

In this document, we describe how to run simplex on a linear program.

Please look in your text on p. 283 for the pseudo-code for Simplex. This is labeled “Algorithm 8.1 Basic Simplex Method.” In what follows, we will refer to this pseudo-code.

Please make sure you have Matlab running so that you can follow the tutorial along with the document. We do exercise 8.1 on p. 308 in this tutorial. The canonical form for this problem can be written as:

$$\begin{array}{rcll}
 \max & 8x_1 + 9x_2 + 5x_3 & & \\
 \text{s.t.} & x_1 + x_2 + 2x_3 + x_4 & & = 2 \\
 & 2x_1 + 3x_2 + 4x_3 & +x_5 & = 3 \\
 & 6x_1 + 6x_2 + 2x_3 & & +x_6 = 8 \\
 & x_1, x_2, x_3, x_4, x_5, x_6 & & \geq 0
 \end{array}$$

In a Matlab editor, we start a *script*, which is a text file with a list of commands for Matlab to run. Here is the beginning of the Matlab script file, tutsimp.m:

```

%%this script corresponds to the tutorial, matlab-simplex-tutorial

%%setting up the problem data:
A = [1 1 2 1 0 0;2 3 4 0 1 0;6 6 2 0 0 1];
b = [2;3;8];
c = [8;9;5;0;0;0];

```

This code has the two comments in green, followed by commands which set up the matrix, A, and vectors b and c. With this in mind, we can start computing Simplex. We need to first execute Step 1 (from p. 283), which is initialization. For this problem, we will use the initial basic feasible solution corresponding to using $x_4, x_5,$ and x_6 as basic variables. Thus, our initial basis is $\mathcal{B} = \{4, 5, 6\}$. In Matlab, we use the following code:

```

%%setting up the initial basis
sB = [4 5 6]

```

Now we need to determine the matrix B and solve for the basic feasible solution. To do so, we use the code:

```

%%finding the initial basis matrix
mB = A(:,sB);
%%solving for the inital basic feasible solution
x = zeros(6,1);
x(sB) = mB^-1 * b;

```

Now, we do something not done in class – we will report what the inital basis and basic feasible solution is. To do so, we use the fprintf command. We will write what the program has done and what the inital bfs is. Here is the code.

```

%%report on our initial bfs
fprintf('Basis elements are:\n');
fprintf('%d ',sB);
fprintf('\n');

fprintf('Initial bfs is the vector:\n');
fprintf('%f\n',x);

```

Now we can start “pivoting”, which, in the algorithm on p. 283, consists of Steps 1 through 4. In our algorithm from class, this is a single iteration of the While loop. We write some code to indicate this:

```

%% start "pivoting"
fprintf('pivot 1\n');

```

In step 1, we need to check, for each simplex direction $\mathbf{d}^{(j)}$, the gradient dot product, $\mathbf{c}^\top \mathbf{d}^{(j)}$. Recall from class that this is the reduced costs, \bar{c}_j , where

$$\bar{c}_j = \mathbf{c}^\top \mathbf{d}^{(j)} = c_j - \mathbf{c}(\mathcal{B})^\top B^{-1} A_j.$$

Instead of using a loop to calculate these in Matlab, we use a vector calculation as follows:

```
%%Step 1:
%%we do not calculate the simplex directions, but just
%%calculate reduced costs:
cbar = c' - c(sB)'*mB^-1*A;
```

Now we determine a potential pivot element, i.e., an entering variable.

```
%%determine the pivot element, which
%%is the index corresponding to the maximum
%%component of cbar
[cbarval p] = max(cbar);

%%report this
fprintf('current_max_reduced_cost_is_%f_with_index%d\n', cbarval, p);
```

Now we can check the optimality condition by observing that if the maximum reduced cost is less than “zero”, then the current basic feasible solution is optimal. This is Step 2 in the pseudo code.

```
%%Step 2:
%%check to see if we're optimal:
%%because of precision issues, matlab does not know
%%what "zero" is, so we have to check against a small
%%positive number
if (cbarval < .00001)
    fprintf('current_basis_is_optimal!\n');
    fprintf('reduced_cost_vector_is:\n_cbar=');
    fprintf('%f', cbar);
    fprintf('\n');
end
```

