MATLAB and Linear Algebra basics for Computer Vision

S. Asteriadis – Department of Data Science and Knowledge Engineering – Computer Vision – KEN4255

LINEAR ALGEBRA

Vectors

- A column vector $\mathbf{v} \in \mathbb{R}^{n \times 1}$ is denoted with:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

while, the corresponding transpose $\mathbf{v}^T \in \mathbb{R}^{1 \times n}$ is: $\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}$

- In our class, we will refer, by default, to column vectors

Vector inner product

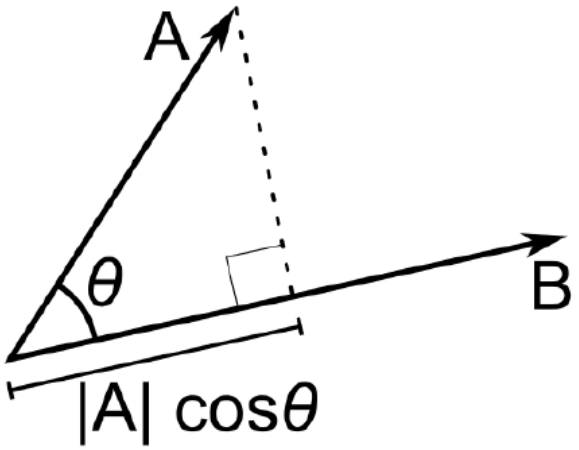
- The inner (or dot) product of two vectors **x, y** is their element-by-element multiplication and summation of results, leading to a *scalar valued* result:

$$\mathbf{x}^T\mathbf{y} = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i y_i$$

- Alternatively, the above can also be $\mathbf{x}\cdot\mathbf{y}=|x||y|\cos(\vartheta)$, where ϑ the angle between **x** and **y**.

Vector innter product

If **B** is a unit vector, then **A** · **B** gives the length of **A** projected on B:



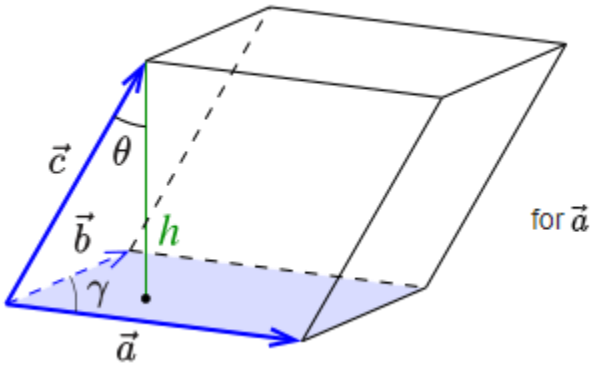
Matrix definition

- A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is an array of values, located on a grid of m rows and n columns
- For m=n, A is square

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & & & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$

Matrix determinant

- The determinant det(A) of a matrix A is a scalar
- In practice, it represents the volume (or area in 2D) of n-dimensional parallelepiped formed by the column (or row) vectors of the matrix



$$V = \left| \det \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{bmatrix} \right|.$$

for $\vec{a} = (a_1, a_2, a_3)^T, \vec{b} = (b_1, b_2, b_3)^T, \vec{c} = (c_1, c_2, c_3)^T$

Matrix trace, identity and diagonal

- Matrix trace tr(**A**) is the sum of the elements on the diagonal (e.g. $\mathrm{tr}(\begin{bmatrix}1 & 3\\ 5 & 7\end{bmatrix}) = 1+7 = 8$)
  - Some properties: tr(**AB**)=tr(**BA**) and tr(**A**+**B**)=tr(**A**)+tr(**B**)
- Identity matrix **I** is a square matrix with 1's along the diagonal and 0's everywhere else.
  - $I \cdot A = A$
- A diagonal matrix has numbers along the diagonal and is 0 everywhere else
  - diagonal · (a matrix) = (that matrix scaled)

Matrix

- Symmetric matrix

$$\mathbf{A}^T = \mathbf{A}$$

$$\begin{bmatrix} 1 & 2 & 5 \\ 2 & 1 & 7 \\ 5 & 7 & 1 \end{bmatrix}$$

- Skew-symmetric matrix

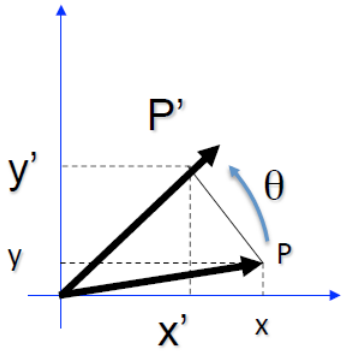
$$\mathbf{A}^T = -\mathbf{A}$$

$$\begin{bmatrix} 0 & -2 & -5 \\ 2 & 0 & -7 \\ 5 & 7 & 0 \end{bmatrix}$$

Transformations

- A matrix can transform a vector through multiplication x'=Ax
- E.g. scaling $\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$
- Rotation by an angle θ:

$$x' = \cos\theta\ x - \sin\theta\ y$$
$$y' = \cos\theta\ y + \sin\theta\ x$$



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\mathbf{P' = R\ P}$$

Compound transformations

- E.g. a scaling followed by two rotations (R_1 and R_2) → $P'=R_2R_1SP$
- Actually, what we do, is: $P'=(R_2(R_1(SP)))$
- We can define the whole transformation first and, then, apply it on the points: $P'=(R_2R_1S)P$

Homogeneous coordinates

- A matrix multiplication, actually allows us to linearly combine components of a vector:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

- This accommodates a number of transformations, such as *scale, skew, rotation,* etc.
- However, notice that we cannot add constants, that is, we cannot perform translations

Homogeneous coordinates

- Suppose we want to translate by *c* and *f* along
- These have to be somehow in the transform
- A hacky solution is to expand the dimensionality of everything by adding a '1' at the end of each vector

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

- Now, we can translate, too. This is called 'homogeneous coordinates'
- A transformation matrix in homogeneous coordinates, in its general form, will have a bottom row in the form of [0 0 1] so the resulting vector also has a 1 as its last element

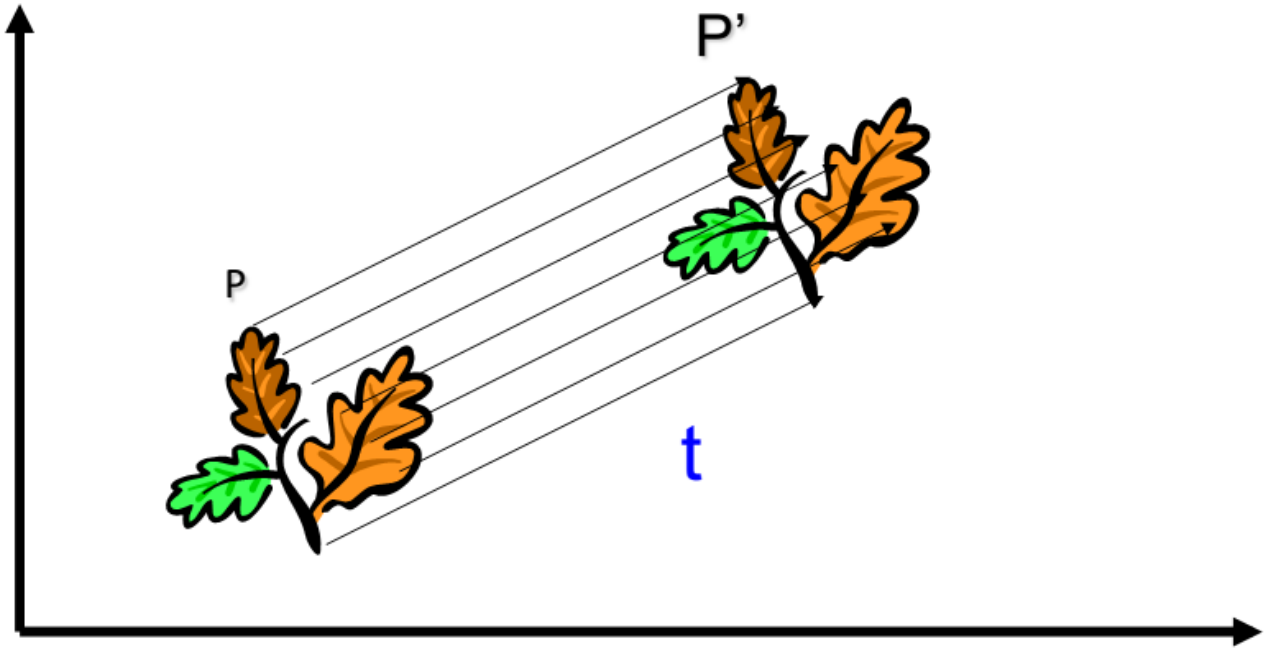
Homogeneous coordinates

- One more thing we might want: to divide the result by something
  - For example, we may want to divide by a coordinate, to make things scale down as they get farther away in a camera image
  - Matrix multiplication can't actually divide
  - So, **by convention**, in homogeneous coordinates, we'll divide the result by its last coordinate after doing a matrix multiplication

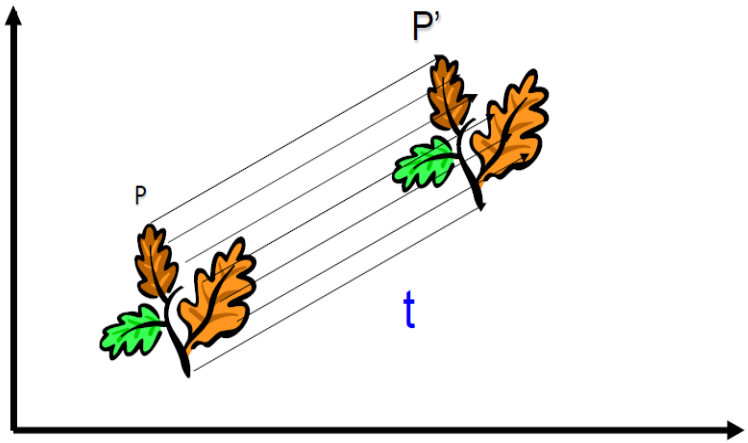
$$\begin{bmatrix} x \\ y \\ 7 \end{bmatrix} \Rightarrow \begin{bmatrix} x/7 \\ y/7 \\ 1 \end{bmatrix}$$

Transformations

2D Translation



2D transformations in homogeneous coordinates



$$\mathbf{P} = (x, y) \rightarrow (x, y, 1)$$

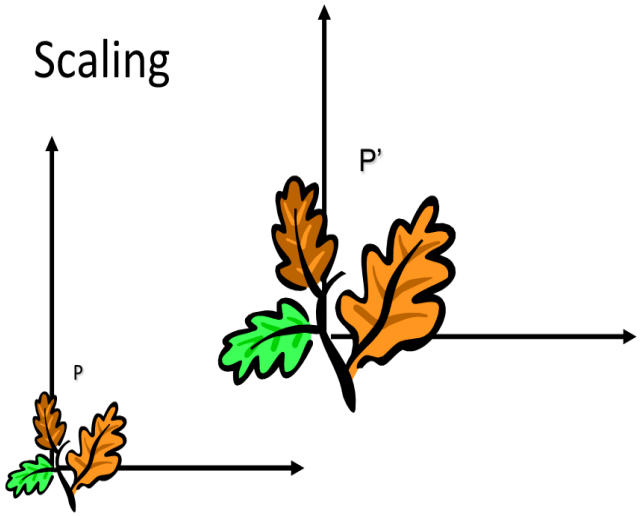
$$\mathbf{t} = (t_x, t_y) \rightarrow (t_x, t_y, 1)$$

t P

$$\mathbf{P'} \rightarrow \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{I} & \mathbf{t} \\ 0 & 1 \end{bmatrix} \cdot \mathbf{P} = \mathbf{T} \cdot \mathbf{P}$$

2D scaling in homogeneous coordinates



$$\mathbf{P} = (x, y) \rightarrow \mathbf{P}' = (s_x x, s_y y)$$

$$\mathbf{P} = (x, y) \rightarrow (x, y, 1)$$

$$\mathbf{P}' = (s_x x, s_y y) \rightarrow (s_x x, s_y y, 1)$$

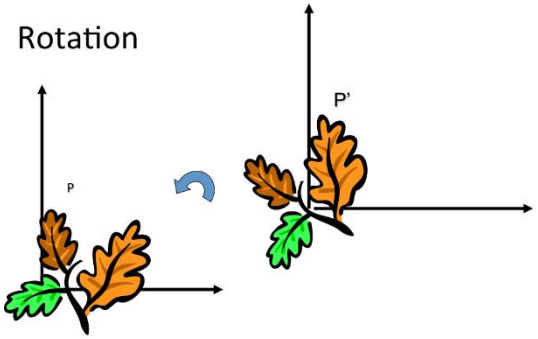
$$\mathbf{P}' \rightarrow \begin{bmatrix} s_x x \\ s_y y \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{S}} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{S}' & \mathbf{0} \\ \mathbf{0} & \mathbf{1} \end{bmatrix} \cdot \mathbf{P} = \mathbf{S} \cdot \mathbf{P}$$

Scaling & Translating

$$\mathbf{P''} = \mathbf{T} \cdot \mathbf{S} \cdot \mathbf{P} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} =$$

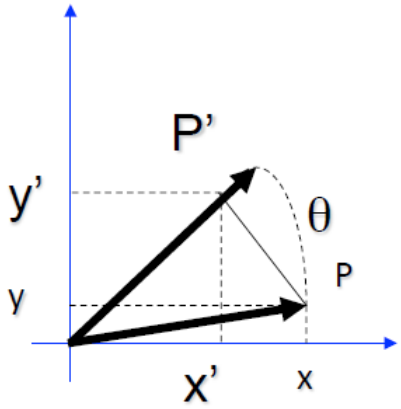
$$= \underbrace{\begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathrm{A}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} s_x x + t_x \\ s_y y + t_y \\ 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} S & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2D rotation in homogeneous coordinates



$$x' = \cos\theta\, x - \sin\theta\, y$$
$$y' = \cos\theta\, y + \sin\theta\, x$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



$$\mathbf{P'} = \mathbf{R}\,\mathbf{P}$$

2D rotation in homogeneous coordinates

- To rotate in the opposite direction, use the transpose of the rotation matrix **R, $\mathbf{R}^T$**
- Note that $\mathbf{R} \cdot \mathbf{R}^T = \mathbf{R}^T \cdot \mathbf{R} = \mathbf{I}$
- The rows (columns) of **R** are orthogonal (i.e. perpendicular to each other), and they are unit vectors.

