Outline of MATLAB Recitation for Course 6.867

1 Basics

We suggest you quickly skim through this section, and see whether there is anything that you are not familiar with. You can find the detailed explanation of each function in the "Getting started" guide that we have posted on the wiki.

1. Creating and manipulating matrices, and accessing elements.
   (a) Creating vectors: `colon syntax`, `linspace`
   (b) Creating matrices: `zeros`, `ones`, `eye`
   (c) Creating random matrices: `rand`, `randn`
   (d) Individual element access
   (e) File IO: `load`, `save`
   (f) Concatenation: `horzcat`, `vertcat`, `cat`
   (g) Slicing the subscripting: select row(s), column(s), or a block, using integer subscripts.
   (h) The `find` function, and logical subscripting
   (i) Other useful manipulation methods: `reshape`, `repmat`, `sub2ind`, `A(:)` (flatten), `fliplr`, `flipud`.
   (j) Sorting: `sort`.
   (k) Dealing with diagonal matrix: `diag`.
2. Computation and Linear algebra.
   (a) Transposition
   (b) Element-wise basic Calculation: `+`, `-`, `.*`, `./`, `.^`.
   (c) Broadcasting calculation: `bsxfun` (e.g. add a row to each row of a matrix).
   (d) Element-wise evaluation of basic functions, e.g. `log(A)`.
   (e) Reduction: `sum`, `mean`, `max`, and `min`, and `prod`.
   (f) Partial reduction: `cumsum`.
   (g) dot product (`dot`) and matrix multiplication: `A * B`.
   (h) Inverse (`inv`) and solving linear system: `A \ B`, `A / B`.
   (i) Determinant: `det(A)`.
   (j) Eigenvalues and eigenvectors: `eig`.
3. Other useful data types (optional)
   (a) Strings: creating a string, getting substring, joining strings, comparing (`strcmp`), and formatting (`sprintf`).
   (b) Cell array: creating (`cell`) and element accessing (`C{i}`).
   (c) Struct and struct array: creating and field accessing.
4. Plotting

(a) Basic plotting: `plot`.

(b) Basic annotation: `xlabel`, `ylabel`, `title`, and `legend`.

(c) Adding graphs: `hold on` and `hold off`.

(d) Basic axis control: `axis([xmin xmax ymin ymax])`, `axis equal`, and `axis square`.

2 Problems

In the following problems, you are required to implement some simple functions. Hopefully, you can get some basic idea of how to write vectorized code in MATLAB by going through these problems. We encourage you to come out with your own solutions, and then compare them with the ones that we provide here.

2.1 Regularization

Given an $n \times n$ matrix C, add a scalar a to each diagonal entry of C.

Solution 1

```
C = C + a * eye(n);
```

Solution 2

```
idx = sub2ind([n n], 1:n, 1:n);
C(idx) = C(idx) + a;
```

While a little bit more complicated, it is much more efficient than solution 1 when n is large.

2.2 Largest Off-diagonal Element

Given an $n \times n$ matrix A, find the largest off-diagonal element.

Solution 1 (this only outputs the maximum value)

```
msk = (eye(1) == 0);  %% creates a mask of off-diagonal elements.
v = max(A(msk));
```

Solution 2 (this also outputs the location of the maximum)

```
I = (1:n)' * ones(1, n);
J = I';
od_idx = find(I ~= J);   %% linear indices of all off-diagonal elements
i_od = I(od_idx);
j_od = J(od_idx);
v_od = V(od_idx);
[v, k] = max(v_od);
i = i_od(k);
j = j_od(k);
```

2.3 Pairwise Computation

Given a vector x of length m, and a vector y of length n, compute $m \times n$ matrices: A and B, such that $A(i, j) = x(i) + y(j)$, and $B(i, j) = x(i) \cdot y(j)$.

Peparation: make sure x is a column, and y is a row.

```
if size(x, 2) > 1;  x = x.'; end
if size(y, 1) > 1;  y = y.'; end
```

To compute A:

```
A = bsxfun(@plus, x, y);
```

To compute B:

```
B = x * y;
```

2.4 Pairwise Euclidean Distances

This is a very good example that I always use when teaching people how to write vectorized codes. Given a $d \times m$ matrix X, and a $d \times n$ matrix Y, compute an $m \times n$ matrix D, such that $D(i,j) = \|x^i - y^j\|^2$, where x^i is the i-th column of X, and y^j is the j-th column of Y.

The solutions evolve as follows.

Solution 1: two-fold for-loop

```
D = zeros(m, n);   % mention that pre-allocation is important
for j = 1 : n
    for i = 1 : m
        D(i, j) = norm(X(:,i) - Y(:,j));
    end
end
```

Solution 2: one-fold for-loop (with the help of bsxfun)

```
D = zeros(m, n);
for j = 1 : n
    Z = bsxfun(@minus, X, Y(:,j));
    D(:, j) = sum(Z.^2, 1)';
end
D = sqrt(D);
```

Solution 3: no for-loop (the best solution). Here, we use the following decomposition

$$\|x^i - y^j\|^2 = \sum_{k=1}^{d} (x_k^i - y_k^j)^2 = \sum_{k=1}^{d} {x_k^i}^2 + \sum_{k=1}^{d} {y_k^j}^2 - \sum_{k=1}^{d} 2x_k^i y_k^j. \tag{1}$$

There are three terms, each can be computed for all $m \times n$ pairs in batch without for-loop.

```
tx2 = sum(X.^2, 1);
ty2 = sum(Y.^2, 1);
Txy = X' * Y;
D = bsxfun(@plus, tx2', ty2) - 2 * Txy;
D = sqrt(D);
```

Simple benchmark shows that solution is $10x$ to $30x$ faster than solution 1 for moderate size problems.

2.5 Compute Mahalanobis Distances

Given a center vector c, a covariance matrix S, and a set of n vectors as columns in matrix X. Compute the distances of each column in X to c, using the following formula:

$$D(i) = (x^i - c)^T S^{-1} (x^i - c). \tag{2}$$

Here, D is a row vector of length n.

Solution 1: naive solution as baseline

```
D = zeros(1, n);
for i = 1 : n
    z = X(:,i) - c;
    D(i) = z' * inv(S) * z;
end
```

Not good: inversing S for n times.
Solution 2: do pre-computation

```
D = zeros(1, n);
invS = inv(S);   % do pre-computation when possible
for i = 1 : n
    z = X(:,i) - c;
    D(i) = z' * invS * z;
end
```

Solution 3: vectorization

```
Z = bsxfun(@minus, X, c);
D = dot(Z, inv(S) * Z, 1);  % 'dot' function performs dot products
                            % for each column.
```

Solution 4: directly solving linear equations is more efficient than doing inverse.

```
Z = bsxfun(@minus, X, c);
D = dot(Z, S \ Z, 1);
```

This is perfect!