As the current bfs is not optimal, this code will not produce anything, so we continue with the script. Step 2 continues by determining the entering variable. We are already using the entering variable p , so we calculate the direction this corresponds to. Recall that this calculation is that

$$d(\mathcal{B}^{(t)}) = -(\mathcal{B}^{(t)})^{-1} A_p$$

for $i \notin \mathcal{B}^{(t)}$,

$$d_i = \begin{cases} 1 & i = p \\ 0 & \text{otherwise.} \end{cases}$$

The corresponding Matlab code is:

```
%%note that selecting the entering variable has been done:
%%we're using p
%%Calculate the direction corresponding to p
d = zeros(6,1);
d(sB) = -mB^-1*A(:,p);
d(p) = 1;
%%report this:
fprintf('the_improving_direction_is:\n');
fprintf('%f\n', d);
```

Step 3 starts with a check whether the direction is unbounded:

```
%%Step 3
%%check to see if the direction is unbounded
```

```

if (d >= -.00001)
    fprintf('unbounded!\nSTOP!\n');
end

```

As it is not, we continue by determining the step size. Recall that we now have a vector \mathbf{d} and are trying to find the maximum λ such that

$$\mathbf{x}^{(t)} + \lambda \mathbf{d} \geq 0.$$

Thus, for each index $i \in \mathcal{B}^{(t)}$ this means that for $d_i < 0$,

$$\lambda \leq \frac{x_i^{(t)}}{-d_i}.$$

Note that if $d_i \geq 0$, then the component is not limited by the direction as $x_i + \lambda d_i \geq 0$ for all $\lambda \geq 0$. To compute and print this in matlab, we use the code:

```

%%determine step size and the limiting component
fprintf('x(B)_i/(-d)_i=%f\n', x(sB)/(-d(sB)));

```

This results in the output:

```

(x(B)_i/(-d)_i = 2.000000
(x(B)_i/(-d)_i = 1.000000
(x(B)_i/(-d)_i = 1.333333

```

Thus the minimum is 1, so we set $\lambda = 1$. The second line has the minimum which corresponds to the second **entry in the basis set**, i.e., $q = 5$. The variable which corresponds to this entry is x_5 . The corresponding Matlab code is:

```

%%the minimum element corresponds to the second entry into sB
%% so
l = 2;
q = sB(l);
%%step size is the min entry
lambda = x(q)/-d(q);

%%record this
fprintf('leaving index is %d\n', q);

```

Thus, the new basis is

$$\mathcal{B}^{(1)} = (\mathcal{B}^{(0)} \cup \{2\}) \setminus \{5\} = \{4, 2, 6\}.$$

and the new basic feasible solution is

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \lambda \mathbf{d} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 2 \end{pmatrix}.$$

The corresponding Matlab code is:

```

%%Step 4
%%update the basis and solution
sB(l) = p;
mB = A(:, sB);
x = x + lambda*d;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%one pivot done.
fprintf('After the first pivot, our basis is\n');
fprintf('%d\n', sB);
fprintf('\n');
fprintf('and our bfs is:\n');
fprintf('%f\n', x);
fprintf('\n\n\n');

```

The rest of the script repeats Steps 1-4 on the new basis and terminates with the optimal basis. Note that we are running Simplex “by hand” so we have to run the code piece by piece when calculating the optimal solution.

Here is the full output from running the script:

```
>> tutsimp

sB =

      4      5      6

Basis elements are:
4 5 6
Initial bfs is the vector:
0.000000
0.000000
0.000000
2.000000
3.000000
8.000000
pivot 1
current max. reduced cost is 9.000000 with index 2
the improving direction is:
0.000000
1.000000
0.000000
-1.000000
-3.000000
-6.000000
(x(sB))_i/(-d)_i = 2.000000
(x(sB))_i/(-d)_i = 1.000000
(x(sB))_i/(-d)_i = 1.333333
leaving index is 5
After the first pivot, our basis is
4 2 6
and our bfs is:
0.000000
1.000000
0.000000
1.000000
0.000000
2.000000

Pivot 2
current max. reduced cost is 2.000000 with index 1
the improving direction is:
1.000000
-0.666667
0.000000
-0.333333
0.000000
-2.000000
(x(B)_i/(Binv Ap)_i = 3.000000
(x(B)_i/(Binv Ap)_i = 1.500000
(x(B)_i/(Binv Ap)_i = 1.000000
leaving index is 6
```

After the second pivot, our basis is

4 2 1

and our bfs is:

1.000000

0.333333

0.000000

0.666667

0.000000

0.000000

Pivot 3

current max. reduced cost is 0.000000 with index 2

current basis is optimal!

reduced cost vector is:

cbar = 0.000000 0.000000 -1.000000 0.000000 -1.000000 -1.000000

The optimal basis is:

4 2 1

and our optimal bfs is:

1.000000

0.333333

0.000000

0.666667

0.000000

0.000000