Homogeneous coordinates

Scaling + Rotation + Translation

$$P' = (T\ R\ S)\ P$$

$$\mathbf{P'} = \mathbf{T} \cdot R \cdot \mathbf{S} \cdot \mathbf{P} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} =$$

$$= \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} S & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} R\,S & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This is the form of the general-purpose transformation matrix

Inverse matrix

- The inverse of a matrix **A** is given by $\mathbf{A}^{-1}$ and it holds that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$
- The inverse does not always exist.
  - If it exists, A is invertible or non-singular
  - If it does not exist, A is singular
- As a quick reminder:

$$(\mathbf{A}^{-1})^{-1} = \mathbf{A}$$

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

$$\mathbf{A}^{-T} \triangleq (\mathbf{A}^{T})^{-1} = (\mathbf{A}^{-1})^{T}$$

Pseudoinverse matrix

- Calculating the inverse of a matrix may be painful, esp. for large matrices (problems with floating-point resolution, as working with very large and very small numbers at the same time may be involved)
- Also, your matrix may not even have an inverse
- Suppose, though, that we have to solve $\mathbf{A}X=\mathbf{B}$ for X

Pseudoinverse

- By simply typing **A \B**, Matlab will look for the best numerical method to solve for X
  - it will return the closest solution if there is no exact one
  - Or the smallest solution if there are multiple ones

Matrix rank

- Suppose we have a set of vectors $\mathbf{v_1}, \ldots, \mathbf{v_n}$
- If we can express $\mathbf{v}_1$ as a linear combination of the other vectors $\mathbf{v}_2 \ldots \mathbf{v}_n$, then $\mathbf{v}_1$ is linearly *dependent* on the other vectors.
  - The direction $\mathbf{v}_1$ can be expressed as a combination of the directions $\mathbf{v}_2 \ldots \mathbf{v}_n$. (E.g. $\mathbf{v}_1 = .7\ \mathbf{v}_2 - .7\ \mathbf{v}_4$)
- If no vector is linearly dependent on the rest of the set, the set is linearly *independent*.
  - Common case: a set of vectors $\mathbf{v_1}, \ldots, \mathbf{v_n}$ is always linearly independent if each vector is perpendicular to every other vector (and non-zero)

Matrix rank

- For a matrix **A**, it holds that:
  - the column rank of the matrix is the maximum number of linear independent column vectors of **A**
  - the row rank of the matrix is the maximum number of linear independent row vectors of **A**

Matrix rank = Column rank of the matrix = Row rank of the matrix

Matrix rank

- An *m*x*m* matrix with rank *m* is said to be *full rank.* Such a matrix is invertible (non-singular) and it maps a *m* x 1 vector to another *m* x 1 vector.
- If the rank of the matrix is smaller than *m,* then the matrix is singular and the inverse does not exist.
- Non-square (*m*x*n, m≠n)* matrices have no inverse, either.

Singular Value Decomposition

SVD

Singular Value Decomposition (SVD)

- Factorizing a matrix means representing it as a product of other matrices
- A typical way to do this is SVD
- SVD can take any matrix **A** and represent it as a product of three matrices, in the form:

$$\mathbf{A=U\Sigma V^T}$$

U, V are rotation matrices and **Σ** is a scaling matrix

$$\underset{U}{\begin{bmatrix} -.40 & .916 \\ .916 & .40 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 5.39 & 0 \\ 0 & 3.154 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.05 & .999 \\ .999 & .05 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 3 & -2 \\ 1 & 5 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- This does not hold only for square matrices but is general:

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

- Each column of **U** and **V** is a unit vector
- **Σ** is a diagonal matrix. The number of its non-zero elements defines the rank of **A**. By defaults, they are sorted from high to low (9.51>0.77 in our example)

Singular Value Decomposition (SVD)

- To better understand SVD, its usage and, mainly, an application in computer vision, first, take a look at the following formulations
- Imagine the following product representation of 2x3 ‘image’ **A**:

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- Look how *each column of U separately, scaled by one of the values in Σ each time, and a row in V^T* contribute to a partial representation of **A**

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

$$+ \underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} -.6 & -.1 & .4 \\ .2 & 0 & -.2 \end{bmatrix}}$$

$$= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Singular Value Decomposition (SVD)

- In other words, we reconstructed **A** as a linear combination of the columns of **U**
- Note the $\mathbf{A}_{\mathbf{partials}}$ and how they are influenced by the scalars in **Σ**
- This is telling us that, the larger the scalar in **Σ**, the larger its influence in the final reconstruction
- So, in practice, ignoring the smallest scalars in **Σ**, is not such a big issue..

Singular Value Decomposition (SVD)

$$\underset{U}{\begin{bmatrix} -.39 & -.92 \\ -.92 & .39 \end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix} 9.51 & 0 & 0 \\ 0 & .77 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} = \underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

This value places much importance on the first column of **U**

This value places less importance on the first column of **U**

$$\underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} 1.6 & 2.1 & 2.6 \\ 3.8 & 5.0 & 6.2 \end{bmatrix}}$$

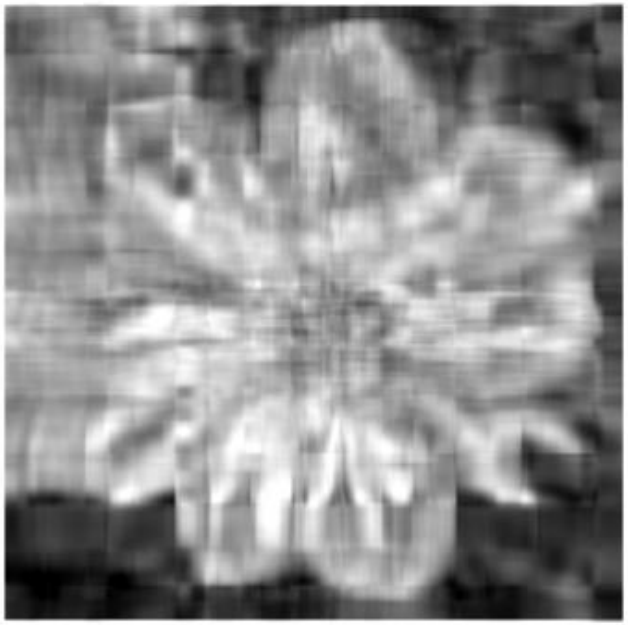
$$\underset{A}{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}$$

$$\begin{bmatrix} \cdot \\ \cdot \end{bmatrix} \underset{U\Sigma}{\begin{bmatrix} -3.67 & -.71 & 0 \\ -8.8 & .30 & 0 \end{bmatrix}} \times \underset{V^T}{\begin{bmatrix} -.42 & -.57 & -.70 \\ .81 & .11 & -.58 \\ .41 & -.82 & .41 \end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix} -.6 & -.1 & .4 \\ .2 & 0 & -.2 \end{bmatrix}}$$

Singular Value Decomposition (SVD)

- The first columns of **U** are the principal components of the data
- They actually account for the major patterns that can be added to reproduce the original data
- The rows in $\mathbf{V}^{\mathbf{T}}$ provide the way the principal components are to be mixed to reconstruct the data (**A**)

Singular Value Decomposition (SVD)



- For this image, using **only the first 10** of 300 principal components produces a recognizable reconstruction
- So, SVD can be used for image compression

Finding SVD
Eigenvector definition

- Suppose we have a square matrix **A**. We can solve for vector x and scalar λ such that $Ax = \lambda x$
- In other words, find vectors where, if we transform them with **A**, the only effect is to scale them with no change in direction.
- These vectors are called eigenvectors (German for "self vector" of the matrix), and the scaling factors λ are called eigenvalues
- An m x m matrix will have $\leq m$ eigenvectors where λ is nonzero

Finding SVD

Finding eigenvectors

- Computers can find an x such that $Ax = \lambda x$ using this iterative algorithm:
  - x=random unit vector
  - while(x hasn't converged)
    - x=Ax
    - normalize x
- x will quickly converge to an eigenvector
- Some simple modifications will let this algorithm find all eigenvectors

Finding SVD

- Eigenvectors are for square matrices, but SVD is for all matrices
- To do svd(A), computers can do this:
  - Take eigenvectors of AA^T (matrix is always square).
    - These eigenvectors are the columns of **U**.
    - Square root of eigen*values* are the singular values (the entries of **Σ**).
  - Take eigenvectors of A^TA (matrix is always square).
    - These eigenvectors are columns of **V** (or rows of $\mathbf{V}^T$)

Finding SVD

- Moral of the story: SVD is fast, even for large matrices
- It's useful for a lot of stuff
- There are also other algorithms to compute SVD or part of the SVD
  - MATLAB's svd() command has options to efficiently compute only what you need, if performance becomes an issue

A detailed geometric explanation of SVD is here:
http://www.ams.org/samplings/feature-column/fcarc-svd

Matrix calculus: the gradient

- Let's consider a function $f: \mathbb{R}^{mxn} \rightarrow \mathbb{R}$ that takes as input a matrix **A** which is *m*x*n* and returns a real value
- The gradient of *f* with respect to A is the *m*x*n* matrix of partial derivatives:

$$\nabla_A f(A) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \frac{\partial f(A)}{\partial A_{12}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\ \frac{\partial f(A)}{\partial A_{21}} & \frac{\partial f(A)}{\partial A_{22}} & \cdots & \frac{\partial f(A)}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{m1}} & \frac{\partial f(A)}{\partial A_{m2}} & \cdots & \frac{\partial f(A)}{\partial A_{mn}} \end{bmatrix}$$

- Important to remember: the gradient is only defined for real-valued scalar functions (e.g. f(x)=Ax, where A is a matrix and x a vector cannot be defined as it outputs a vector)

Matrix calculus: the gradient

- The size of the gradient is the same as that of **A**.
- Usually, **A** is a vector $\mathbf{x} \in \mathbb{R}^n$. In this case:

$$\nabla_x f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

- Properties:

$$\nabla_x (f(x) + g(x)) = \nabla_x f(x) + \nabla_x g(x).$$

$$\text{For } t \in \mathbb{R},\ \nabla_x (t\ f(x)) = t \nabla_x f(x).$$

Matrix calculus: the gradient

- Gradients are an extension of partial derivatives of functions of multiple variables
- Suppose **A** is a fixed *m*x*n* matrix, **x** an *n*-dimensional vector and *f:* $\mathbb{R}^m \rightarrow \mathbb{R}$ a function defined as $f(z)=z^Tz$. It follows that $\nabla_z f(z) = 2z$
- Consider now the expression $\nabla f(Ax)$
- This last expression has two interpretations
  1. Considering that $\nabla_z f(z) = 2z$, it follows that

$$\nabla f(Ax) = 2(Ax) = 2Ax \in \mathbb{R}^m$$

Matrix calculus: the gradient

2. Considering that *f(Ax)* is a function of **x**, then let *g(x)=f(Ax)*

$$\nabla f(Ax) = \nabla_x g(x) \in \mathbb{R}^n$$

- The above two interpretations are different!
- The first one led to an *m*-dimensional result, and the second one gave an *n*-dimensional result
- The solution is to be careful with notations, that is, what we are differentiating with (e.g. *z* or *x* in this example)

The Hessian

- Let's consider a function *f:* $\mathbb{R}^n \rightarrow \mathbb{R}$ that takes as input an n-dimensional vector and returns a real value
- The *Hessian* matrix is the *nxn* matrix of partial derivatives, defines as:

$$\nabla_x^2 f(x) \in \mathbb{R}^{n \times n} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}.$$

Note: the Hessian of a matrix is also defined but is beyond the scopes of this class and a bit difficult to represent in matrix form

The Hessian

- The Hessian is always symmetric: $\frac{\partial^2 f(x)}{\partial x_i \partial x_j} = \frac{\partial^2 f(x)}{\partial x_j \partial x_i}$
- Similar to the gradient, it is only defined for real-valued functions
- One would naturally think that, for vector functions, the Hessian is for gradients what the second derivative is for first derivatives
- In single-variable real-valued functions, the derivative of the derivative can be defined.

