# Worksheet 1, for the MATLAB course by Hans G. Feichtinger, Edinburgh, Jan. 9th

**Start MATLAB via: Start > Programs > School applications > Sci.+Eng. > Eng.+Electronics > MATLAB > R2007 (preferably).**

Generally speaking I suggest that you start your session by opening up a diary with the command: `diary mydiary1.m` and concluding the session with the command `diary off`. If you want to save your workspace you my want to call `save today1.m` in order to save all the current variable (names + values).

Moreover, using the HELP command from MATLAB you can get help on more or less every MATLAB command. Try simply `help inv` or `help plot`.

1. Define an arbitrary (random) $3 \times 3$ matrix $A$, and check whether it is invertible. Calculate the inverse matrix. Then define an arbitrary right hand side vector $b$ and determine the (unique) solution to the equation $A * x = b$. Find in two different ways the solution to this problem, by either using the inverse matrix, or alternatively by applying Gauss elimination (i.e. the RREF command) to the extended system matrix $[A, b]$. In addition look at the output of the command `rref([A,eye(3)])`. What does it tell you?

2. Produce an "arbitrary" $7 \times 7$ matrix of rank 5. There are at east two simple ways to do this. Either by factorization, i.e. by obtaining it as a product of some $7 \times 5$ matrix with another random $5 \times 7$ matrix, or by purposely making two of the rows or columns linear dependent from the remaining ones. Maybe the second method is interesting (if you have time also try the first one):

3. Plotting routines are quite simple in MATLAB. `plot(d)` simply plots the data vector $d$ of length $n$ over $1 \ldots n$. If $d$ is real-valued this is done in the usual way, if $d$ is complex-valued, e.g. $x = rand(1, n) + i * rand(1, n)$ then the plot is performed in the complex plane. So try e.g.

   ```
   n = 35; x = rand(1,n) + i * rand(1,n); plot(x,'r*'); title('a plot in the complex plane');
   ```

   See what happens if you do in addition: `axis square; plot(x,'k'); hold off; grid; figure(gcf);` If you want to mark the first and the last point on this polygonal curve specifically add `hold on; plot(x(1),'b*'); plot(x(n),'g*'); hold off; figure(gcf);` This last command "figure(gcf)" brings the last figure to the surface. The command "figure" itself opens up a new figure.

   If a matrix is inserted into the plotting command the entries (columns) are plotted column-wise. Just as an example try: `T = zeros(8,6); T(:) = 4 : 51; plot(T); xlabel('plotting a matrix');` From such a plot you can learn the sequence of colors automatically assigned to the first, second and so on column to be depicted as a graph.

4. Generate an arithmetic progression consisting of $12 + 1$ equidistant points on the interval $[0, 2\pi]$ by resizing the simple arithmetic progression `0 : 1/12 : 1` by the factor `2 * pi`. Recalling then *Euler's formula*

   $$e^{ix} = cos(x) + i * sin(x)$$

   or simple using the fact that the coordinates of a point on the unit circle are expressed as a pair $(cos(x), sin(x)$ for some $x \in [0, 2\pi)$ we can obtain the set of unit roots of order 12 by the simple command $r12 = exp(i * bas)$. To check that we have obtained the right object we can plot it in the complex plane, simply by the command `plot(r12); axis square;`. You may want to mark the corners specifically with the additional command `hold on; plot(r12,'r*'); hold off; figure(gcf);` .

I can/will also make the LATEX code for THIS document and diaries available to you!

How can you realize such a task. Take the unit roots of order 12 above as example. The MATLAB input could be

```
>> format compact
>> diary  r12plot.m
>> r = 0 : 1/12 : 1
r =
         0    0.0833    0.1667    0.2500    0.3333    0.4167    0.5000    0.5833
                                           0.6667    0.7500    0.8333    0.9167    1.0000
>> bas = 2*pi*r;
>> r12 =  exp(i*bas);
>> plot(r12); hold on; plot(r12,'r*'); hold off; title('unit roots of order 12');
>> grid; axis square; figure(gcf);
>> diary off;
```

Then you would edit this file, to allow such a plot for general $n$, not just $n = 12$. Hence we continue, by calling now the command

```
>> edit  r12plot
```

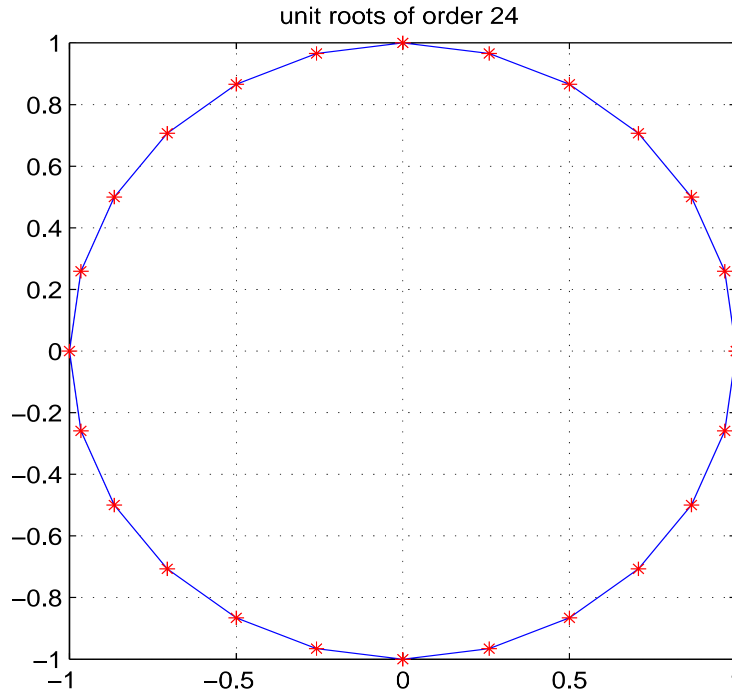
in order to obtain this file:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  PLUROOTS.M
%
% Plot unit-roots
%
% USAGE:  rts =  pluroots(n);

