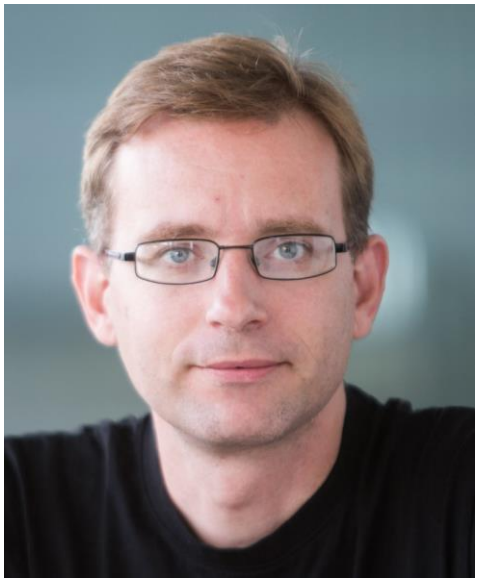
CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

Machine Vision

Lecture 1: Module Introduction

Introduction



- Dr Christian Beder
  - Lecturer at the Department of Computing
  - Email: Christian.Beder@cit.ie
  - Office Room: B180A

- Qualification & Experience:
  - Researcher at CIT's Nimbus centre 2010-2019 working on large-scale collaborative European Research Projects
  - PhD in Photogrammetry 2006
  - MSc in Computer Science 2002

What is Machine Vision?

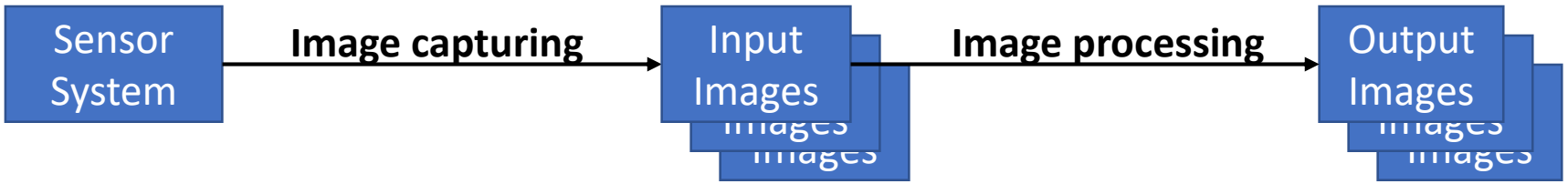
- Machine Vision (or Computer Vision) is the field of study that aims at extracting relevant information from images
- It tries to replicate our visual perception, i.e. the ability to interpret the environment using visible light
- Central to machine vision are images, which are mappings from a (usually) 2D image space to a (usually) 3D colour space
- We will also look at sequences of images over time, i.e. videos

What is Machine Vision?

- In this module we will address three main areas
  - Image capturing & processing
  - Content extraction
  - Scene reconstruction

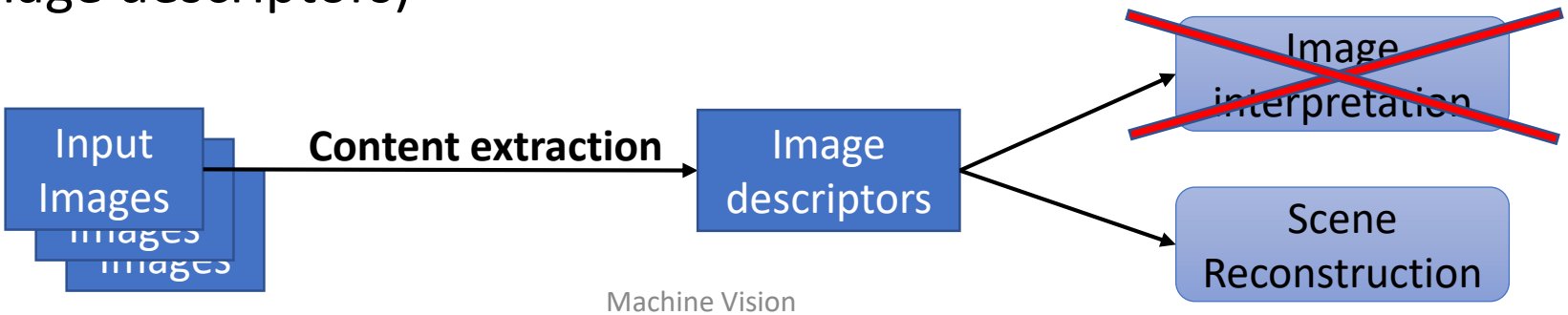
What is Machine Vision?

- **Image capturing** is the process of acquiring image data from a sensor system (e.g. a camera)
- **Image processing** are all algorithms that take image data as input and produce image data as output
- Results of image processing can be visualised and can be the input to more image processing



What is Machine Vision?