The Hessian

- This is not exactly what is happening with the Hessian, though, as we cannot take the gradient of the gradient (the first gradient gives a vector and, then, we cannot take the gradient of a vector)
- What is actually happening is that we can think of the Hessian as a bunch of gradients on each one of the elements of the gradient…

The Hessian

- That is, considering that each element of the gradient is $\nabla_x f(x))_i = \partial f(x)/\partial x_i$
- , we take the gradient with respect to **x** and we get:

$$\nabla_x \frac{\partial f(x)}{\partial x_i} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_i \partial x_1} \\ \frac{\partial^2 f(x)}{\partial x_i \partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_i \partial x_n} \end{bmatrix}$$

- The above expression is just one of the columns of the Hessian

Gradients and Hessians of quadratic and linear functions

Some examples of typical functions follow:

- Consider **x**,**b** *n*-dimensional vectors with **x** being our variable and **b** fixed coefficients. Then, let $f(x) = b^T x$
- It follows that

$$f(x) = \sum_{i=1}^{n} b_i x_i$$

- And $$\frac{\partial f(x)}{\partial x_k} = \frac{\partial}{\partial x_k} \sum_{i=1}^{n} b_i x_i = b_k.$$

Gradients and Hessians of quadratic and linear functions

- It becomes obvious that $\nabla_x b^T x = b$ which is an *n*-dimensional vector
- Look at the analogy with single variable functions where $\partial/(\partial x)\ ax = a$

Gradients and Hessians of quadratic and linear functions

- Let's now consider the quadratic function

$$f(x) = x^T Ax \text{ for } A \in \mathbb{S}^n$$

- It is easy to show that

$$f(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j.$$

- By breaking down the expression above, and considering the partial derivatives with respect to x_k, we get:

Gradients and Hessians of quadratic and linear functions

$$
\begin{aligned}
\frac{\partial f(x)}{\partial x_k} &= \frac{\partial}{\partial x_k} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j \\
&= \frac{\partial}{\partial x_k} \left[\sum_{i \neq k} \sum_{j \neq k} A_{ij} x_i x_j + \sum_{i \neq k} A_{ik} x_i x_k + \sum_{j \neq k} A_{kj} x_k x_j + A_{kk} x_k^2 \right] \\
&= \sum_{i \neq k} A_{ik} x_i + \sum_{j \neq k} A_{kj} x_j + 2 A_{kk} x_k \\
&= \sum_{i=1}^{n} A_{ik} x_i + \sum_{j=1}^{n} A_{kj} x_j = 2 \sum_{i=1}^{n} A_{ki} x_i,
\end{aligned}
$$

- The last equality follows since **A** is symmetric
- Obviously, the k_{th} entry of $\nabla_x f(x)$ is the inner product of the k_{th} row of **A** and **x**
- Therefore, $\nabla_x x^T A x = 2Ax$
- Note the analogy with single-variable functions where $\partial/(\partial x)\ ax^2 = 2ax$

Gradients and Hessians of quadratic and linear functions

- Each entry of the Hessian of $f(x) = x^T Ax$ is:

$$\frac{\partial^2 f(x)}{\partial x_k \partial x_\ell} = \frac{\partial}{\partial x_k}\left[\frac{\partial f(x)}{\partial x_\ell}\right] = \frac{\partial}{\partial x_k}\left[2\sum_{i=1}^{n} A_{\ell i} x_i\right] = 2A_{\ell k} = 2A_{k\ell}.$$

- So, the Hessian is $\nabla_x^2 x^T Ax = 2A,$
- Again, note the analogy with $\partial^2/(\partial x^2)\; ax^2 = 2a$
- As a summary: $\nabla_x b^T x = b$

$$\nabla_x x^T Ax = 2Ax \text{ (if } A \text{ symmetric)}$$

$$\nabla_x^2 x^T Ax = 2A \text{ (if } A \text{ symmetric)}$$

Least Squares using the gradient

- Consider the system of equations **Ax**=**b**, where **A**$\in\mathbb{R}^{m\times n}$, **x**$\in\mathbb{R}^{n}$ and **b**$\in\mathbb{R}^{m}$. **A** and **b** are known (consider also that A is full rank)
- We want to solve for **x,** when no exact solution exists and m>n
- We have that:

$$\begin{aligned} \|Ax-b\|_2^2 &= (Ax-b)^T(Ax-b) \\ &= x^TA^TAx-2b^TAx+b^Tb \end{aligned}$$

Least Squares using the gradient

- We want to take the gradient with respect to **x** and set it to zero, in order to find critical points, where there is a good chance to have a minimum of the above expression

$$\begin{aligned}\nabla_x(x^T A^T Ax - 2b^T Ax + b^T b) &= \nabla_x x^T A^T Ax - \nabla_x 2b^T Ax + \nabla_x b^T b \\ &= 2A^T Ax - 2A^T b\end{aligned}$$

Setting this to zero gives: $x = (A^T A)^{-1} A^T b$

MATLAB

Outline

- **Introduction to MATLAB**
- Image Processing toolbox with MATLAB
  - ☐ Simple functions, just to get to know basic ideas

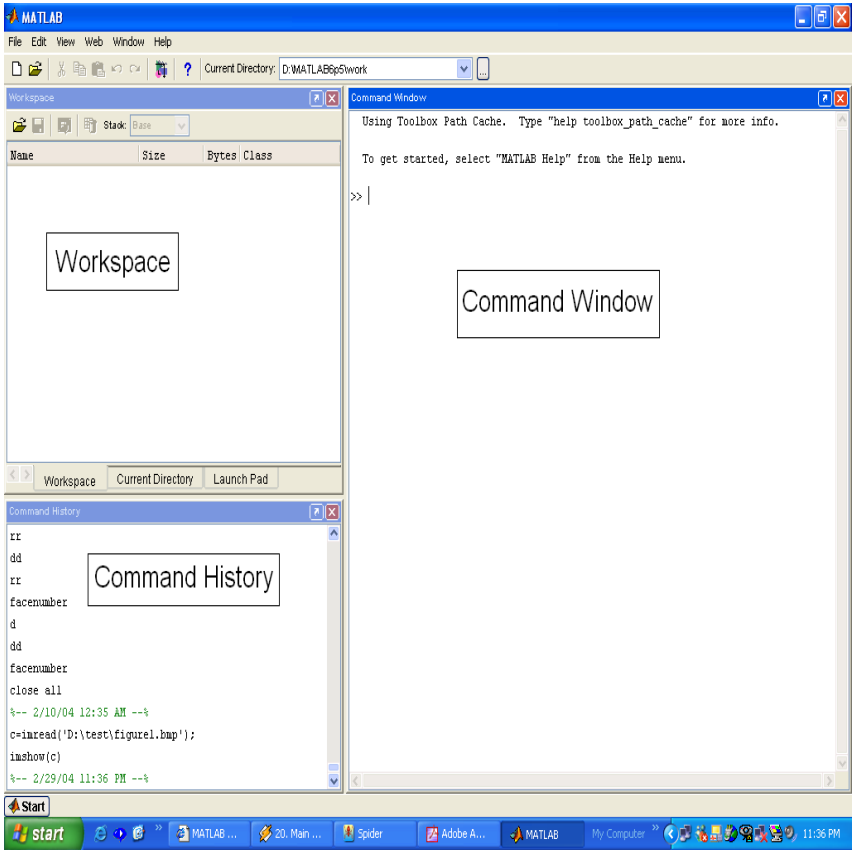
What is MATLAB?

- MATLAB = MATrix LABoratory

- "MATLAB is a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++ and Fortran." (www.mathworks.com)

- MATLAB is an interactive, interpreted language that is designed for fast numerical matrix calculations
- Easy to use (really easy to get to know if you want to do basic operations)

The MATLAB Environment



- Basic elements of the MATLAB environment

Workspace

> Displays all the defined, active, loaded variables

Command Window

> To execute commands in the MATLAB environment. A console

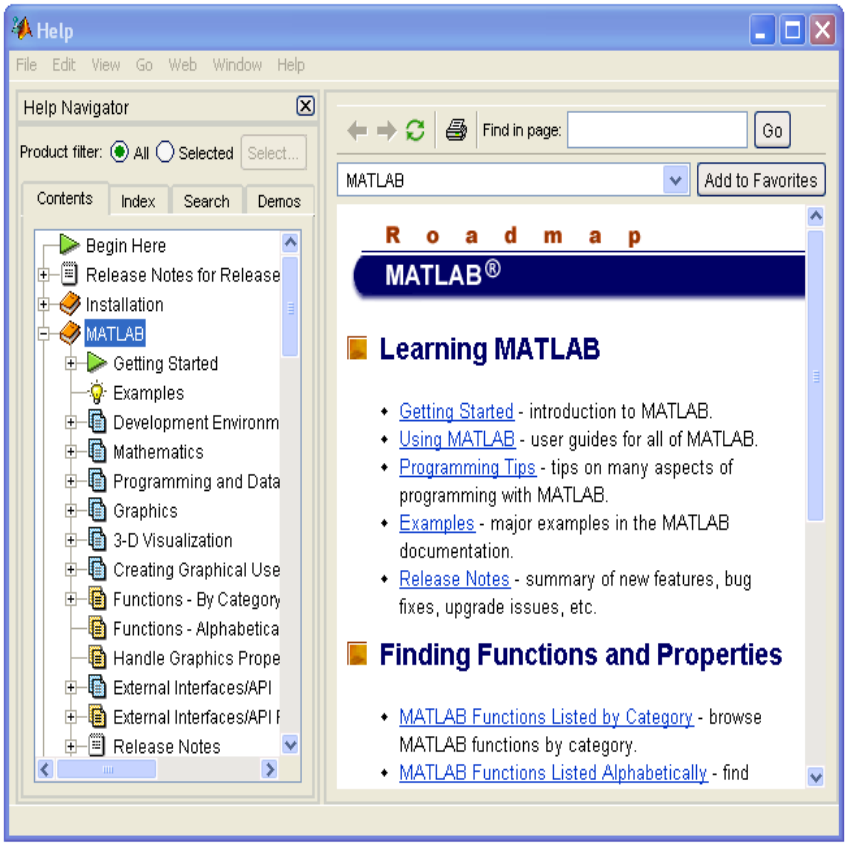
Command History

> Displays record of the commands used

File Editor Window

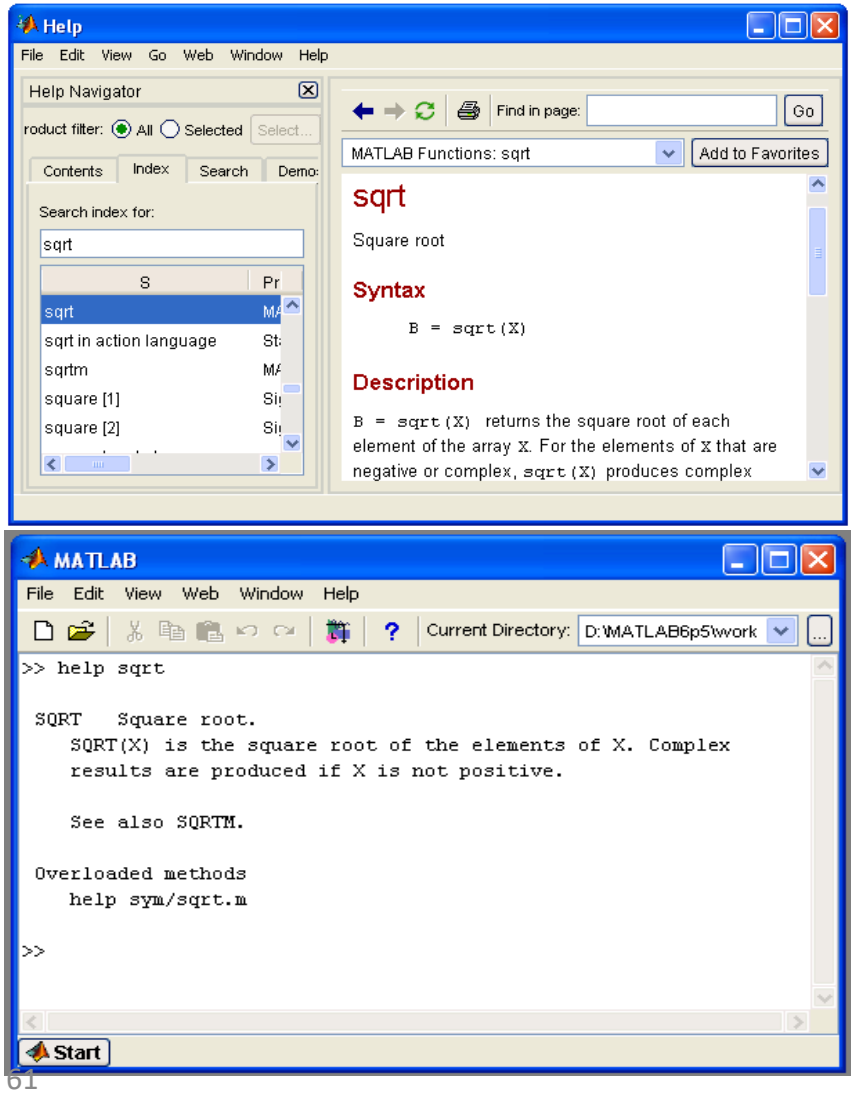
> Define your functions or m-files

MATLAB Help



- Help not only contains the theoretical background, but also shows demos for implementation
- Very good online resources on Mathworks.com
- .. Or just type help <function> in the command window to see what a MATLAB function can do and how

MATLAB Help (cont.)



- Any command description can be found by typing the command in the search field
- As shown above, the command to take square root (`sqrt`) is searched
- We can also utilize MATLAB Help from the command window as shown

Matrices in MATLAB

- Matlab works with Matrices.
- How to build a matrix?
  - `A=[1 2 3; 4 5 6; 7 8 9];`
  - Creates matrix A of size 3 x 3
- Special matrices:
  - *`zeros(n,m), ones(n,m), eye(n,m), rand(), randn()`*

Basic Operations on Matrices

- All operators in MATLAB are defined on matrices: `+, -, *, /, ^, sqrt, sin, cos,` etc.
- Element-wise operators defined with a preceding dot: `.*, ./, .^`
- `size(A)` – size vector
- `sum(A,1)` – columns sums vector
- `sum(A,2)` – row sums vector
- `sum(sum(A))` – sum of all the elements

Variable Name in Matlab

- Variable names must
  - begin with a letter
  - not contain blank spaces or other types of punctuation
- Also
  - they contain any combination of letters, digits, and underscores
  - they are case-sensitive
  - should not use Matlab keywords
- Pre-defined names
  - `pi` & embedded function names

Logical Operators

- `==, <, >,` (not equal) ~=, (not) ~
- `find('condition')` – Returns indexes of A's elements that satisfy the condition

Logical Operators (cont.)