% this is only a comment which will NOT be displayed upon calling
% ``help pluroots''.  Of course the file has to be in your working directory
% in order to be accessible
% ``which  pluroots''  displays that directory
% ``pwd''  prints the current working directory

function   r12 = pluroots(n);

if nargin == 0;  n = 12; end;
r = 0: 1/n :1;
bas = 2*pi*r;
r12 =  exp(i*bas);
plot(r12); hold on; plot(r12,'r*'); hold off;
title(['unit roots of order ' num2str(n) ' ']);
grid; axis square; figure(gcf);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

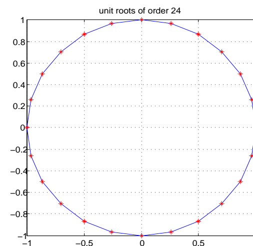
This file can now called as `pluroots(24)` in order to see a similar plot of the unit roots of order 24. You can save and store the figure (by clicking on the icon “file” on the left top the display, then go into export and

choose the mode, such as *.eps or *.jpg or *.pdf format. I usually export into EPS and run a converter called "epstopdf" on the resulting EPS-file, which takes away the unnecessary spaces.

Now we include the thus converted figure into the report:




If you do not go via EPS you get a lot of empty space around, but probably you can fix that using some commands at the time of PDF output from MATLAB. Concrete dimensions obtained experimentally!



# Worksheet 2, for the MATLAB course by Hans G. Feichtinger, Edinburgh, Jan. 16th

1. Write a routine (an M-file which you may want to call testmvmul.m) which checks that the effect of matrix multiplication of some $m \times n$-matrix $A$ with some column vector $x$ is just a linear combination of the columns of $A$ (let us describe them mathematically as $a^k$, as a symbol for the $k$-th column, i.e. check that $A * x = \sum_{k=1}^{n} x_k \mathbf{a}^k$ (recall that you get it via `A(:,k)`).

2. Write a routine defining a "random complex matrix" with entries (both real and imaginary part) equidistributed in the interval $[-1/2, 1/2]$, and call it RANDC.M. Input should be the same as that of `RAND` (call "help rand"), i.e. either `rand(n)` or `rand(m,n)`.

3. Take an arbitrary rectangular matrix and verify that for any pair of complex-valued vectors $x \in C^n$ and $y \in C^m$ one has:
$$\langle A * x, y\rangle = \langle x, A' * y\rangle.$$
*Memorize this by thinking*: moving the matrix to the other side of the scalar product one has to replace the matrix $A$ by $A'$, i.e. one has to add the conjugation and transposition to the matrix (in this way one thinks more symmetrically of this fact).

4. Given a general vector $x \in C^n$ (or for simplicity just in $R^n$) and another vector $y \in R^n$, describe (experimentally) that the orthogonal projection of $y$ onto the one-dimensional subspace generated by $x$ is given by
$$P_x(y) := \frac{\langle y, x\rangle}{\langle x, x\rangle} x = \langle y, \frac{x}{\|x\|}\rangle \frac{x}{\|x\|} = \langle y, x_1\rangle x_1,$$
were $x_1 = x/\|x\|$ is just the *normalized* version of the vector $x$. Hence the orthogonal projection can be written as $y \mapsto y * x1' * x$, or in other words, the $n \times n$-matrix `x1 * x1` (or equivalently `x*x'/x'*x` (!!!) describes the orthogonal projection (as a mapping from $R^n$ into $R^n$.

5. We know that this is giving us the best approximation of $y$ by scalar multiples of $x$ and that the reminder, i.e. $y - P_x(y)$ should be orthogonal to $y$ (hence to all the other vectors in the 1D-space of scalar multiples of $x$). We can test that also experimentally. For simplicity I suggest to work directly with $x1$, i.e. to do the computation with normalized vectors. Recall the Cauchy-Schwartz inequality, which says that the scalar product is never larger than the product of norms:
$$|\langle x, y\rangle| \leq \|x\| \cdot \|y\|.$$

6. Take two vectors in $u, v \in R^7$ of norm 1. Find out for which value $\lambda \in [-1, 1]$ the distance between $v$ and $\lambda u$ is minimal (it should be for $\lambda = \langle v, u\rangle$.

```
>> LAM = -1 : 0.001 : 1;  % possible range of values, in steps of h = 0.01;
>> for jj = 1 : length(LAM); dd(jj) = norm( v - u*LAM(jj)); end;
>> plot(LAM,dd);
>> [mindd, ndmin] = min(dd)  %determining minimal value and index of minimal value
>> norm(v- u*u'*v)
>> norm(v- u*u'*v) - mindd
>> hold on; plot(LAM(ndmin),mindd,'r*'); hold off;
>> grid; title('distance of  v  to  lam * v ');
>> gtext(['minimal distance at ' num2str(LAM(ndmin))]);  % drop near red cross!
```

7. As time permits: check (by some random experiment) that for a random orthogonal matrix (e.g. `U = orth(rand(6,4))` the projection onto the linear span by the 4 column vectors (which in fact equals the $6 \times 6$-matrix `U * U` is the same as the sum of the rank one projections on each of the columns of $U$.

8. Given some collection of vectors in $C^m$, say 4 column vectors in $C^6$, or equivalently a complex $6 \times 4$-matrix (call it $V$). Form another collection of 7 vectors, which are random linear combinations of those 4 column vectors. You can do this by multiplying $V$ *from the right* with some random matrix $U$ $(4 \times 7)$

$$H = V * U.$$

Then (most likely) these new 7 vectors (the columns of H) span the same 4-dimensional space in $C^6$ as the original 4 columns of $U$. How can you verify this? ONE concrete way would be to calculate the projection onto the linear span of the corresponding vectors? (HINT: two families of vectors of equal length span the **same** linear subspace if and only if their projection operators onto their linear span are equal. Recall that one natural way to obtain the projection onto the column-space of $H$ (for example) is best obtained by applying the Gram-Schmidt procedure ($OH = orth(H)$) to $H$ and than build $PH = OH * OH'$. Recall that we have

$$PH * PH = OH * (OH' * OH) * OH = OH * Id * OH' = PH \quad \text{due to orthogonality, and} \quad PH = PH'.$$

Even if the orthonormal basis (here $OH$) is different, the projection is claimed to be the same, and of course the number of members in the orthonormal system is always the same, namely the dimension of the space generated by the ON system.

