 1.5 5.5 8.5 9.5 2.5];
>> the_shortest_route = traveler_comb(x,y)
        the_shortest_route = [1 3 10 11 4 6 9 5 2 8 7 12]
```

The shortest route connecting the cities with given above coordinates is shown in fig. 1.9. The author's computer takes about 90 seconds to find this route.

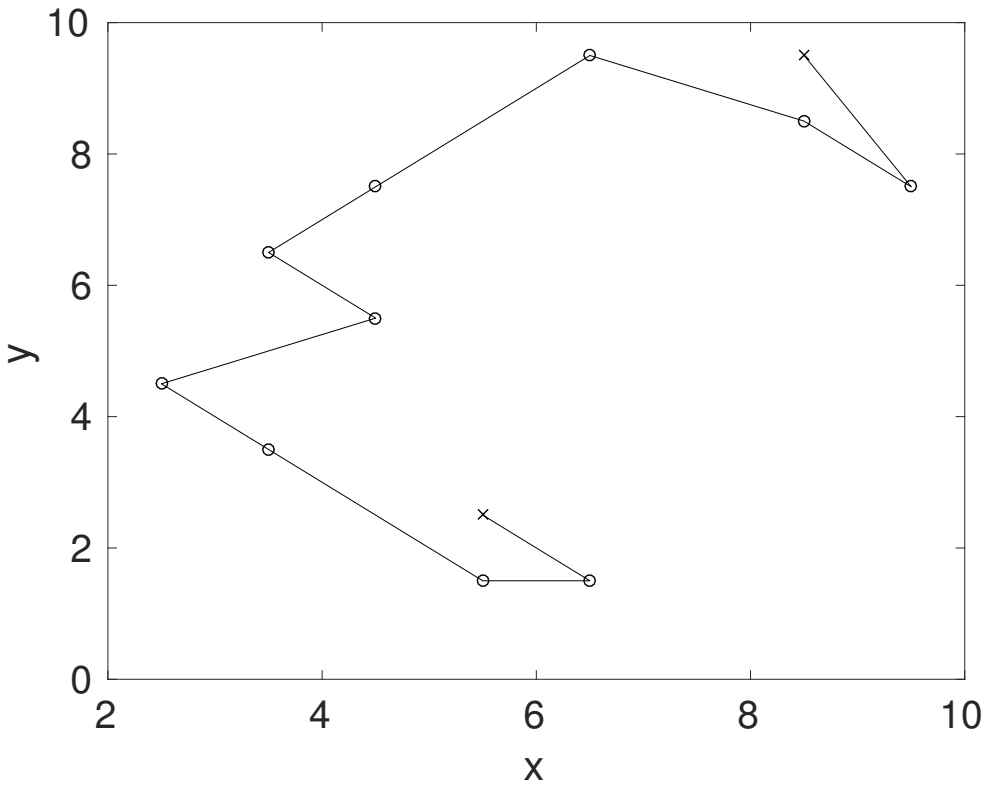

Figure 1.9: The shortest route connecting 12 cities. The first and the last cities to visit are marked with 'x' and the others are marked with 'o'.

1.5 Simulated annealing algorithm

We see that probing the full space permitted by combinatorics is not practical even for a seemingly small set of options. However, nature seems to handle the problem of energy minimization without any trouble. For example, if you think about a piece of metal, it has many atoms (the Avogadro number 6×10^{23} gives an order of magnitude estimate). Each of the atoms can be in many different states. So the problem space must be humongous. Yet, if we slowly cool the metal, i.e. anneal, then the system will reach the minimum energy state.

In 1953, Metropolis and coworkers suggested an algorithm which can mimic the distribution of system states according to energies of the states and the overall temperature of the whole physical system (see [5]), i.e. according to the Boltzmann energy distribution law. This law states that the probability to have energy E is given by

$$p(E) \sim \exp\left(-\frac{E - E_0}{kT}\right) \tag{1.20}$$

where E_0 is the energy of the lowest energy state, k is the Boltzmann constant, and T is the temperature of the system[10]. Recall that one of the names of the merit function is energy, which is very handy here. Note that if temperature goes to zero, the probability of any higher than global minimum energy state drops to zero according to the above equation. Now we have an idea for the general algorithm: evolve the system according to the Metropolis algorithm to mimic its physical behavior and simultaneously lower the temperature (anneal) to force the system into the lowest energy state. We spell out all the steps of this algorithm below.

[10] The derivation of this law is done in Statistical Mechanics and Thermodynamics courses.

Simulated annealing or modified Metropolis algorithm

1. Set the temperature to a high value, so kT is larger than typical energy (merit) function fluctuation.
   - This requires some experiments if you do not know this a priori
2. Assign a state $\vec{x}$ and calculate its energy $E(\vec{x})$
3. Change, somehow, the old $\vec{x}$ to generate a new one, $\vec{x}_{new}$
   - $\vec{x}_{new}$ should be somewhat **close or related** to the old optimal $\vec{x}$
4. Calculate the energy at the new point $E_{new} = E(\vec{x})$.
5. If $E_{new} < E$ then $x = x_{new}$ and $E = E_{new}$
   - i.e. we move to the new point of the lower energy

   otherwise, move to the new point with probability

$$p = \exp\left(-\frac{E_{new} - E}{kT}\right) \tag{1.21}$$

6. Decrease the temperature a bit, i.e. keep annealing.
7. Repeat from the step 3 for a given number of cycles.
8. $\vec{x}$ will hold the local optimum solution.

You are probably wondering what the temperature of an optimization problem is. Well, it is just a parameter, i.e. a number, which a physicist would insist to call temperature because of its overall resemblance to the one in physics. So, do not worry — you do not need to put a thermometer inside of your computer.