- Example:

```
>>A=[7 3 5; 6 2 1]
```

```
Id=find(A<4)
   A=
      7 3 5
      6 2 1
   Id= 3 4 6
```

```
[Idx, Idy]=find(A<4)
   A=
      7 3 5
      6 2 1
   Idx = 1 2 2
   Idy = 2 2 3
```

Flow Control

- Five flow control constructs:
  - `if` statement
  - `switch` statement
  - `for` loop
  - `while` loop

if

- IF statement condition
  - The general form of the IF statement is

```
if (i==1/2)
   imshow(image*1/2)
elseif (i==2)
   imshow(image*2)
else
   imshow(image)
end
```

switch

- SWITCH – Switch among several cases based on expression
- The general form of SWITCH statement is:

```
switch i
    case 1/2,
    imshow(image*1/2)
    case 2,
    imshow(image*2)
    otherwise
    imshow(image)
end
```

switch (cont.)

- Note:
  - Only the statements between the matching `CASE` and the next `CASE`, `OTHERWISE`, or `END` are executed
  - Unlike C, the `SWITCH` statement does not fall through (so `BREAKS` are unnecessary)

for

- FOR repeats statements a specific number of times
- The general form of a FOR statement is:

```
FOR variable=expr
    statements
END
```

```
rows_image=size(image,1)
for i=1:rows_image
  image(i,1)=i;
end
```

while

- WHILE repeats statements an indefinite number of times
- The general form of a WHILE statement is:

```
WHILE expression
   statements
END
```

```
i=1;
while(i<1000)
imshow(C1/i);
i=i+1;
end
```

Scripts and Functions

- There are two kinds of M-files:
  - Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace and can replace command window operations.
  - Functions, which can accept input arguments and return output arguments. Internal variables are local to the function.

Functions in MATLAB (cont.)

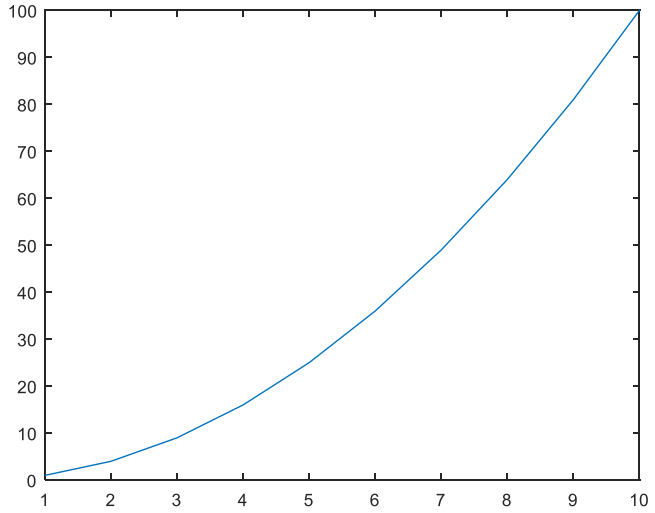
- Example:
  - A file called STAT.M:

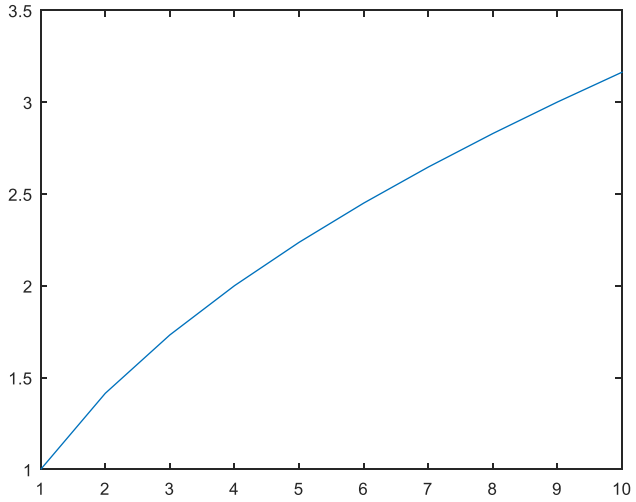
```
function [new_image, m,
 n]=darken_image(image)
%this function darkens an image and
 returns its dimensions.
n=size(image,1); m=size(image,2);
new_image=image-30;
```

  - Defines a new function called darken_image that calculates image dimensions (n,m) and a new, darker image

Visualization, Graphics, Plots

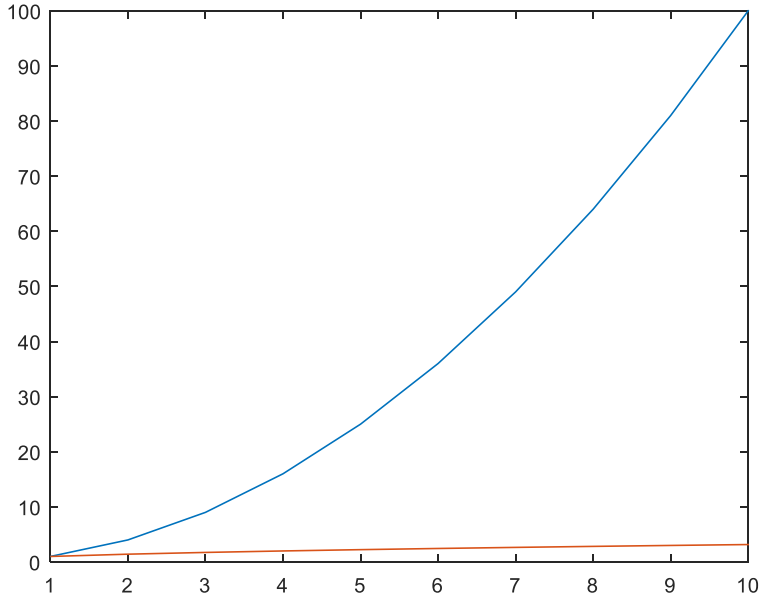
```
close all;
x=1:10; %vector [1 2 … 10]
y=x.^2;
z=sqrt(x);
plot(x,y)
figure % creates a new
%figure, keeping the old
%one alive
plot(x,z)
```





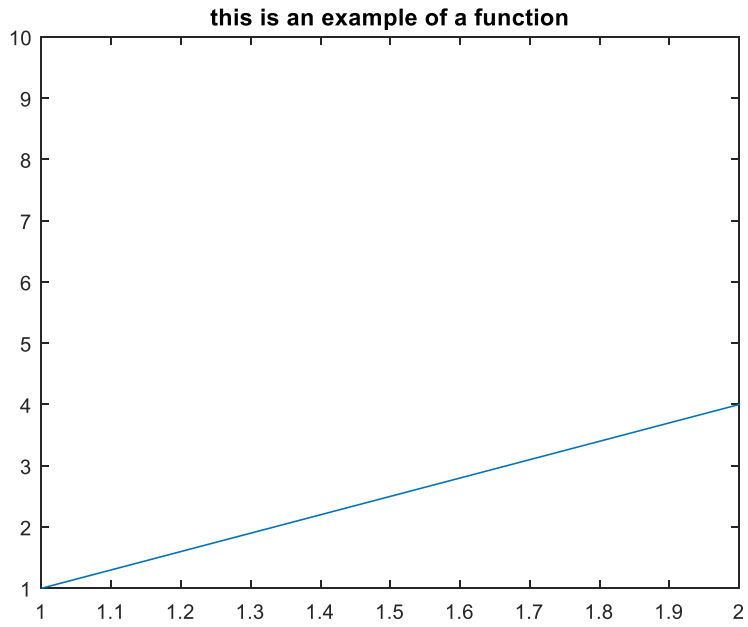
Visualization, Graphics, Plots

```
close all;
x=1:10; %vector [1 2 … 10]
y=x.^2;
z=sqrt(x);
plot(x,y)
hold on % refreshing (hold
%off renews)
plot(x,z)
```



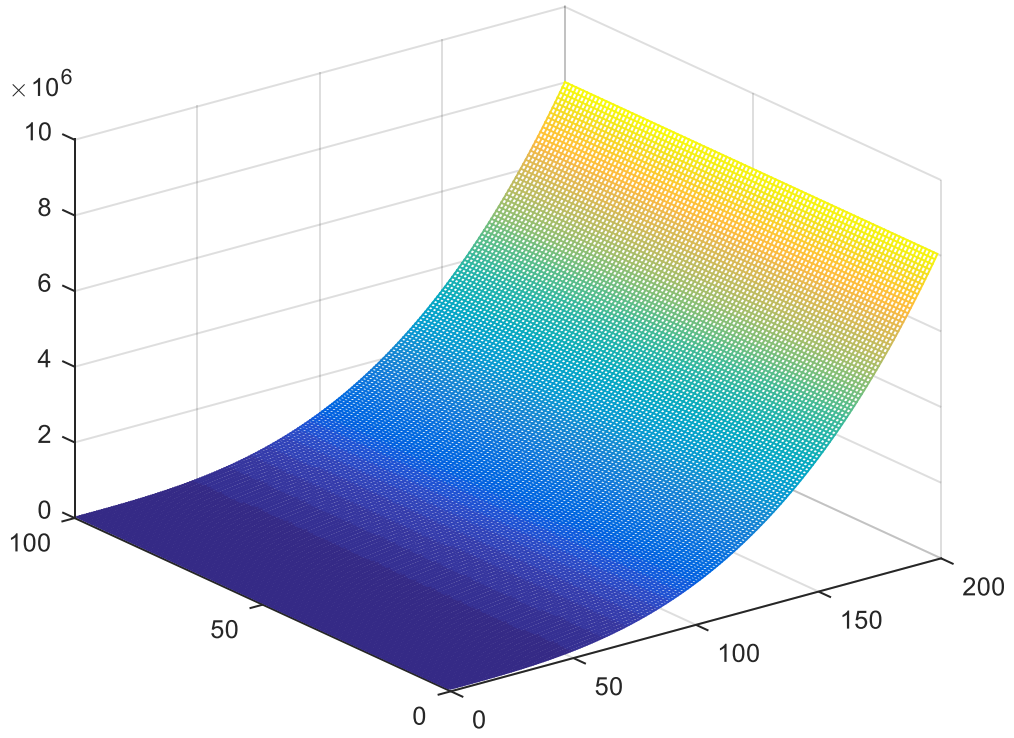
Visualization, Graphics, Plots

```
close all;
x=1:10; %vector [1 2 … 10]
y=x.^2;
z=sqrt(x);
plot(x,y)
axis([1 2 1 10])
title('this is an example of a
function')
```



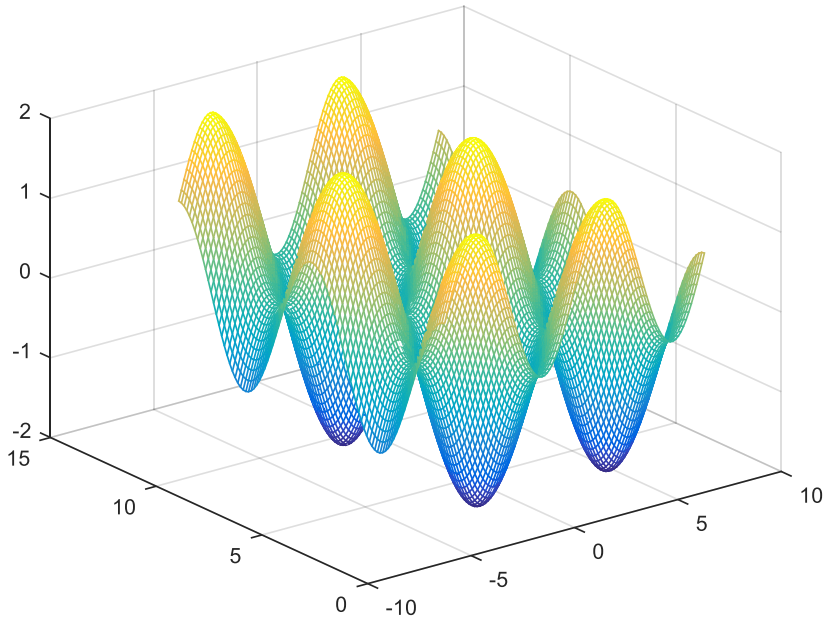
Visualization, Graphics, Plots

```
i=1:100;
j=1:100;
[i,j]=meshgrid(i,j);
m=i.^2+j.^3;
mesh(i,j,m);
```



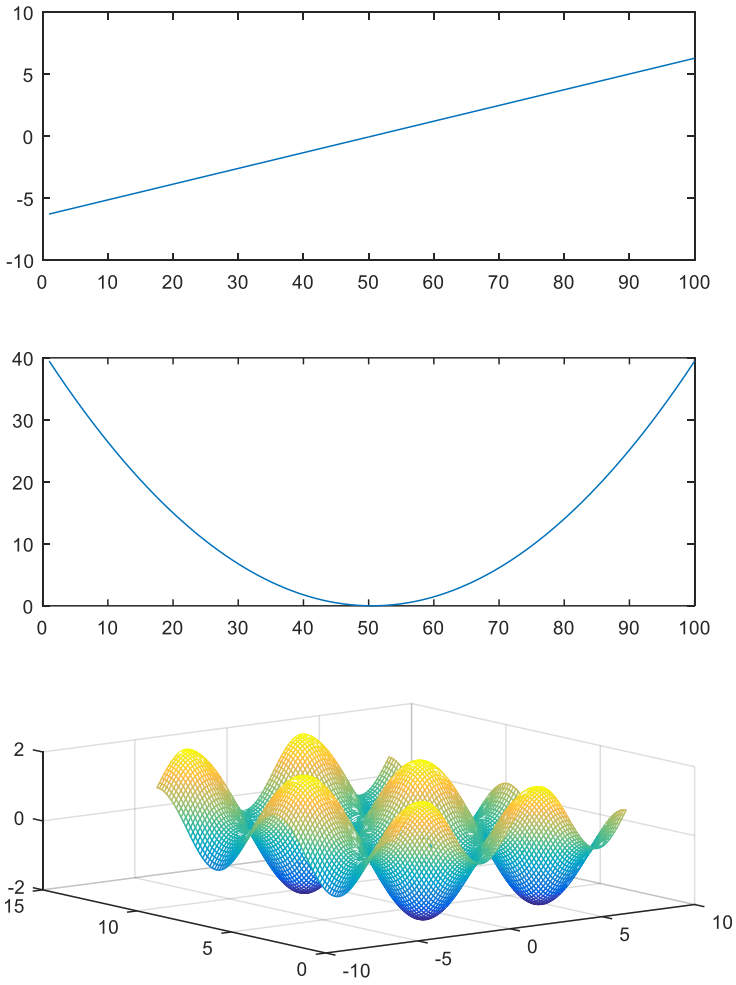
Visualization, Graphics, Plots

```
x = linspace(-2*pi,2*pi);
y = linspace(0,4*pi);
[X,Y] = meshgrid(x,y);
Z = sin(X)+cos(Y);
figure
mesh(X,Y,Z)
```