9. Verify on an arbitrary complex, rectangular matrix that the following two spaces are pairwise orthogonal: the null-space of $A'$ and the range space or column space of $A$ (and by the same reasoning the null-space of $A$ and the column-space of $A'$, usually referred to as the *row-space* of $A$ are orthogonal complements to each other!

   EXTRA comment: Therefore in both cases their dimensions add up to $n$ and $m$ respectively. This explains the well-known dimension formula:

$$defect(A) + rank(A) = n$$

   which is best seen via Gauss Elimination again: There are $r = rank(A)$ Pivot elements, and $n - r$ free variables, determining the dimension of the nullspace of $A$. The MATLAB command `null(A);` provides an orthonormal basis for that null space $= \{x | A * x = 0\}$ directly.

10. Polynomials: Recall the "CONV" command of MATLAB, corresponding to forming the **Cauchy-product** of two polynomials (described by the sequence of coefficients, starting from the highest, non-zero, exponent), and generate **Pascal's triangle**, or equivalently the sequence of binomial coefficients!

11. In order to verify that the central limit theorem is valid try to following. Start with a general discrete probability measure over the integers. This is just an abstract way of saying: choose any finite sequence of non-negative numbers, adding up to one. This could be the sequence `[1,1,1,1,1,1,0]/6`, describing the probability of obtaining one of the numbers $6, 5, 4, 3, 2, 1$ (in decreasing order!, in order to match the conventions of MATLAB about polynomials and powers), and do the following

```
p = rand(1,7);  p = p/sum(p); sum(p)
q = p;  for jj = 1 : 25   q = conv(q,p); stem(q); pause(2); end;
```

Some comments about basic properties of MATLAB programming

1. Generating RANDOM MATRICES

```
A = rand(5,3); disp(A);
```

generates a random matrix, with entries drawn with equal probability from the interval $[0, 1]$.

2. x

3. FOR LOOPS:

```
fak=1; for  kk=1:10;  fak = fak*kk; end; disp(['10 factorial:  ' num2str(fak)]);
```

Here $[\cdots]$ groups partial strings together, while `numstr` converts a variable into its acutally value, displayed as a string;

4. WHILE LOOPS:

```
x=rand(1);jj=1;while abs( x - cos(x)) > 0.001; jj=jj+1; x = cos(x); end; disp(jj);
```

Here we determine how many iterations one has to do in order to find out the fix point of the function $cos(x)$. The actual value can also be determined by plotting the function $cos(x)$ over the interval $[0, \pi/2]$, and see where it intersects the line given by $y = x!$).

5. SORTING:

```
h = rand(1,7),  hs = sort(h,'descend'), [hsort, nds] = sort(h), nds(length(h))
```

shows the following: With a single output the sorting routine sorts the given sequence of real numbers in increasing order (by default); in order to obtain the sequence of indexes (coordinate number) of the sorted elements within the original sequence; hence $nds(7)$ gives the coordinate in which the maximum occurred, while $nds(1)$ would be the index at which the minimum of the sequence occurs in the original sequence. Of course, by the randomness of the sequence $h$ one has created a random index sequence (permutation) $nds$ in such a way;

6. IF STATEMENTS:

```
tic; s = 0; for jj = 1:1000;  r=rand; if  (r > 0.3)*(r < 0.5);s=s+1;end;end; s, toc;
```

verifies for each of 1000 random numbers whether it is simultanuously true that it is larger than 0.3 and smaller than 0.5. Since this is an interval of lenght 1/5 one would expect ca. 20% of the points, so approx. 200 of them in this interval; much faster is however (`tic` and `toc` stop the time!)

```
tic; rr = rand(1,1000);  s1 = sum((rr > 0.3).*(rr < 0.5)), toc;
```

Some comments about basic properties of MATLAB programming, ctd.

Making use of the sorting command to do random permuations (RANDPERM.M):

```
>> n = 6; [sr, rdpm] = sort(rand(1,n)); rdpm
>> [xxx, irdpm] = sort(rdpm); irdpm
>> irdpm(rdpm)
% showing that these two permuations are really doing mutually
% inverse permutations
>> rdpm(irdpm)
```

MATLAB code for the Gram-Schmidt process:

```
% GRAMSFEI.M - Gram-Schmidt orthogonalization.
% Input : A = matrix
% Output     : Q = matrix with orthonormal columns
%                  R = upper triangular matrix
%
% Usage : [Q,R] = gramsfei(A);
%
% Comments : This algorithm is numerically unstable.

% adapted from G.Strang's collection by H.G.Feichtinger
% April 1994, Storrs  fei@tyche.mat.univie.ac.at .
% Renewed by   :    Noha ElAmary  , 08.1999 .

function [Q,R] = gramsfei(A)

[m,n] = size(A);
Asave = A;
tol = 1.e-6;

Q(:,1) =  A(:,1)/norm(A(:,1));
for j = 2:n
 Q(:,j) = A(:,j) - Q*(Q'*A(:,j));
    if norm(Q(:,j)) < tol;
      error('Columns are linearly dependent.');
    end
 Q(:,j) = Q(:,j)/norm(Q(:,j));
 %check = norm(Q'*Q - eye(j)),
end
R = Q'*Asave;
```

Note that this procedure may be unstable, and depends on the order in which the column vectors or coming in. A "symmetric version" (more stable) is given as ORTHBEST.M (NuHAG routine, doing the Loewdin orthonormalization!)

FEW more facts/rules for MATLAB:

- `A = diag([1,2,3]);` defines a diagonal matrix, with main diagonal equal to `[1,2,3]` or `[1,2,3]'`. Conversely, `A = rand(4), dA = diag(A),` pulls the diagonal out of the matrix (!? what else!);
- `s = S(:),` turns any given matrix (hence also row or column vector) into its column version; on the other hand $s = s.'$ does ordinary transposition of a vector, so `s = S(:).'` puts the content of `S` into row format, with the natural MATLAB order of coordinates;
- Given two vectors (or matrices) of equal format, one can take their pointwise product (coordinate by coordinate) by the command `A.*B`;
- `norm(x);` determines the Euclidian norm of a vector; check that for the case of matrices the command `norm(A,'fro')`, determining the so-called Frobenius norm (or Hilbert-Schmidt norm) is the same as `norm(A(:),`.