In finite time (limited number of cycles), the algorithm is guaranteed to find only the local minimum[11]. But there is a theorem (see [3]) which states:

> The probability to find the best solution goes to 1 if we run the algorithm for a longer and longer time with a slower and slower rate of cooling.

Unfortunately, this theorem is of no use since it does not give a constrictive recipe of how long to run the algorithm. It is even suggested that it will need more cycles than the brute force combinatorial search. However, in practice a good enough solution, i.e. quite close to the global minimum, can be found in quite a short time with a quite small number of cycles.

A nice feature of the simulated annealing algorithm is that it is not limited to discrete space problems and can be used for problems accepting real values of $\vec{x}$ components. The algorithm also has the ability to climb away from a local minimum if it is given enough time.

To make an efficient (fast) implementation of this algorithm, we need to choose the optimal

[11] Actually, if the final temperature is not zero, the final x could be away from the minimum due to a finite probability of going to the higher energy state at the step 5 above.

cooling rate[12] and the proper way to modify $\vec{x}$, so that the new value is not changing too much, i.e. majority of the time we are in the vicinity of the optimal solution. It is quite challenging to make the right choices.

1.5.1 The backpack problem solution with the annealing algorithm

We will show the solution of the backpack problem with the simulated annealing algorithm below. As we discussed above, the main challenge is to find a good routine to generate a new candidate for the $\vec{x}_{new}$ which should be related to the previous best $\vec{x}$. We do not want to randomly sample arbitrary positions of the problem space.

Recall that $\vec{x}$ generally looks like $[0, 1, 1, 0, 1, \cdots, 0, 1, 1]$, so we should randomly toggle or mutate a small subset of the bits. We do it randomly[13], so we do not need to keep track of flipped or not flipped positions. This is done by the `change_x` subfunction in listing 1.9.

The rest is quite straightforward, as long as we remember that we are looking for the maximum value in the backpack. The Metropolis algorithm is designed for merit function minimization. So, we choose our merit function to be the negative value of all items in the backpack. Note that a random mutation could lead to a state with an overfilled backpack. So, we need to add a penalty for the case of the overfilled backpack. The positive number proportional to the overfilled portion is a good choice, which is a good way to send feedback to the minimization algorithm that such states are not welcomed.

Words of wisdom

The penalty points, which are usually taken for incorrect homework assignment completion, should be called the negative feedback strength points. They help students to see how far from the optimum they are. Also, control theory teaches us that the most effective feedback is negative feedback.

All of these points are taken care of in the `backpack_merit` subfunction shown in listing 1.9. The rest is just bookkeeping and the straightforward realization of the simulated annealing algorithm. The code for the backpack problem implementing this method is shown below

Listing 1.9: `backpack_metropolis.m` (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/code/backpack_metropolis.m)