Visualization, Graphics, Plots

```
x = linspace(-
2*pi,2*pi,100);
y = linspace(0,4*pi,100);
[X,Y] = meshgrid(x,y);
Z = sin(X)+cos(Y);
subplot(3,1,1)
plot(x)
subplot(3,1,2)
plot(x.^2)
subplot(3,1,3)
mesh(X,Y,Z)
```



More about the Workspace

- *who, whos* – current variables in the workspace
- `save mysession`

  % creates mysession.mat with all variables
- `save mysession a b`

  % save only variables a and b *load* – load variables from *.mat file
- `load mysession`

  % load session
- *clear a b* – clear workspace variables
- *clear all* – clear all workspace variable
- *close all* – close all active windows (figures, plots)
- *clc* – clean command window – variables stay but are hidden

Outline

- Introduction to MATLAB
  - ☐ **Simple operations with images**

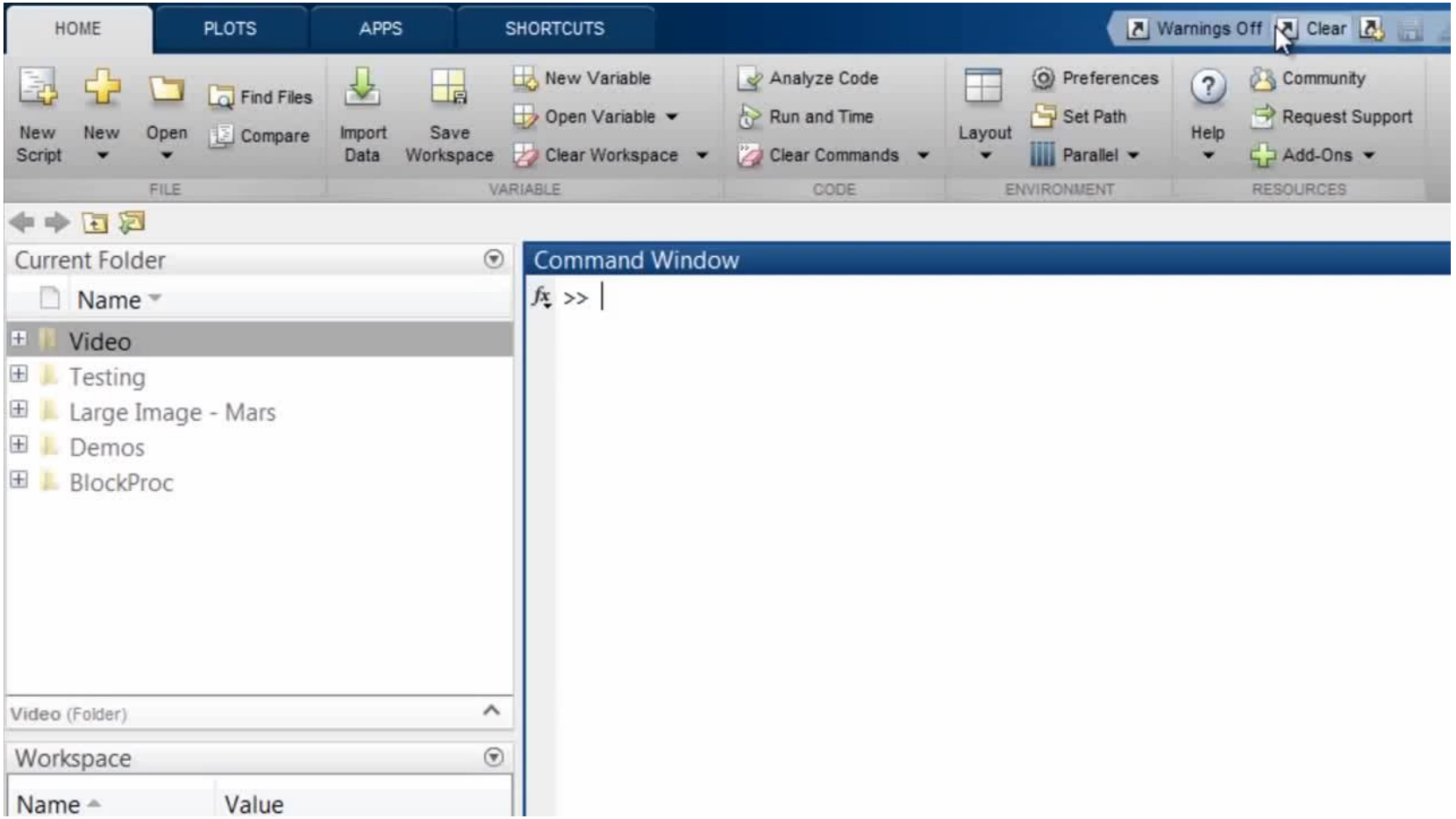
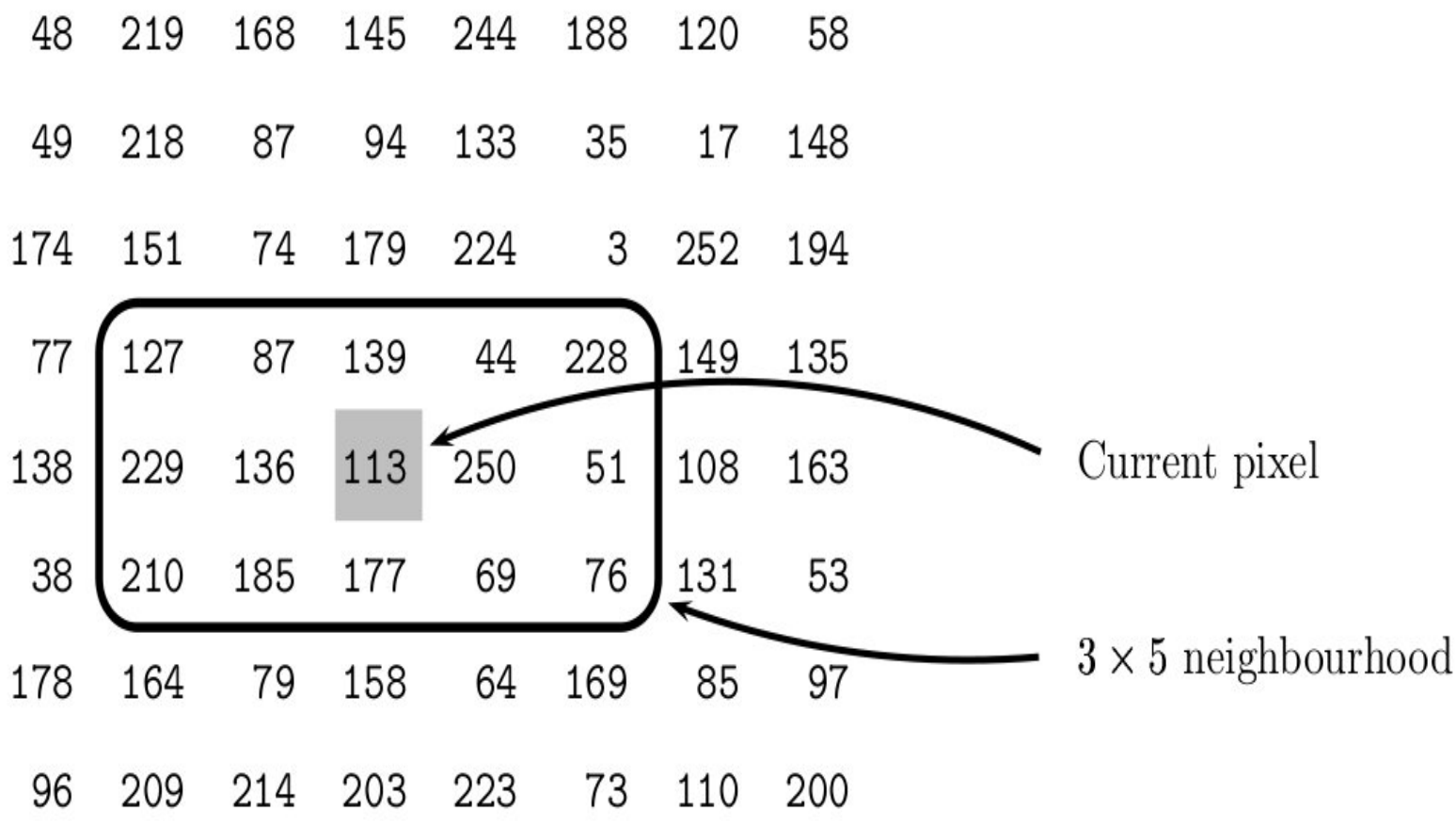

HOME
PLOTS
APPS
SHORTCUTS
Warnings Off
Clear
New Script
New
Open
Find Files
Compare
Import Data
Save Workspace
New Variable
Open Variable
Clear Workspace
Analyze Code
Run and Time
Clear Commands
Layout
Preferences
Set Path
Parallel
Help
Community
Request Support
Add-Ons
FILE
VARIABLE
CODE
ENVIRONMENT
RESOURCES
Current Folder
Name
Video
Testing
Large Image - Mars
Demos
BlockProc
Video (Folder)
Workspace
Name
Value
Command Window
>>

Image and digital images

A **digital image has discrete values and can be seen as a multi-dimensional matrix of size M,N,x**.

- **M** is the number of rows, **N** is the number of columns and **x** is the number of color channels (usually 1 or 3)
- A digital image can be considered as a large array of discrete dots, each of which has a brightness associated with it. These dots are called picture elements, or more simply **pixels**.
  - Pixels take **integer** values, usually from 0 (black/dark) to 255 (white/bright).

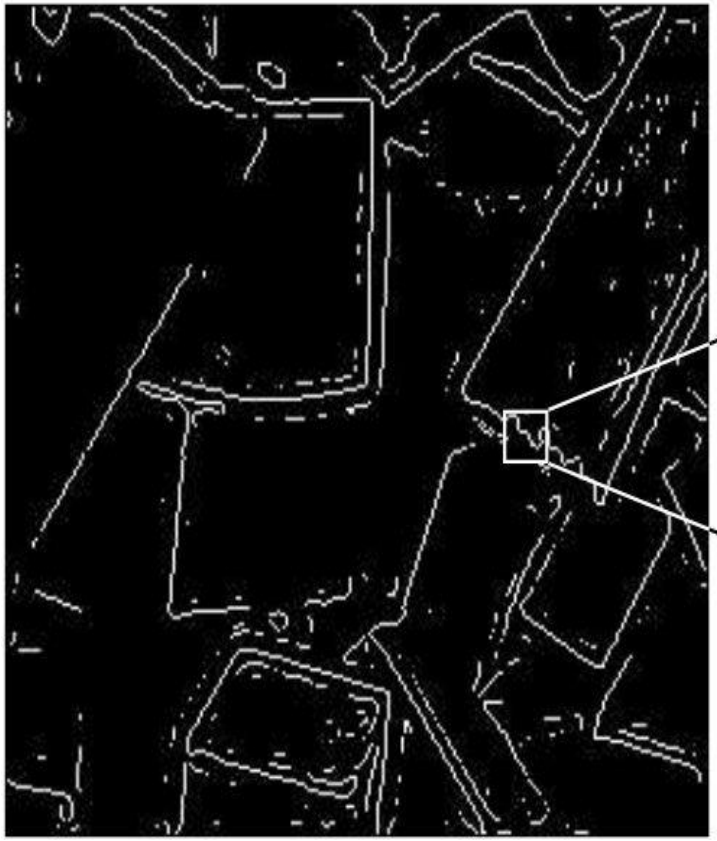
Pixel neighbourhood



Types of digital images

- **Binary:** Each pixel is just **black** or **white**. Since there are only two possible values for each pixel (0,1), we only need **one bit** per pixel.
- **Grayscale:** Each pixel is a shade of gray, normally from **0** (black) to **255** (white). This range means that each pixel can be represented by **eight bits**, or exactly **one byte**. Other greyscale ranges are used, but generally they are a power of **2**.
- **True Color**, or **RGB**: Each pixel has a particular color; that color is described by the amount of **red**, **green** and **blue** in it. If each of these components has a range 0–255, this gives a total of $\mathbf{256^3}$ different possible colors. Such an image is a "stack" of **three matrices**; representing the **red**, **green** and **blue** values for each pixel. This means that for every pixel there correspond 3 values.