## Recalling basic facts from LINEAR ALGEBRA

- A vector space is a set endowed with some ADDITION and SCALAR MULTIPLICATION (typically from the field of either real or complex numbers), with corresponding rules (such as the distributive rule, etc. (.. just like in $R^3$)!); Hence (by induction) for any finite sequence of its elements $(v_k)_{k=1}^K$ and a corresponding sequence of scalars $(c_k)_{k=1}^l$ one can form the LINEAR COMBINATION $\sum_{k=1}^l c_k v_k$. Note that the pairing of $c_k$ with $v_k$ is important, not the order of summation.
- The most important examples of the spaces $\mathbf{R}^n$, $\mathbf{C}^k$ (of real resp. complex vectors, be it row or column vectors); the space of Polynomials of a fixed degree $r$, to be written as $\mathcal{P}_r(\mathbf{R})$, or all the Polynomials (of arbitrary degree), denoted here by $\mathcal{P}(\mathbf{R})$.
- Given any set of so-called vectors (elements of a vector space) the smallest linear subspace (itself a vector space) is the set of all possible linear combinations of the given vectors. If those vectors are linear independent
- Using matrices on can easily check whether the columns of a given matrix $\mathbf{A}$ of size $m \times n$ are spanning all of $\mathbf{R}^m$ (resp. $\mathbf{C}^m$), namely iff and only if `rank(A) == m` holds, resp. when those columns are linear independent (if and only if `min(size(null(A))) == 0`, i.e. if the number of column vectors spanning the null-space of $\mathbf{A}$ is ZERO. Obviously a set is a BASIS if and only if EVERY vector can be represented in a UNIQUE way as a linear combination of its vectors (one also calls such as set a "coordinate system"). It is easy to remember that the columns of a *square* matrix $\mathbf{A}$ form a basis for $\mathbf{C}^m$ if and only if $\det(\mathbf{A}) \neq 0$ resp. if and only if $\mathbf{A}$ is invertible (then $\mathbf{A}^{-1} = inv(A)$ in MATLAB can be calculated. Equivalently, Gauss elimination leads to the unit matrix `eye(m)`.
- Expressed in MATLAB language (or in any software package allowing for matrix-vector multiplication) the inhomogeneous linear system $Ax = b$ is solvable if and only if $\mathbf{b}$ is in the range of $\mathbf{x} \to \mathbf{A} * \mathbf{x}$, resp. if and only if $\mathbf{b}$ is in the column space of $\mathbf{A}$.
- A mapping $T$ from $\mathbf{R}^n$ into $\mathbf{R}^m$ is called *LINEAR* if and only if it respects linear combinations. For practical purposes it is enough to check the compatibility with addition an scalar multiplication, i.e. to verify that $T(v + w) = T(v) + T(w)$ and that $T(c \cdot w) = c \cdot T(w)$ holds true, for any two elements $v, w \in \mathbf{R}^n$ and $c \in \mathbf{R}$.

- The connection between the theory of finite dimensional vector spaces (those which *have a finite basis*) and matrix calculus is the following FUNDAMENTAL (deep, although elementary) fact: The linear mappings from $\mathbf{R}^n$ to $\mathbf{R}^m$ (resp. two, possibly different, finite dimensional spaes) are given by matrix multiplication with some (uniquely determined) matrix $\mathbf{A}$, ie. $T(\mathbf{x}) = \mathbf{A} * \mathbf{x}$. In the general setting one has to say: the COORDINATES of $T(\mathbf{x})$ with respect to the basis prevalent in the target spaces can be obtained from the coefficients of the element $\mathbf{x}$ in the domain of $T$ by applying matrix multiplication with such an $\mathbf{A}$! The columns of this matrix are exactly the images of the unit vectors under $T$. And how can we find the matrix $\mathbf{A}$ if $T$ is given, in the general situation? The RECIPE is: just map each of the $n$ basis elements of the domain space over, by applying $T$ to it, that will give you $n$ vectors in the target space, and then expand those image vectors into the basis system over there (this give you $m$ numbers, for each of them), to be written as entries of the corresponding column of $A$.

## Recalling basic facts from LINEAR ALGEBRA, continued! Jan. 22nd, 2008

- Something to recall about Gauss elimination? such as:
  - The goal is the *reduced echelon form*, in MATLAB `rref(A)`! Ideally, starting from a square matrix, you end up with the identity matrix, i.e. (up to numerical error) you have `eye(n)`, which means that the matrix is invertible! (recall that `rref([A, eye(n)])` equals `[eye(n), inv(A)]` !);
  - The steps which are allowed to be carried out are three "elementary deformations", which are ...?
  - Each of these steps is both REVERSIBLE (hence the solution set if preserved) and does NOT change the row-space of the matrix during the process! (not that the NEW rows are just linear combinations of the old ones, reversibility implies that you don't loose any dimension during the process), hence at the end of the process
  - geometrically speaking one has an intersection of so-called hyperplanes (for $d = 3$ each equation is in fact describing an (affine) plane, while the normal equation (i.e. a single equation with $n$ variables) describes some (affine) $n-1$-dimensional subspace of $R^n$. So we are in each step change the concrete hyperplanes (by manipulation on the rows of the system), but do NOT change the solution set, i.e. their intersection!
- THIS PART IS STILL UNFINISHED

# WORKSHEET 3. (Jan. 23rd, 2008, HGFei, MATLAB Course)

SUMMARY of yesterdays course on the "FOUR SPACES":

First let us recall a few simple MATLAB facts:

1. The functions $cos$ and $sin$ are using arclength, so their full period is going from over $[0, 2 * pi]$. Hence there inverse function also go back from $[-1, 1]$ (the range of $cos$, say, over $[0, pi]$, where it is strictly decreasing (mean-value theorem!). It is called $acos(y)$. Test it be thinking of the angle of 45 degrees (or $pi/4$). Not that $cos^2(x) + sin^2(x) = 1$ for any $x$, while for this specific argument $sin(x) = cos(x)$, hence $cos(pi/4) == 1/sqrt(2)$;.