```
function [items_to_take, max_packed_value, max_packed_volume] =
    backpack_metropolis( backpack_size, volumes, values )
% Returns the list of items which fit in the  backpack and have the maximum
    total value.
```

[12] If it is too fast we will get stuck in a local minimum and if it is too slow we will waste a lot of CPU cycles by probing around global minimum.

[13] If you have a choice to make and you cannot reason which is better, then make a decision by a coin flip, i.e. randomly. After all, any solution is better than none.

```
% Solving the backpack problem with the simulated annealing (aka Metropolis)
    algorithm.
% backpack_size — the total volume of backpack
% volumes       — the vector of items volumes
% values        — the vector of items values

N=length(volumes); % number of items

function xnew=change_x(xold)
    % x is the state vector consisting of the take or no take flags
    % (i.e. 0/1 values) for each item
    % The new vector will be generated via random mutation
    % of every take or no take flag of the old one.
    flip_probability = 1./N; % in average 1 bit will be flipped
    bits_to_flip = (rand(1,N) < flip_probability );
    xnew=xold;
    xnew(bits_to_flip)=xor( xold(bits_to_flip) , 1 ); % xor operator flips
       the chosen flags
    if ( any( xnew ~= xold) )
           % at least 1 flag is flipped, so we are good to return
           return;
       else
           % none of the flags is flipped, so we try again
           xnew=change_x(xold); % recursive call to itself
       end
end

function [E,  items_value, items_volume] = backpack_merit(x, backpack_size,
   volumes, values)
    % Calculates the merit function for the backpack problem
    items_volume=sum(volumes .* x);
    items_value=sum(values  .* x);
    % The Metropolis algorithm is the minimization algorithm,
    % thus, we flip the packed items value (which we are maximizing)
    % to make the merit function to be minimization algorithm compatible.
    E= - items_value;

    % we should take care of the situations when the backpack is overfilled
    if ( (items_volume > backpack_size) )
        % Items do not fit and backpack, i.e. bad choice of the input vector
            'x'.
        % We need to add a penalty.
```

```
        penalty=(items_volume-backpack_size); % overfill penalty
        % The penalty coefficient (mu) must be quite big,
        % but not too big or we will get stack in a local minimum.
        % Choosing this coefficient require a little tweaking and
        % depends on size of backpack, values and volumes vectors
        mu=100;
        E=E+mu*penalty;
    end
end


%% Initialization
% the current 'x' is the best one, since no other choices were checked.
xbest=zeros(1,N);
[Ebest, max_packed_value, max_packed_volume]=backpack_merit(xbest,
   backpack_size, volumes, values);

Ncycles=10000; % number of annealing cycles
kT=max(values)*5; % should be large enough to permit even large and non
   optimal merit values
kTmin=min(values)/5; % should be smaller than the smallest step in energy
% we choose annealing coefficient by solving: kTmin=kT*annealing_coef^
   Ncycles
annealing_coef= power(kTmin/kT, 1/Ncycles); % the temperature lowering rate

best_energy_at_cycle=NaN(1,Ncycles); % this array is used for illustrations
   of the annealing

% the main annealing cycle
for c=1:Ncycles
    xnew=change_x(xbest);
    [Enew, items_value_new, items_volume_new] = ...
            backpack_merit(xnew, backpack_size, volumes, values);

    prob=rand(1,1);
    if ( (Enew < Ebest) || ( prob < exp(-(Enew-Ebest)/kT) ) )
        % Either this point has smaller energy
        %  and we go there without thinking
        % or
        %  according to the Metropolis algorithm
        %  there is the probability exp(-dE/kT) to move away from the
           current optimum
```

```
        xbest = xnew;
        Ebest = Enew;
        max_packed_value=items_value_new;
        max_packed_volume=items_volume_new;
    end
    % anneal or cool the temperature
    kT=annealing_coef*kT;

    best_energy_at_cycle(c)=Ebest; % keeping track of the current best
       energy value
end
plot(1:Ncycles, best_energy_at_cycle); % the annealing illustrating plot
xlabel('Cycle number');
ylabel('Energy');