Binary Image



1	1	0	0	0	0
0	0	1	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	0	0	0	1

Greyscale Image

230	229	232	234	235	232	148
237	236	236	234	233	234	152
255	255	255	251	230	236	161
99	90	67	37	94	247	130
222	152	255	129	129	246	132
154	199	255	150	189	241	147
216	132	162	163	170	239	122

Color Image

49	55	56	57	52	53
58	60	60	58	55	57
58	58	54	53	55	56
83	78	72	69	68	69
88	91	91	84	83	82
69	76	83	78	76	75
61	69	73	78	76	76

Red

64	76	82	79	78	78
93	93	91	91	86	86
88	82	88	90	88	89
125	119	113	108	111	110
137	136	132	128	126	120
105	108	114	114	118	113
96	103	112	108	111	107

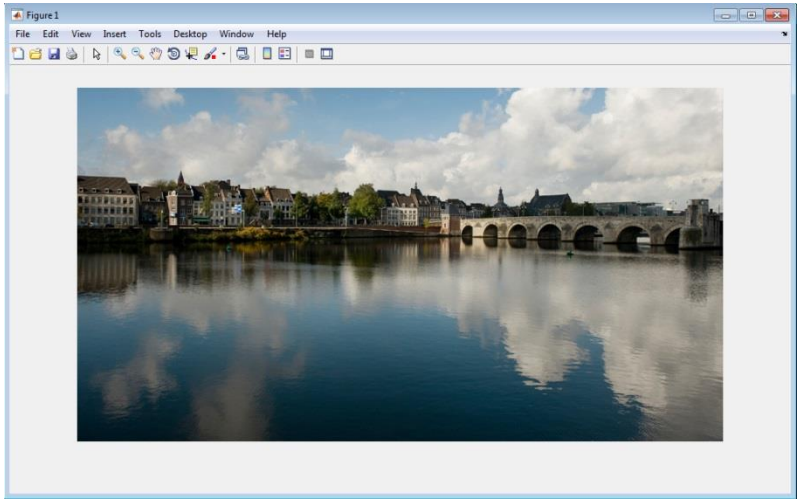
Green

66	80	77	80	87	77
81	93	96	99	86	85
83	83	91	94	92	88
135	128	126	112	107	106
141	129	129	117	115	101
95	99	109	108	112	109
84	93	107	101	105	102

Blue

General commands

- `cd('C:\....\Courses\Data and Image Analysis\m-files')`
- `img=imread('maas.jpg')`: Read an image and store values in img
- `figure`: creates a figure on the screen.
- `imshow(img)`: which displays the matrix img as an image.
- `impixel(img,i,j)`: the command returns the value of the pixel (i,j) (e.g. impixel(img,1,1)=[96 141 172]) - need image proc. Toolbox
- Can do the same through `img(i,j)` – no need for image. Proc. Toolbox / image as a matrix
- `imwrite(img, 'output.bmp', 'bmp');`



Images in MATLAB

- MATLAB can import/export several image formats:
  - BMP (Microsoft Windows Bitmap)
  - GIF (Graphics Interchange Files)
  - HDF (Hierarchical Data Format)
  - JPEG (Joint Photographic Experts Group)
  - PCX (Paintbrush)
  - PNG (Portable Network Graphics)
  - TIFF (Tagged Image File Format)
  - XWD (X Window Dump)
  - raw-data and other types of image data
- Data types in MATLAB
  - Double (64-bit double-precision floating point)
  - Single (32-bit single-precision floating point)
  - Int32 (32-bit signed integer)
  - Int16 (16-bit signed integer)
  - Int8 (8-bit signed integer)
  - Uint32 (32-bit unsigned integer)
  - Uint16 (16-bit unsigned integer)
  - Uint8 (8-bit unsigned integer)

Data types

Data type	Description	Range
`int8`	8-bit integer	−128 — 127
`uint8`	8-bit unsigned integer	0 — 255
`int16`	16-bit integer	−32768 — 32767
`uint16`	16-bit unsigned integer	0 — 65535
`double`	Double precision real number	Machine specific

Switch between data types
as follows:
B=double(A) and
A=uint8(B)

Image Information

`imageinfo('maas.jpg')` – **(image proc.toolbox needed)**

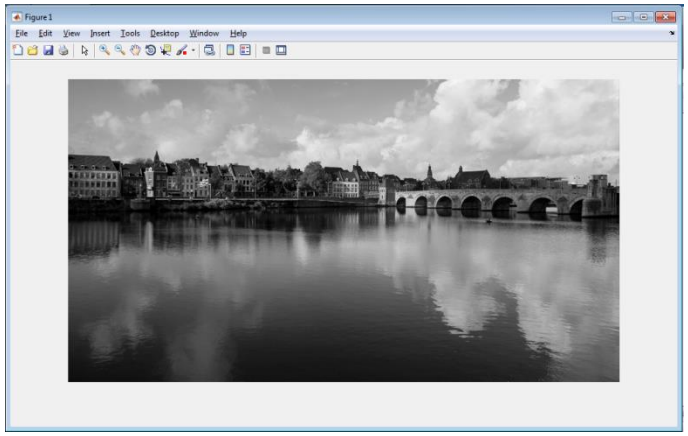
Image Info (maas.jpg)

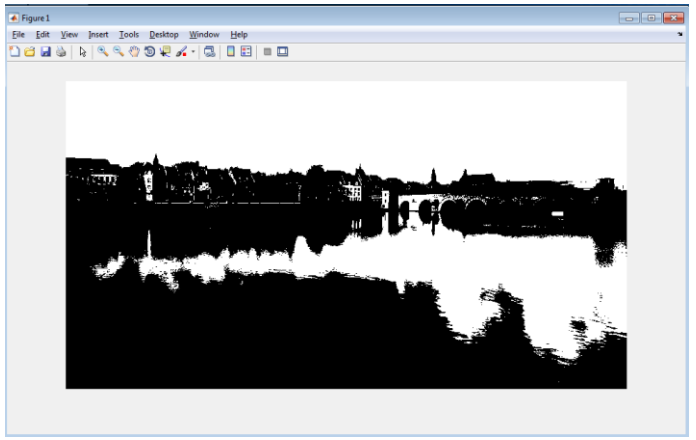
Metadata (maas.jpg)

Attribute	
Filename	C:\Users\Stelios.Asteriadis\Google Drive\Courses\Data and Image Analysis\m-files\maas.jpg
FileModDate	16-Jul-2015 13:51:45
FileSize	197315
Format	jpg
FormatVersion	''
Width	830
Height	455
BitDepth	24
ColorType	truecolor
FormatSignature	''
NumberOfSamples	3
CodingMethod	Huffman
CodingProcess	Sequential
Comment	{}
Orientation	1
XResolution	72
YResolution	72
ResolutionUnit	Inch
Software	Adobe Photoshop CS6 (Windows)
DateTime	2013:05:02 10:09:39
DigitalCamera	[1x1 struct]
ExifThumbnail	[1x1 struct]

Converting between different data type

- `rgb2gray` - RGB image to grayscale
- `im2bw` - image to binary – image proc. toolbox needed



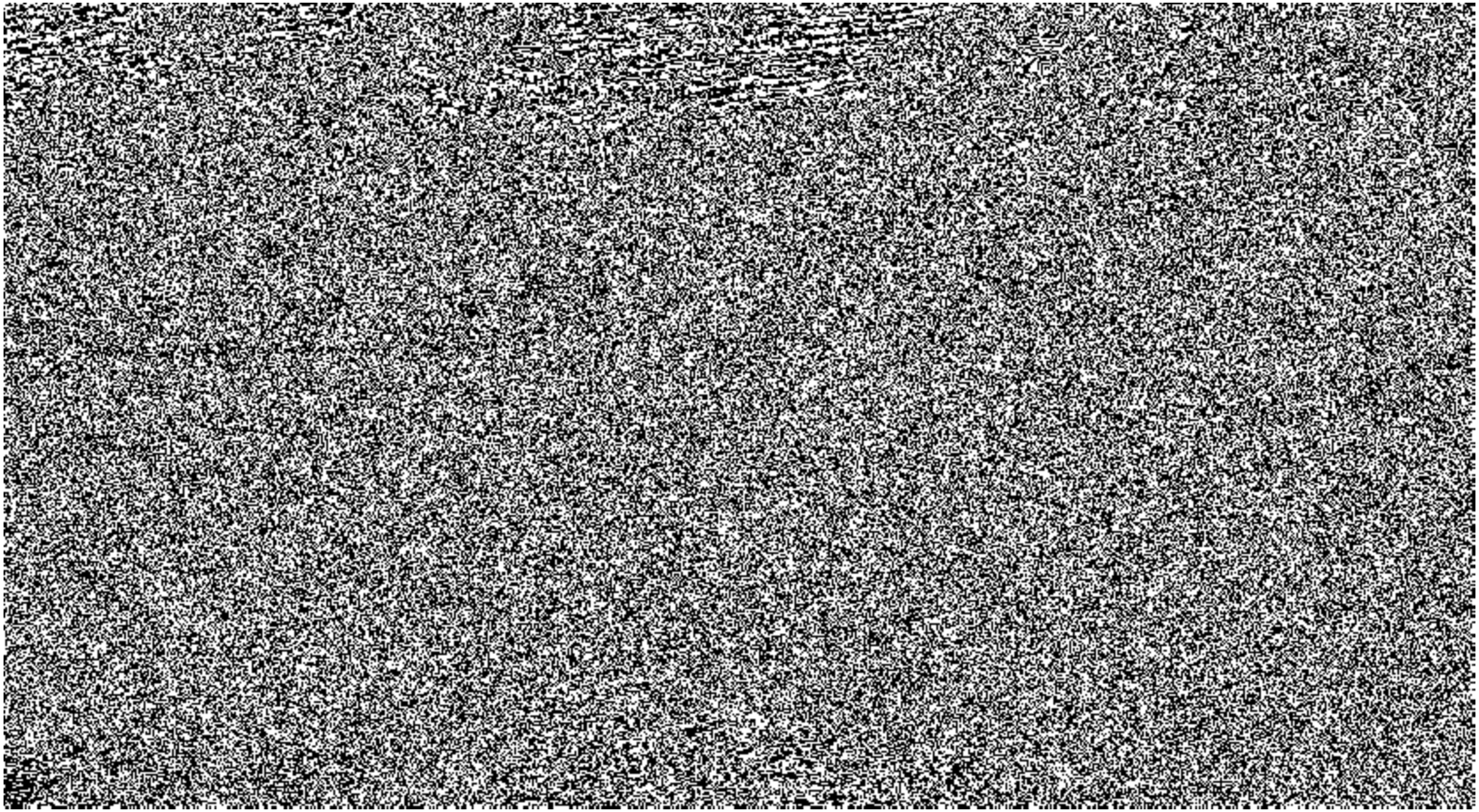


Bit planes

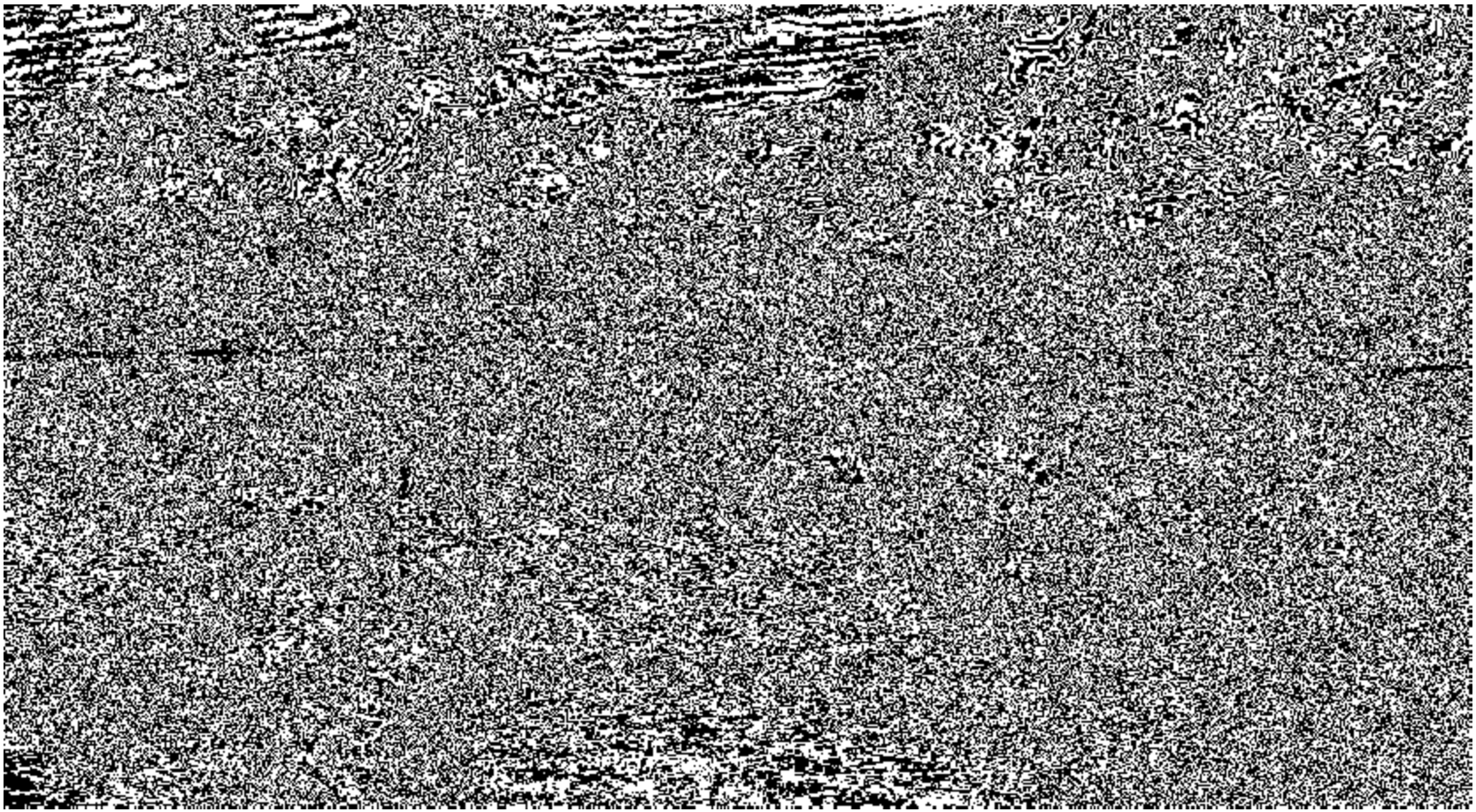
- The (gray) value of each pixel, in an 8-bit image, can be considered as an 8-bit binary word
- Each grayscale image can be decomposed into 8 different levels of information, called **bit-planes**
- The **0th bit plane** consists of the **last bit** of each grey value.
- Since this bit has the least effect (**least significant bit plane**).
- The **7th bit plane** consists of the first bit in each value (**most significant bit plane)**.