2. It is easy to have multiple parts of a graph, by splitting the figure vertically and horizontally. For example you could plot $sin(x), cos(x), sin^2(x), cos^2(x)$ in four corners of one figure, by using the commands `subplot(221), ..., subplot(224)` consecutively, in order to open the new windows (see HELP subplot!)

3. In order to automatically plot, say the monomials, into 12 windows, one can use the "running" argument to automatically open the right subwindow:

```
  >> bas1 = linspace(0,1,100);
>> for kk = 1: 12; eval(['subplot(3,4,'   num2str(kk) ')']); plot(bas1,bas1.^kk);  end; shg
```

4. Given 2 random vectors in (specifically) $\mathbf{R}^3$ one can determine the normal vector to the plane generated by those two vectors using the cross-product. The core part is

```
% Calculate cross product
c = [a(2,:).*b(3,:)-a(3,:).*b(2,:)
     a(3,:).*b(1,:)-a(1,:).*b(3,:)
     a(1,:).*b(2,:)-a(2,:).*b(1,:)];
```

   Start with two random vectors in $\mathbf{R}^3$, or `M = rand(3,2)`. Verify that the crossproduct $c = cross(M(:,1), M(:,2))$ is a multiple of the vector describing the span of the two column vectors, hence to $null(M')$.

5. One can easily graph curves in the plane, by just providing a sequence of $(x, y)$ coordinates to the system, e.g. (just for fun, you may skip this one
   `bb=linspace(-1,2,600); plot(cos(23*bb),sin(25*bb));axis square;shg`

   More seriously you can plot straight lines by the usual point-vector realization of straight lines. For example, let us plot the line, given by the equation $x + 2y = 4$ or equivalently $y = -x/2 + 2$, e.g. by observing (setting $x$ or $y$ to zero) that $P = (0, 2)$ and $Q = (4, 0)$ are on that line. Hence we can get all the points by taking coordinates of the form $R = P + lam * v$, with $v = P - Q = (4, -2)$ (or of course $v1 = (2, -1)$ or any other multiple of $v$), and $lam \in \mathbf{R}$. For example we could do the following:

```
>> bb=linspace(-1,2,600); % as above
>> for jj = 1 : length(bb); Rlin(:,jj) = P + bb(jj)*v; end;
>> plot(Rlin(1,:), Rlin(2,:)); grid; hold on; plot(0,0,'r*'); hold off;  shg;
```

shows that straight line (we skip doing axes to save time). The QUESTION now is: what is the closest point of this straight line to $(0,0)$, marked with a red star already. Obviously we have to cut the given line with another line (to be plotted in black) perpendicular to the given one, and going through the origin. Since $v == [4,-2]$ we find that $u = [2,4]$ has to be perpendicular (switching positions of entries and taking one minus, otherwise call $null(v')$ (cf. discussion below), and compare to $u/norm(u)$, the normed version of $u$ (should be equal up to sign, at most). So we do

```
for jj=1:length(bb); Qlin(:,jj) = bb(jj)*u; end;hold on; plot(Qlin(1,:),Qlin(2,:),'k');
hold off; axis square;  shg; % to put things into correct perspective
```

Obviously the intersection point $S$ (write `gtext(' < intersection point');` as lying on both straight lines, so its coordinates must satisfy the linear system

```
x + 2y = 4
2x - y  = 0
```

and we can find this solution by applying Gauss elimination to the extended system matrix

```
1  2   4
2 -1  0
i.e. in MATLAB:
>> SM = [1,2,4; 2, -1, 0]
>> rref(SM)
```

reveals the solution (the point $S = (0.8, 1.6)$ does the job), its distance from $(0,0)$ is just `norm(S)`.

Of course one could use Hesse's formula in this concrete situation, but the method described above extends to higher dimensions as well!

Recall that we have FOUR SPACES associated with each matrix, the null-space NULL(A) of $A$ resp. that of $A'$, which we will denote by NULL(A'). These are the solution sets of homogeneous linear equations described by the matrices $A$ and $A'$ respectively. The first sits inside of $\mathbf{R}^n$, the second inside of $\mathbf{R}^m$. Of course each of them got it's orthogonal complement within its ambient space. What is remarkable is, that these are just the row-spaces of $A$ (we write V1: COL(A') because this is "more correct", and V2 : COL(A), the column space of $A$.), the range of the matrix multiplications with $A'$ and $A$ resp. or $A * \mathbf{R}^n$ and $A' * \mathbf{R}^m$.

Without explaining the background of how they are computed etc. let us just recall that they are of equal dimension, and that in principle the combination of Gauss elimination with the Gram-Schmidt procedure would allow us to obtain an orthonormal basis for these two spaces. Fortunately MATLAB provides us with a faster (easier and more efficient) way to obtain such (non-unique) ONBs, just call `O1 = orth(A')` or `O2 = orth(A)` to obtain them. The rank-property mentioned above of course implies that they have an equal number of columns ( namely `rA = rank(A)`).

Having now two orthonormal bases for $V1$ and $V2$ we can describe the orthnormal projections onto those spaces! $P1 = O1 * O1'$ and $P2 = O2 * O2'$.

1. Let us start with the testmatrix $A, 3 \times 3$, obtained via `A = zeros(3); A(:)  = 1:9,` and let us see what we can say about it: For sure it is square, but not invertible for different reasons: 1) $det(A) = 0$; or 2) $nA = null(A)$ is a non-trivial basis for the nullspace of $A$, consisting of exactly one vector, or 3) checking that $rank(A) = 2 = 3 - 1$, by looking at the row-reduced echelon form: In fact, `rref(A)` gives us

```
  >> rref(A)
ans =
     1     0    -1
     0     1     2
     0     0     0
```

Hence `BRA = ans(1:2,:)'` is a system of vectors in $R^n = R^3$ which span the column-space of $A'$. or

```
     >> BRA = ans(1:2,:)'
BRA =
     1     0
     0     1
    -1     2
```

We can easily get an orthonormal bases for the same object by applying Gram Schmidt: `OR = orth(BRA);`check that this is an orthonormal system, by checking that `PROW = OR * OR'` is a symmetric matrix `(PROW == PROW')`, which is idempotent, i.e. $norm(PROW * PROW - PROW$ is negligible, and that it is describing a space perpendicular to the null-space of `A` by looking at `null(A)' * OR`. That this orthonormal projection operator leaves the columns of `A'` invariant is easily seen by looking at PROW * A' (recall that the matrix PROW is multiplied with A' by letting it act on the individual columns!).