% the Metropolis algorithm can return a non valid solution,
% i.e. with combined volume larger than the volume of the backpack.
% For simplicity, no checks are done to prevent it.
indexes=1:N;
items_to_take=indexes(xbest==1);

end
```

At first we will test our code with the same inputs as we did for the binary search algorithm in section 1.5.1

```
>> backpack_size=7;
>> volumes=[  2,  5,  1,  3, 3];
>> values =[ 10, 12, 23, 45, 4];
>> [items_to_take, max_packed_value] = ...
            backpack_metropolis( backpack_size, volumes, values)

  items_to_take = [1 3 4]
  max_packed_value = 78
```

As you can see, the result is exactly the same as in the case of the search over the full combinatorial space. This is not too surprising, since we did 10000 cycles of probing (or annealing) for the problem with only 5 items whose parameter space is $2^5 = 32$. Let's test it with the 20 items problem:

```
>> Vb=35;
>> val = [ 12 13 22 24 97 30 21 67 91 43 36 10 52 30 15 73 43 25 55  6 ];
>> vol = [ 20 27 34 23  4 22 32  2 30 34 34 24  8 23 18 30 14  4 27 22];
>> tic; [items, max_val, max_vol] = backpack_binary(Vb, vol, val); toc
```

```
        Elapsed time is 23.823041 seconds.
>>      items
  items = 5     8    13    17    18
>> max_val
  max_val = 284
>> tic; [items, max_val, max_vol] = backpack_metropolis(Vb, vol, val); toc
        Elapsed time is 0.515279 seconds.
>> items
        items = 5     8    13    17    18
>> max_val
  max_val = 284
```

As we can see, both algorithms produced the same result, i.e. the list of items to choose and the maximum packed value of 284. Your answer might be slightly different when you run `backpack_metropolis`, since there is a small probability that on the last step the algorithm would end up with less favorable energy state (i.e. away from optimal). The binary search takes more than 20 seconds, while the simulated annealing search takes only a half of a second. The best part is that, even for the larger problem with more items to choose from, it will still take only a half of a second. So the small probability to get sub optimal, but still very good, results is a small price to pay for the drastic increase in speed.

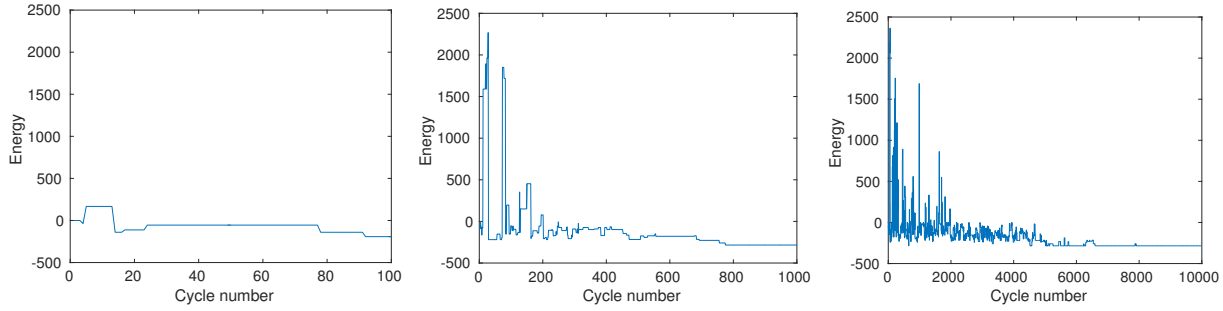

Figure 1.10: The current lowest energy state vs. the annealing cycle number for different total number of annealing cycles: 100 (left), 1000 (middle), and 10000 (right).

You might have noticed that the `backpack_metropolis` function produces the plot of the merit or energy of the state used at the given cycle number of the annealing. This plot is very useful to help in judging if the speed of annealing is chosen properly. Let's have a look at fig. 1.10. These plots are generated for the above problem with 20 items and exactly the same code of `backpack_metropolis` with only one difference: the number of cycles `Ncycles` chosen was 100, 1000, and 10000. When we choose to do only 100 cycles, the algorithm quickly locked itself in a local minimum (as shown in the left insert), with energy somewhat higher than the lowest possible energy of -284. For the 1000 cycles case (shown in the middle insert), the algorithm explores energy space and went quite high in energy, but after about 200 cycles, it starts to search around the global minimum. In the last case of 10000 cycles

(shown in the right insert), the story is somewhat similar, except we converged to global minimum only after about 6000 cycles.

So, we would say that 100 cycles are too few to cool the system sufficiently. The 10000 cycles case seems to be cooling too slowly since we spend a lot of cycles wandering around. However, probability to end up in the global minimum is the highest in this case. The 1000 cycles case seems to be the best for this particular set of input parameters, since we find a good answer much faster than with 10000 cycles (and the solution's energy seems to be the same), while a run with 100 cycles produces a quick, but sub optimal, solution.