Initial Image

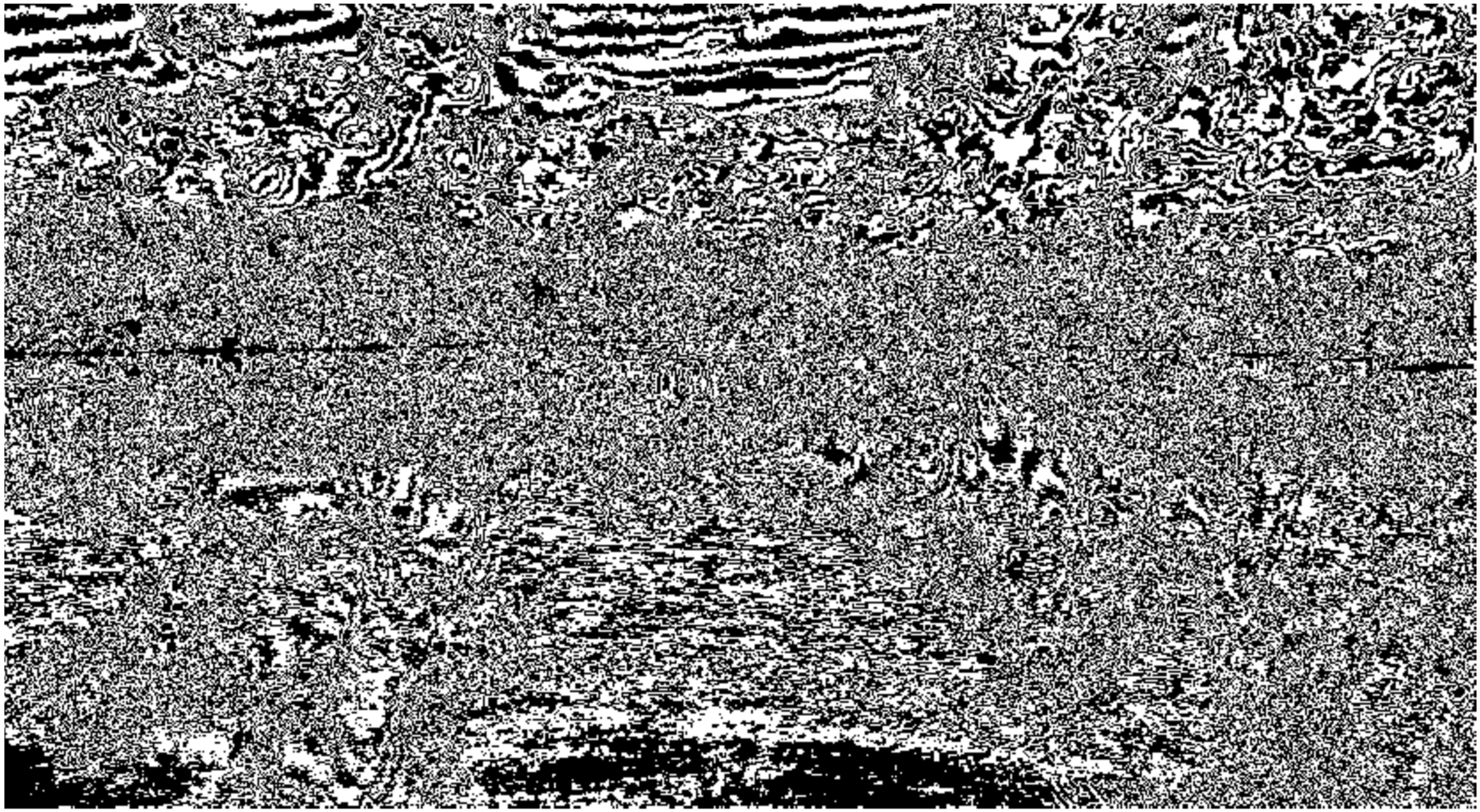
0^{th} bit plane



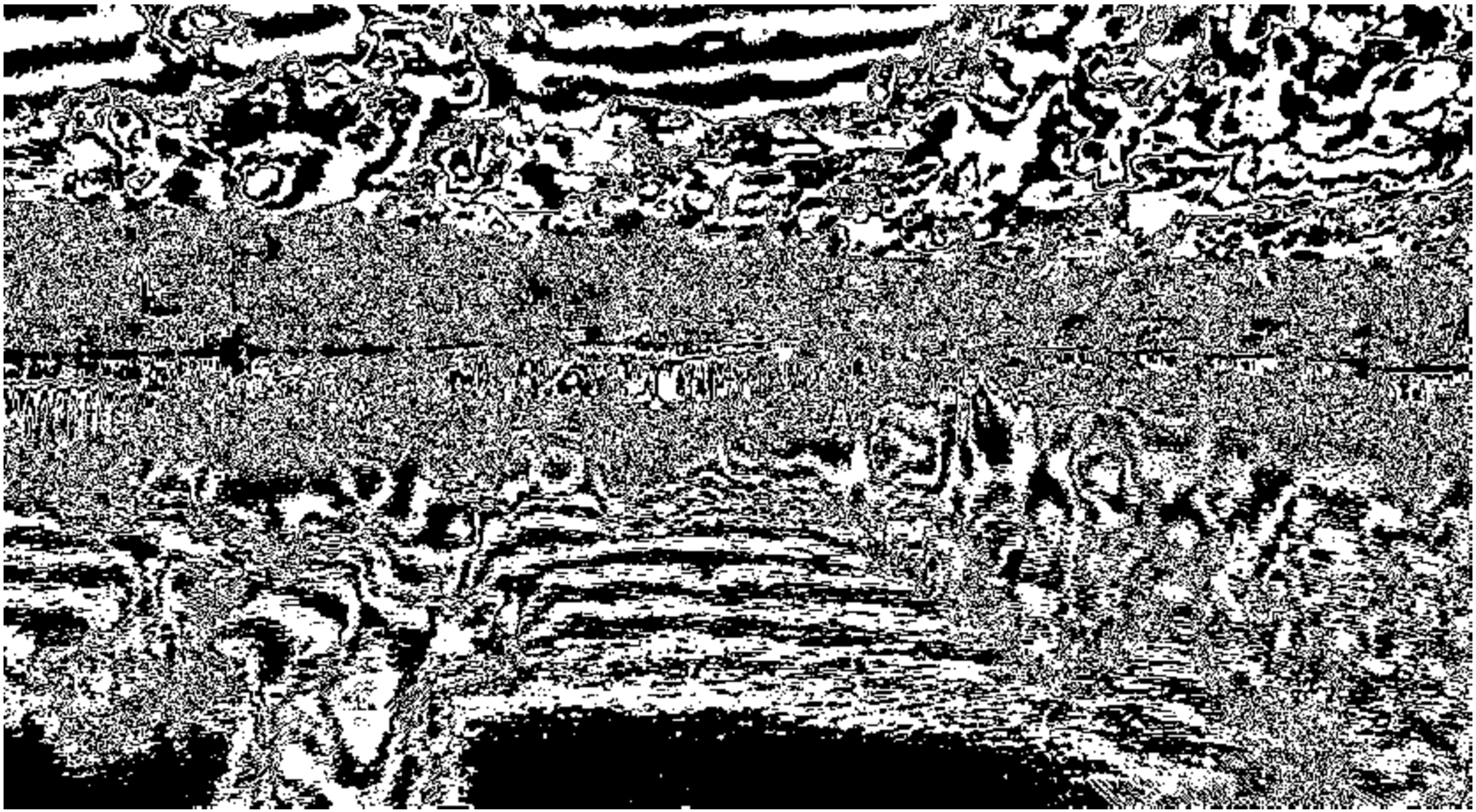
1st bit plane



2nd bit plane

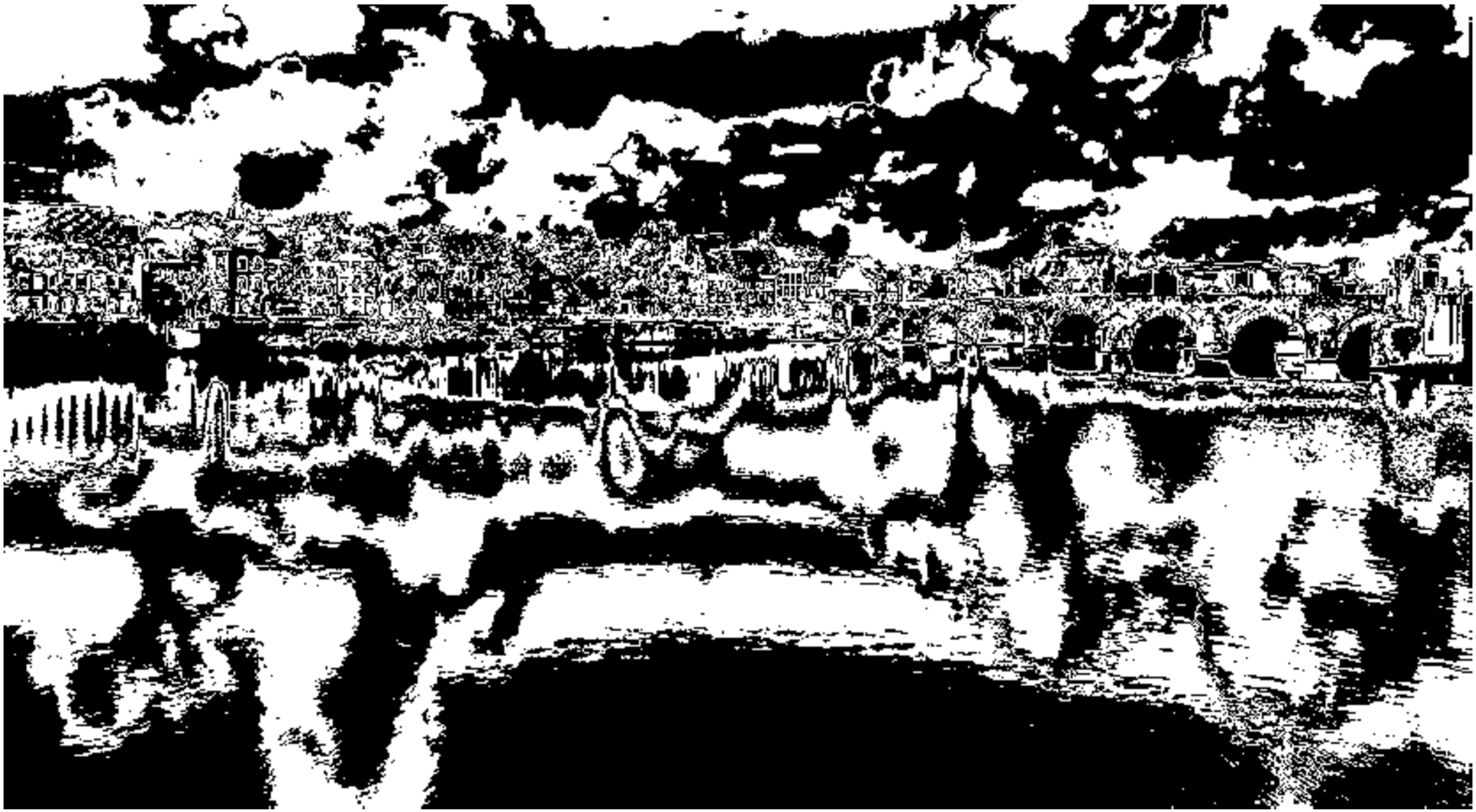


3rd bit plane



4^{th} bit plane

5^{th} bit plane



6th bit plane

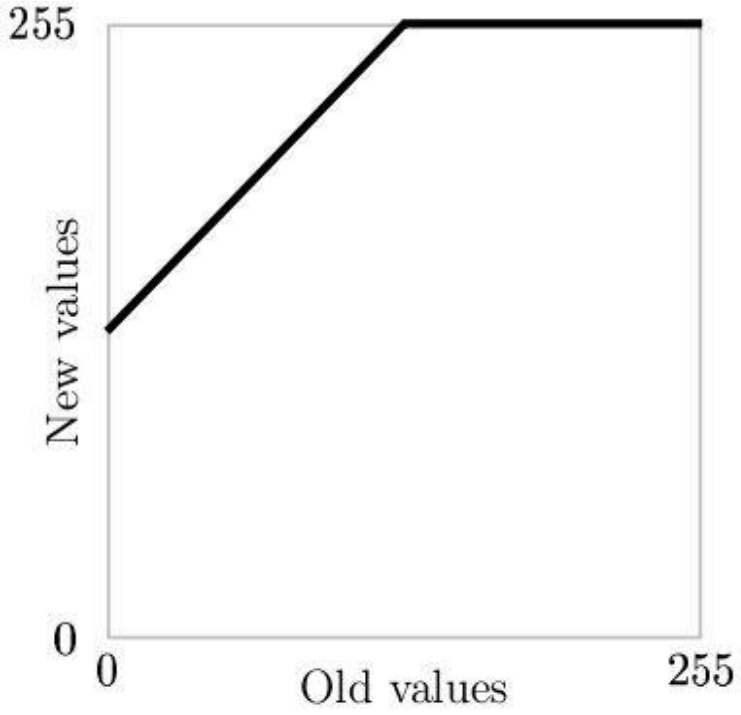
7^{th} bit plane

Arithmetic operations on pixels

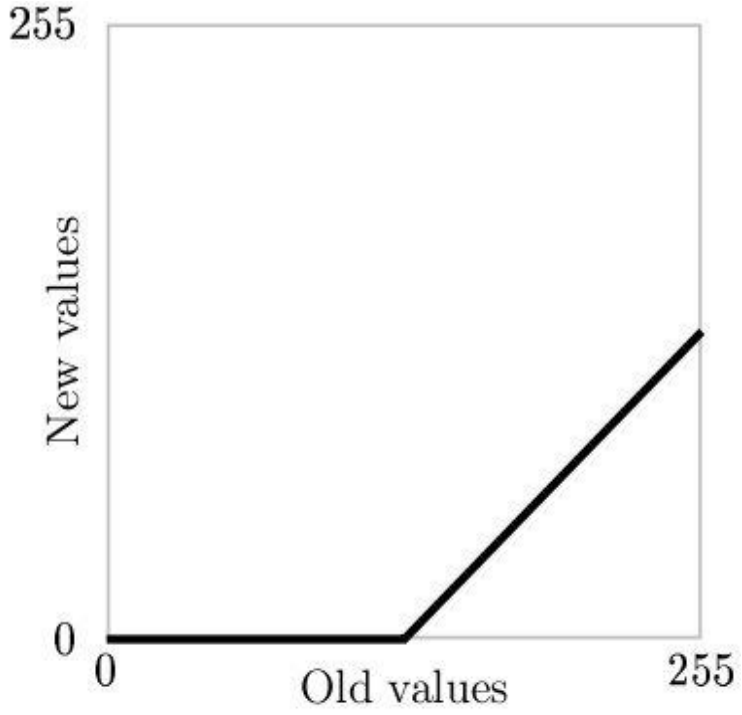
Functions y=f(x) can be applied on each pixel value (gray, R, G, B) *x*

- **Addition/subtraction:** y = x±C (`imadd`, `imsubtract`)
- **Multiplication**: y = C·x (`immultiply`, `imdivide`)
- **Complement**: For a grayscale image, this is its photographic negative.