We continue by doing alternatives, we could for example take just the second and third row of the original matrix (better, the second and third column of `A'` and then apply Gram Schmidt to it in order to obtain another orthonormal basis for the same space! Worked out this means:

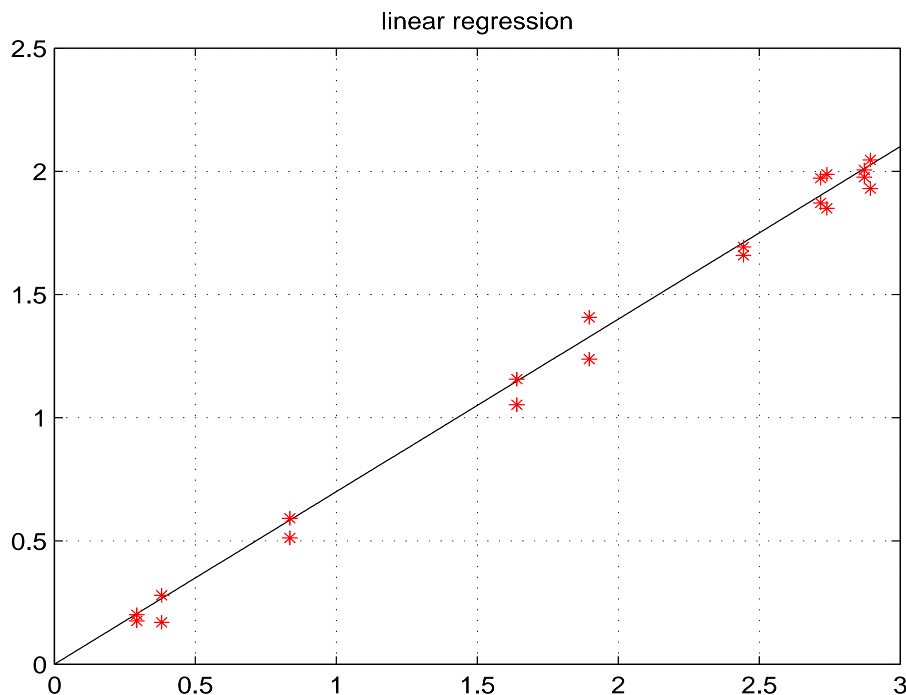
```
>> A',   OR1 = orth(ans(:,2:3)); check = norm( OR1 * OR1' - PROW),
>> check = norm(A * pinv(A) - PROW)
```

The last command then shows that the same projection operator can be obtained (more directly to the user, and in a more stable way) using the PINV command. Just for the sake of exercise check that `Q = pinv(A)*A` leaves the columns of `A'` invariant, and is the projection onto the "row space of `A'`".

2. Next we want to do a regression line. Assume you have data along some line, with some random and/or measure error. We can simulate such data easily using RAND. For example let us look at the "true linear function $y = 0.7 * x$, over $[0, 3]$ say. We can plot it as usual in the following way:

```
bas = 0: 1/100 : 1;  y = 2 + 0.7 * bas; plot(bas,y,'k'); title(' the true function');
```

Next we assume that we take some measurements, at random positions, say $K = 10$ measurements, $x = 3 * rand(K, 1)$, to b sorted via `x = sort(x,'ascend')`. Let the imprecise measurement be of the form $s = 2+0.7*x+0.3*x.*(rand(K, 1)-1/2)$;, so that the error is both negative and positive, and somehow proportional to the actual value. Let us plot them into the same figure: `hold on; plot(x,s,'r*'); grid; shg;` Next we want to find out the straight line from the red (noisy, measured) datapoints in our plot. It should look like:




For this we recall that despite the $K = 10$ datapoints we are only looking for a straight line, i.e. a function of the simple for $y = k * x + d$, or to make it more general $y = a(1) * x + a(2)$ (MATLAB format). The set of all such data points is 2D, namely all the possible linear combinations of the vector `ones(K,1)` and the vector $x$. They are obviously linear independent hence they span a 2D subspace of $\mathbf{R}^K$. In order to project our data sequence $s$ onto this space (i.e. finding an actual sequence of a polynomial of degree 1, or of the form $z = alpha * x + beta$ we have to project onto that 2D-space using pinv! Hence we put

```
A = [x, ones(K,1)];  % respecting MATLAB order of monomials;
coeff =  pinv(A) * s, % displaying the estimated parameters;
projs =  A * coeff; sa =  coeff(1)*x + coeff(2);
plot(bas, coeff(1)*bas + coeff(2),'b'); plot(x, sa,'g+'); shg;
```

The resulting "regression line" and the values on that straight line are then visible (and typically quite close to the original line, especially if the nubmer of measurements is large, or if the error (noise level) is small.

For those who progress fast and look out for a challenge!

1. Generate a random $5 \times 5$ matrix of rank 3 (say), e.g. `A = rand(5,3) * rand(3,5)`. Hence we know that it will not be invertible. The claim is, that nevertheless the matrix can be viewed as a mapping between the three-dimensional row space of $A$ (better: the three-dim. column space of $A'$) and the column space of $A$ (which has the same dimension, since column-rank(A) equals row-rank(A), and is then called the rank of $A$). Check by setting `rA = rank(A)`.

First let us generate two bases for COL(A') and COL(A), the two spaces (of equal dimension!) in discussion. Since Gauss Elimination gives us a basis for the row space we can use the following (don't get confused about rows and columns here): `rref(A'); BASCOL = ans(:,1:rA);` gives us a basis for the column space (again transformed back into column format!) and `rref(A); BASROW = ans(1:rA,:)'` gives us a basis for the row space of $A$, we should say column space of $A'$.