Generally, we would like the state energy to behave similarly to the middle and the right insert of fig. 1.10, where the energy oscillates at the beginning and then moves mostly downwards toward the minimum (global or maximum). Behavior like this is the sign of the proper choice of the annealing rate.

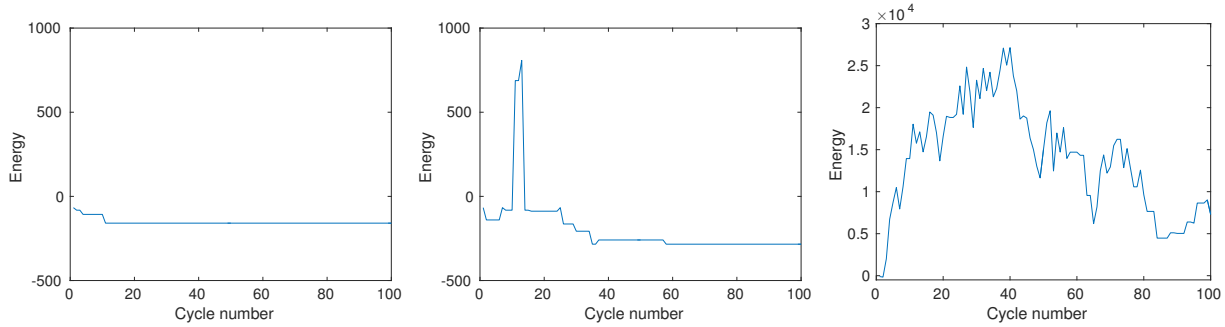

Figure 1.11: The current lowest energy state vs. the annealing cycle number for different annealing temperatures: in the left insert the temperatures 1000 times smaller than in the middle, and in the right insert the temperature is 1000 times larger. In all cases, the simulation is ran for 100 cycles.

Another tricky part is the proper selection of the initial and final temperatures. Have a look at fig. 1.11. These plots are all produced with the same code as in listing 1.9 but with only 100 overall cycles. In one case, we reduce both the initial and final temperatures by 1000 times, and in the other case both temperatures are 1000 times higher. For the case of the low temperatures (the left insert), the algorithm locks itself in the local minimum which is higher than the global one. You can see it by the presence of only downward changes in the energy plot. For the higher temperature case (the right insert), the algorithm is continuously jumping up and down, since for high temperatures, upward motion is almost as likely as downward motion. Essentially, the temperature is never cold enough for the system to settle in any minimum. It is clear that the resulting solution for this case is the worst one: notice the scale for the energy and the fact that final energy is positive, i.e. this is an illegal solution with an overfilled backpack. The middle insert, with the intermediate temperature settings, shows the behavior when the temperature parameters are better tuned: the energy climbs out of the local minimum around cycle 15, and then has **mostly** downhill dynamics with decreasing energy, as we approach the end of the annealing.

1.6 Genetic algorithm

The idea of the genetic algorithm is taken from nature, which is usually able to find the optimal solution via natural selection (see [2]). This algorithm has many modifications but the main idea is shown below

Genetic algorithm

1. Generate a population (set of $\{\vec{x}\}$).
   - It is up to you to decide how large this set should be.
2. Find the fitness (i.e. merit) function for each member of the population.
3. Remove from the pool all but the most fit.
   - How many should stay is up to heuristic tweaks.
4. From the most fitted[a] (parents) breed a new population (children) to the size of the original population.
5. Repeat several times starting from step 2.
6. Select the fittest member of your population to be the final solution.

[a] We are literally implementing "the survival of the fittest". Thus, the name for the merit or energy function is "fitness" for the genetic algorithm.

As usual, the trickiest part is to generate a new $\vec{x}$, i.e. a child, from the older ones. Let's use the recipe provided by nature. We will refer to $\vec{x}$ as a chromosome or genome (hence the name of the algorithm).

Generation of the children's genomes

1. Choose two parents randomly from the most fit set.
2. Crossover or recombine parents chromosomes: take genes. (i.e. $\vec{x}$ components) randomly from either of the parents and assign them to the new child chromosome.
3. Mutate (i.e. change) randomly some of the child's genes.

Some algorithm modifications allow parents to be in the new cycle of selection, others eliminate them in the hope of moving away from a local minimum.

To find a good solution, you need a large population, since this lets you explore a larger parameter space. Think about the evolution strategies of microbes versus humans. However, this in turn leads to a longer computational time for every selection cycle. A nice feature of the genetic algorithm is that it suits the parallel computation paradigm: you can evaluate the fitness of each child on a different CPU and then compare their fitnesses.