Arithmetic operations on pixels

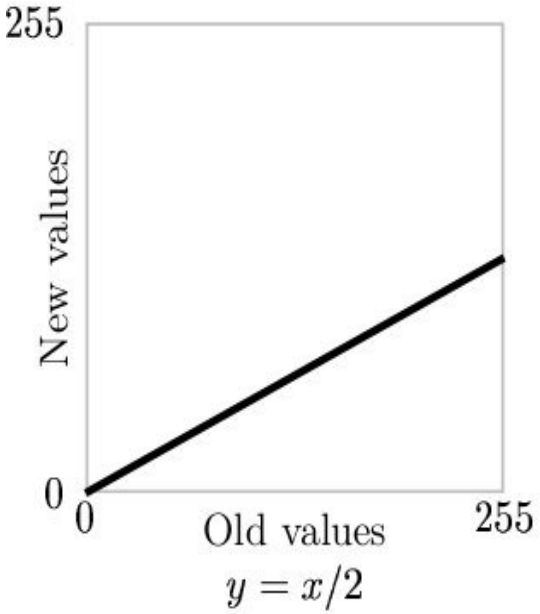


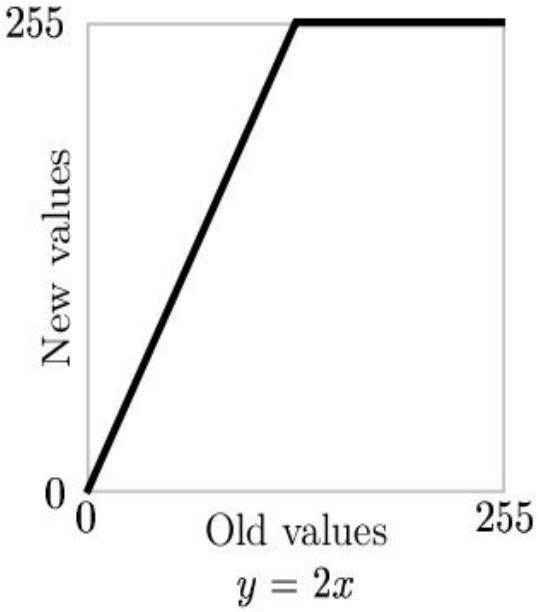
Adding 128 to each pixel

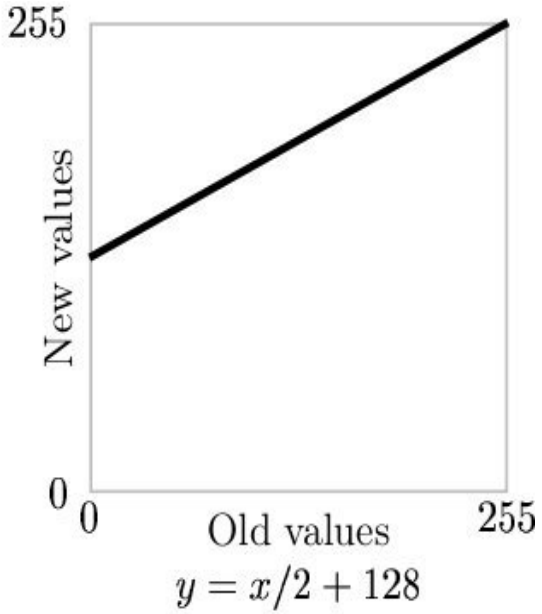


Subtracting 128 from each pixel

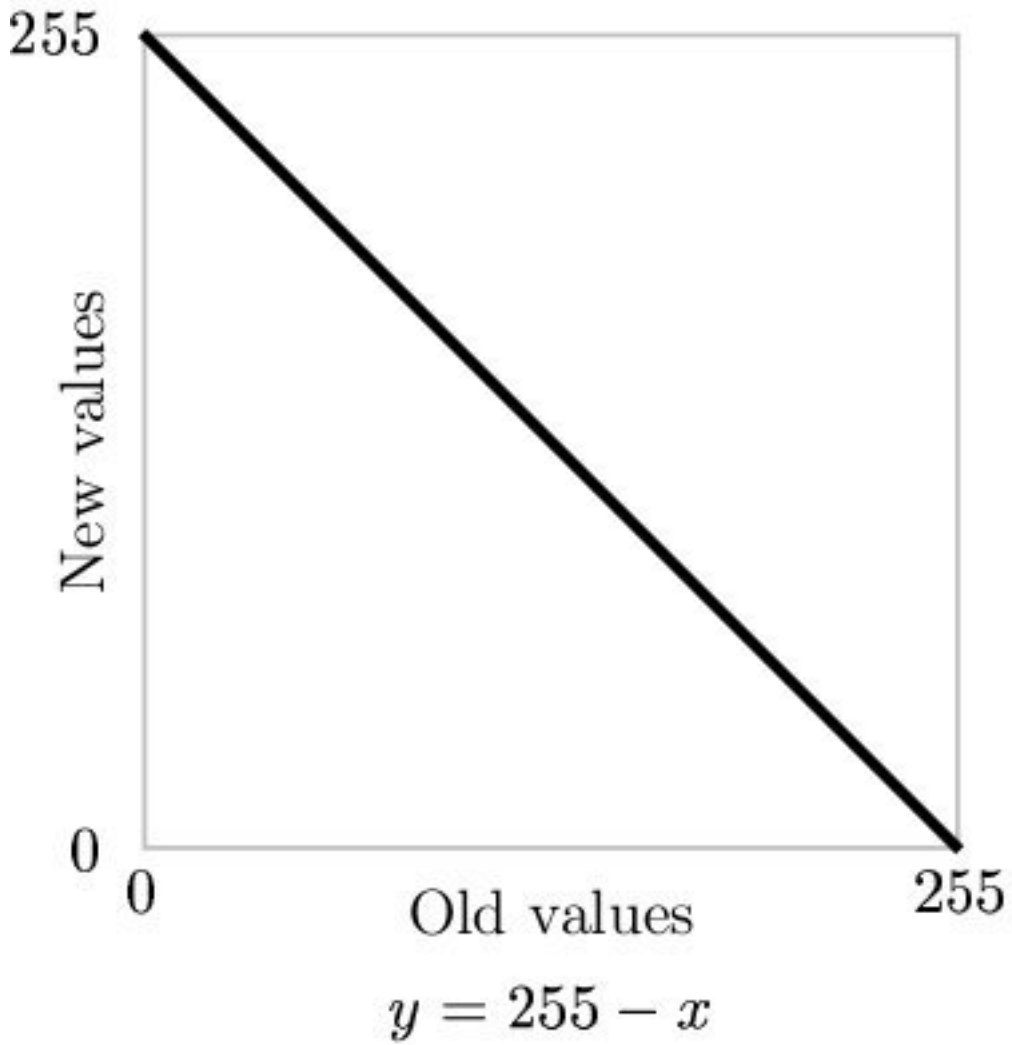
Arithmetic operations on pixels







Complement



Addition

```
A=imread('maas.jpg');
A=rgb2gray(A);
% addition
B=imadd(A,50);
subplot(2,1,1)
imshow(A);
subplot(2,1,2)
imshow(B);
```

Subtraction

```
A=imread('maas.jpg');
A=rgb2gray(A);
% subtraction
B=imsubtract(A,50);
subplot(2,1,1)
imshow(A);
subplot(2,1,2)
imshow(B);
```

Multiplication

```
A=imread('maas.jpg');
A=rgb2gray(A);
% multiplication
B=immultiply(A,2);
subplot(2,1,1)
imshow(A);
subplot(2,1,2)
imshow(B);
```

Division

```
A=imread('maas.jpg');
A=rgb2gray(A);
% division
B=imdivide(A,2);
subplot(2,1,1)
imshow(A);
subplot(2,1,2)
imshow(B);
```

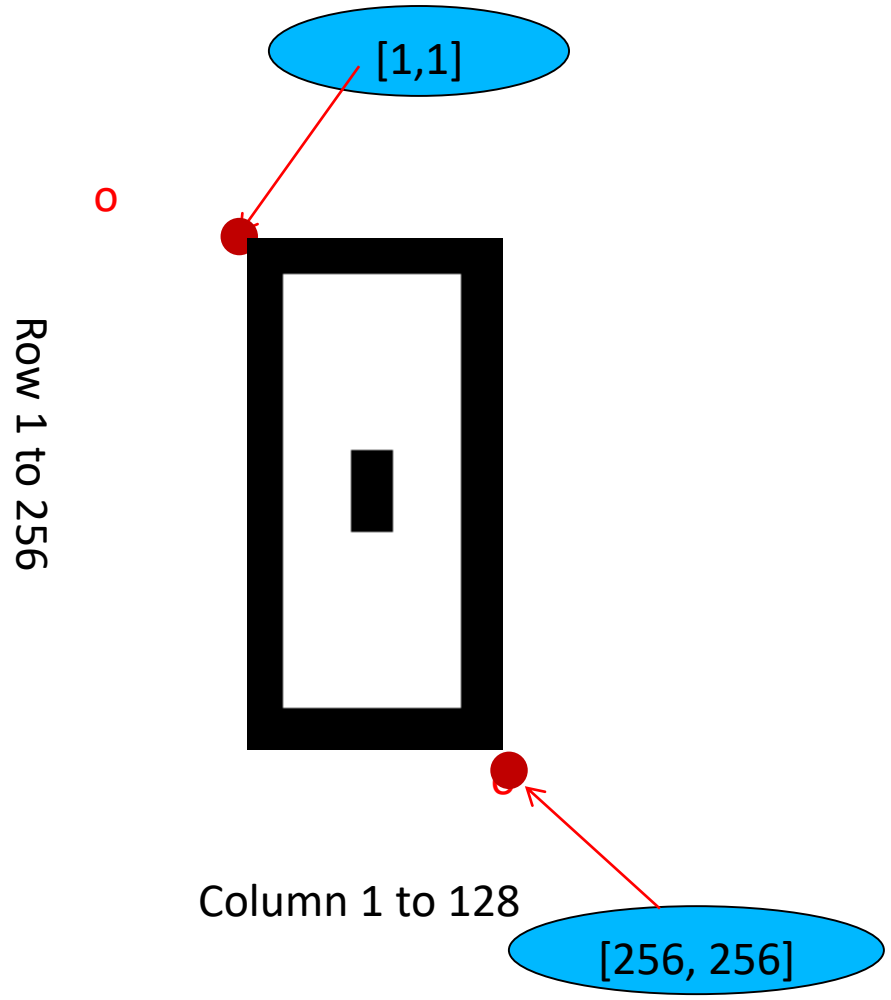
Complement

```
A=imread('maas.jpg');
A=rgb2gray(A);
% complement
B=imcomplement(A);
subplot(2,1,1)
imshow(A);
subplot(2,1,2)
imshow(B);
```

Images and Matrices

Create images

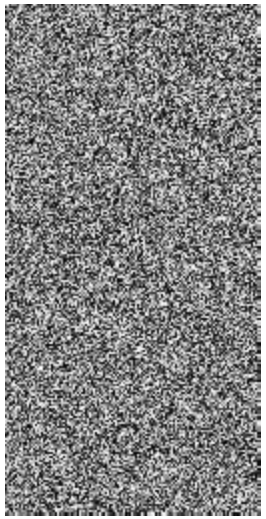
```
% create image
r=256;
c=128;
my_image=zeros(r,c);
my_image(20:236,20:108)=255;
my_image(108:148,54:74)=0;
imshow(my_image)
```



Images and Matrices

Binary Image:

```
% create random image
r=256;
c=128;
my_image=rand(r,c);
imshow(my_image)
```



Performance Issues

- The idea: MATLAB is
  - very fast on vector and matrix operations
  - Correspondingly slow with loops

- Try to avoid loops
- Try to vectorize your code

  http://www.mathworks.com/support/tech-notes/1100/1109.html

Vectorize Loops

- Example
  - Given image matrices, A and B, of the same size (540*380), blend these two images

```
apple = imread('apple.jpg');
orange = imread('orange.jpg');
```

- Poor Style

```
% measure performance using stopwatch timer
tic
for i = 1 : size(apple, 1)
   for j = 1 : size(apple, 2)
         for k = 1 : size(apple, 3)
               output(i, j, k) = (apple(i, j, k) + orange(i, j, k))/2;
         end
   end
end
toc
```

- Elapsed time is 0.138116 seconds

Vectorize Loops (cont.)

- Example
  - Given image matrices, A and B, of the same size (600*400), blend these two images

```
apple = imread('apple.jpg');
orange = imread('orange.jpg');
```

- Better Style

```
tic  % measure performance using stopwatch timer
Output = (apple + orange)/2;
toc
```

  - Elapsed time is 0.099802 seconds

- Computation is faster!