Let us now verify (experimentally) what linear algebra tells us: If we restrict the mapping $\mathbf{x} \rightarrow A * \mathbf{x}$ to the row space of $A$ then it is simply a bijection, i.e. it can be inverted. In other words, whatever element of the column space you start with, e.g. $\mathbf{b} = A * rand(n, 1)$, one can find a (unique) linear combination $\mathbf{u}$ of elements from that row space such that $A * \mathbf{u} = \mathbf{b}$!

How can be go about this? Well, any $\mathbf{u}$ is to be written (uniquely, because the columns of BASROW are a basis! for this space) as $BASROW * \mathbf{c}$, for a set of coefficients $\mathbf{c} \in \mathbf{R}^{rA}$. Hence we are looking for a solution of

$$A * \mathbf{u} = A * (BASROW * \mathbf{c}) = (A * BASROW) * \mathbf{c} = \mathbf{b},$$

which can be obtained by applying Gauss elimination to the extended inhomogeneous system $AEXT = [A * BASROW, b]$, or in other words the last row of `rref(AEXT)` should contain that answer!

Try also to call $ccpinv = pinv(A) * \mathbf{b}$ to see that it is exactly this solution that you get!

The interesting thing with this second way of obtaining it (aside from this being more suitable) is that it also works for an arbitrary right hand side (while the Gauss elimination will certainly do the job if and only if $\mathbf{b}$ is in the column space of $A$)! What it does, is to solve the above system, but - if necessary - with $\mathbf{b}$ being replaced by the ORTHOGONAL PROJECTION of whatever $\mathbf{b}$ is onto the column space of $A$. Hence one could explain what PINV does by saying: Take any $\mathbf{b} \in \mathbf{R}^m$. First project it onto the column space of $A$, e.g. by applying explicitly the orthogonal projection to $\mathbf{b}$, i.e. to `OA = orth(A); bproj = OA * OA' * b;` , and then continue (with $bproj$ instead of $b$ in the above procedure).

## Table of content for Lesson Nr. 2 (Jan. 15th, 2008)

1. MATLAB and Polynomials (quote: `help polyval`), allows to evaluate a polynomial with possibly complex coefficients **a** at a sequence of points on the real line or also the complex plane (if matrices are inserted one should use `polyvalm`).

   Recall that any polynomial over the complex can be split in linear factors, i.e. is a product of factors of the form $(z - z_i)$. When the polynomial is symmetric, then those complex numbers are either real or come in pairs of complex conjugate numbers. ·

   By the command `roots(a)` those linear factors, resp. the zeros (counted with multiplicity) are numerically evaluated resp. presented.

   Also the usual convention of writing integers, say 123 for $1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$ is in accordance with this MATLAB rule. Hence one can multiple "arbitrary long integers" using the `CONV` command, if only the out put is recast in standard format (a nice little excercise to do so, assume you get the output `[2 34 23 45]` you have to start from the end: `45 = 5 + 4 *10` , but `23 + 4 = 27` in turn `10` equals `70 * 10 + 200` etc. etc. . . . )

2. MATLAB does not have a specific call for scalar products, because the scalar product of two column vectors $x, y$ is simply calculated by the command `y' * x`, while it is just `u * v'` if you have row vectors

$u, v \in C^n$. Correspondingly, for each collection $A$ of $n$ column vectors in $C^m$ the collection of all scalar product (the so-called **Gramian matrix**) is obtained as $A' \times A$ (it contains exactly $n^2$ entries.

Note that one can define a family of vectors to be an orthonormal system if and only if `A' * A == eye(m)`

3. Recall that a length of a vector can be calculated from its coordinates with respect to *any* orthonormal basis (resp. orthonormal system with the correct number of vectors = $dim(A)$).

4. TEST `cumprod(1:171)`, what is the problem here! We can't we simply implement the law of the *binomial formula* in order to compute e.g. probabilities for large repetitions of experiments (e.g. coin tossing)? (remember, they are given as $\frac{n!}{k!(n-k)!}$.

   MAYBE SOME MORE COMMENTS ONCE I HAVE SCRIPTS OF THE COURSE

## Table of content for Lesson Nr. 2 (Jan. 22th, 2008)

Gilbert Strang's FOUR spaces, the column spaces of $\mathbf{A}$ and $\mathbf{A}'$ (usually referred to as "row space of A"), which are the spaces generated within $\mathbf{R}^m$ and $\mathbf{R}^n$ respectively, by taking all possible linear combinations of those systems of column vectors are explained geometrically. It is shown that every matrix does a one-to-one correspondence between the so-called ROW-space of $\mathbf{A}$( COL(A') is more appropriate ) and the column space of $\mathbf{A}$, we write COL(A). The pinv-command (the so-called MOORE-Penrose or pseudo-inverse) is inverting such a linear operator or matrix in the best possible way. IF $\mathbf{A} * \mathbf{x} = \mathbf{b}$ is NOT solvable, it makes the equation solvable at the cost of changing the right hand side from $\mathbf{b}$ to $\widetilde{\mathbf{b}} = P(\mathbf{b})$, the projection of $\mathbf{b}$ onto the column space (or image of the mapping $\mathbf{x} \to \mathbf{A} * \mathbf{x}$). This change is minimal in the sense of a minimal distance, which is traditionally described by a SUM OF SQUARES, so we have a MINIMUM for the quadratic function $d = \sum_{k=1}^n |\tilde{b}_k - b_k|^2$, among all elements $\widetilde{\mathbf{b}}$ $\in$COL(A). Having now assured that $\mathbf{A} * \mathbf{x} = \widetilde{\mathbf{b}}$ has a solution we have to choose among the possible solutions a unique one. If NULL(A) was trivial (just the zero-element) we would have been done by now anyway, but let us look at the general situation:

Recall that all solutions of a solvable inhomogeneous linear system differ only by an "arbitrary" solution from NULL(A), the null-space of the matrix (general solution to the homogeneous system). Since COL(A') and NULL(A) are perpendicular subspaces, the *Pythagorean Theorem* tells us, that the partner of $\widetilde{\mathbf{b}}$ in COL(A') (recall that the matrix establishes a bijection between COL(A') and COL(A) !) is the right object, i.e. the solution of the modified with MINIMAL NORM. This explains why the solution assigned to $\mathbf{b}$ via $\mathbf{x} = pinv(A) * \mathbf{b}$ is called the MNLSQ-solution to the problem, the MINIMAL NORM LEAST SQUARES SOLUTION to the original problem. Often one gets to the same result using the so-called *normal equations*, with the argument: IF $\mathbf{A} * \mathbf{x} = \mathbf{b}$ was solvable, we would have $\mathbf{A}' * \mathbf{A} * \mathbf{x} = \mathbf{A}' * \mathbf{b}$. If the columns of $\mathbf{A}$ are linear independent (and only if this is the case) $GR = \mathbf{A}' * \mathbf{A}$, the Gram matrix, is invertible, hence $\mathbf{x} = inv(\mathbf{A}' * \mathbf{A}) * (\mathbf{A}' * \mathbf{b})$ is the appropriate solution. However, it is not hard to show that in general one has (Feichtinger's magic formula):

$$pinv(\mathbf{A}) = \mathbf{A}' * pinv(\mathbf{A} * \mathbf{A}') = pinv(\mathbf{A}' * \mathbf{A}) * \mathbf{A}'$$

which brings us back to the viewpoint that pinv is doing a good job, with the same result as the normal equation approach, however also in more general situations!

Some MORE comments, MATLAB programming II

1. TIMING:

```
format bank;  tic; jj=1; while toc < .5;  jj = jj +1; end;  jjfinal = jj,
```

   This forces MATLAB to display numbers in bank format (specifially good for the full presentation of large integers) an shows how for MATLAB gets whithin half a second on a given machine. For more complicated commands one can see that speed improves as variables and code is then already loaded into memory. The counterpart

```
tic, jj=1; for jj = 1 : jjfinal; jj = jj+1; end, toc
```

   reveals that looking up the time is more costly than counting (and comparable to the time needed to build a $4 \times 4$ or $5 \times 5$ matrix! The commend `date` spells out the date.

2. WORKSPACE:

```
r1 = 1, r2 = rand(2),   whos  r*,
```

   shows all the variables with variable names starting with `r`.

1. SAVING an LOADING:

```
save rvariab   r* ,
clear r1  r2   % NO COMMAS, otherwise it does
% clear r1, r2  % meaning: clear r1 and DISPLAY r2!
whos r*
```

   simple stores all the `r`-name variables in a file named `rvariab.mat` (so $*.MAT$-files are MATLAB storages), from where you can also load selective. Clear deletes a variable from the workspace. The calls `clear, save` etc. applied without argument take ALL the given variables in that situation.

2. ASKING for INPUT:

```
ri = input('Please type a number: ');
```

   suggest to the user at runtime (or from the command line) to provide a number. Obviously this number is then assigned to the variable name `ri`. If there is no left hand argument the variable `ans` is used

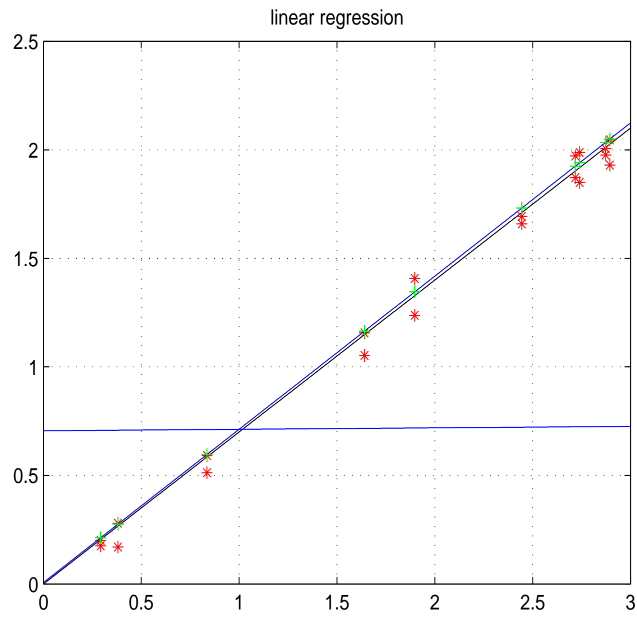
```
[xx,yy] = ginput(4);    %  or ginput(4),  if you want to see the output
```

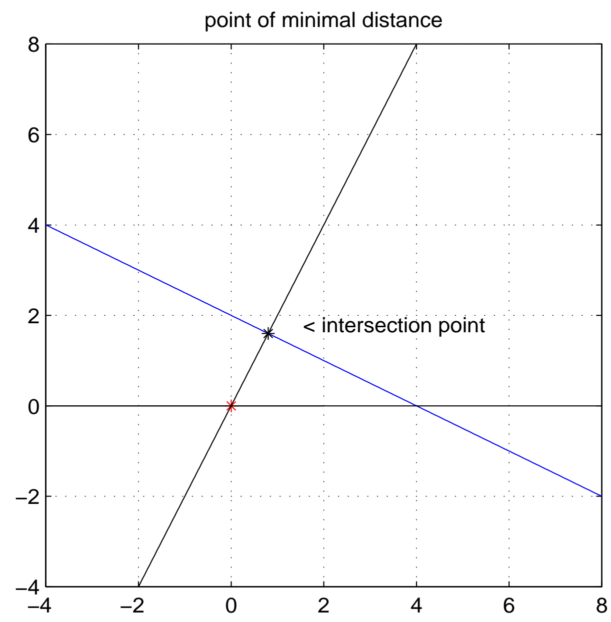
   allows to take coordinates from a plot. The second argument is the number if points to be taken. For example

```
bas = linspace(0,2,301); plot(bas,bas.^);  % plotting a curve
[xx,yy] = ginput(4),  % would be a good use;
```

   task: try to guess the quadratic curve to which those points belong;

3. more to come later (planned), hgfei


linear regression
2.5
2
1.5
1
0.5
0
0
0.5
1
1.5
2
2.5
3


point of minimal distance
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
< intersection point