As in any other optimum search algorithm, except the full combinatorial search, the genetic algorithm is not guaranteed to find the global optimum in finite time.

1.7 Self-Study

General comments:

- Do not forget to run some test cases.

Problem 1: Prove (analytically) that the golden section algorithm R is still given by the same expression even if we need to choose $a' = x_1$ and $b' = b$.

Problem 2: Assume that the initial spacing between initial bracket points is h. Estimate (analytically) how many iterations it requires to narrow the bracket to the $10^{-9} \times h$ space.

Problem 3: Implement the golden section algorithm. Do not forget to check your code with simple test cases. Find where the function $E1(x) = x^2 - 100 * (1 - \exp(-x))$ has a minimum.

Problem 4: For the coin flipping game described in section 12.2, find the optimal (maximizing your gain) betting fraction using the golden section algorithm and Monte Carlo simulation. Feel free to reuse provided complimentary codes.

Note: you need a lot of game runs to have reasonably small uncertainty for the merit function evaluations. I would suggest to average at least 1000 runs with the length of 100 coin flips each.

Problem 5: Find the point where the function

$$F(x, y, z, w, u) = (x - 3)^2 + (y - 1)^4 + (u - z)^2 + (u - 2 * w)^2 + (u - 6)^2 + 12$$

has a minimum. What is the value of $F(x, y, z, w, u)$ at this point?

Problem 6: Modify the provided traveling salesman combinatorial algorithm to solve a slightly different problem. You are looking for the shortest route which goes through all cities, while it starts and ends in the same city (the first one), i.e. we need a close loop route.

Coordinates of the cities are provided in the `'cities_for_combinatorial_search.dat'`[14] file: the first column of the data file corresponds to 'x' coordinate and the second one to 'y' coordinate. The coordinates of the city where the route begins and ends are in the first row.

Provide your answers to the following questions:

- What is the sequence of all cities in the shortest route?
- What is the total length of the best route?
- Provide the plot with the visible cities' locations and the shortest route.

[14] The file is available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/data/cities_for_combinatorial_search.dat

Problem 7: Implement the Metropolis algorithm to solve the above problem. A good way to obtain a new test route is to randomly swap two cities along the route. You need to choose the number of cycles and initial and final temperature (kT). Provide the reasons for your choices.

As a test, compare this algorithm's solution with the above combinatorial solution.

Now load the cities coordinates from the `'cities_for_metropolis_search.dat'`[15] file. Find the shortest route for this set of cities.

- What is the sequence of all cities in the shortest route?
- What is the total length of the best route?
- Provide the plot with the visible cities' locations and the shortest route.

[15]The file is available at http://physics.wm.edu/programming_with_MATLAB_book/ch_optimization/data/cities_for_metropolis_search.dat

Bibliography

[1] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, Jan. 1962.

[2] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. 1 edition, 1859.

[3] V. Granville, M. Krivanek, and J. P. Rasson. Simulated annealing: a proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):652–656, Jun 1994.

[4] D. E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 3 edition, 2011.

[5] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *J. Chem. Phys.*, 21:1087–1092, June 1953.

[6] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.