- **Content extraction** algorithms use images as input and generate some (typically) vector valued output data describing the content of the images
- These are typically positions and/or numerical descriptions of something visible in the images
- The descriptors can be the input to other algorithms, typically a mixture of scene reconstruction and image interpretation
- (In this module we will not cover image interpretation, which is typically done by applying **machine learning** algorithms to the image descriptors)



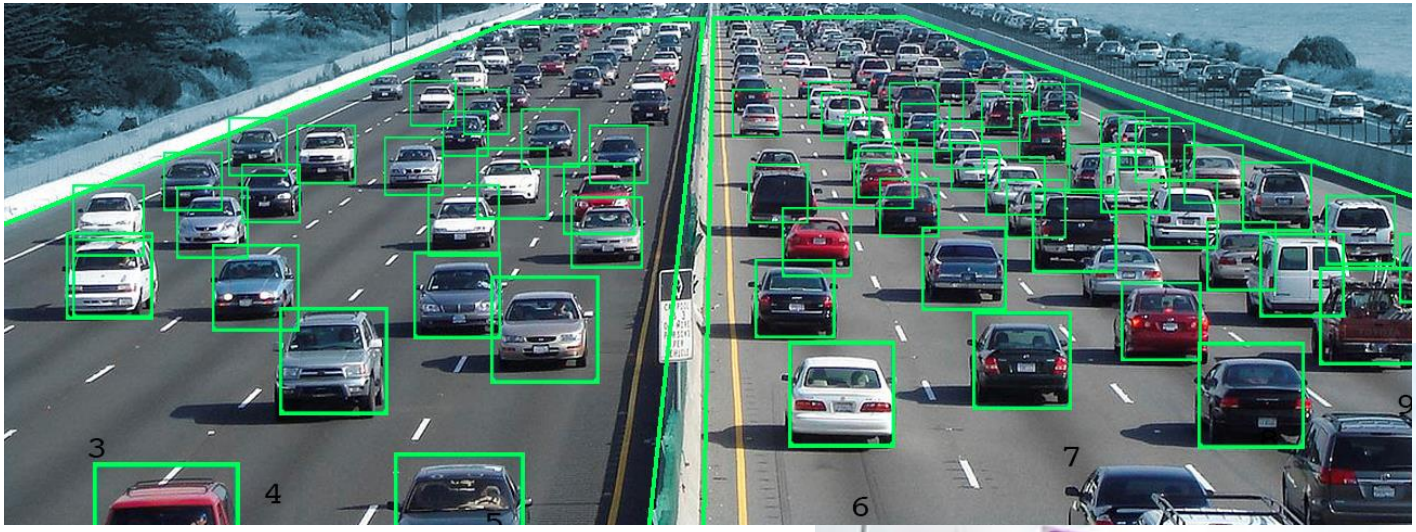
What is Machine Vision?

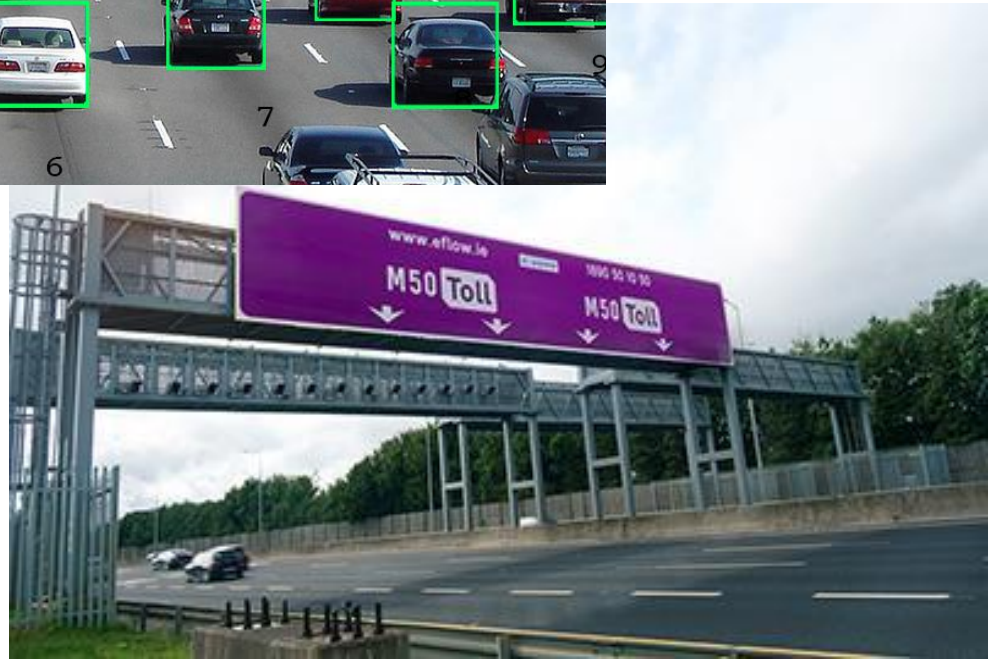
- Using the extracted image descriptors, **scene reconstruction** algorithms finally derive properties of the depicted objects

- We will look in depth at deriving accurate geometric measurements from image data, which is important in many applications

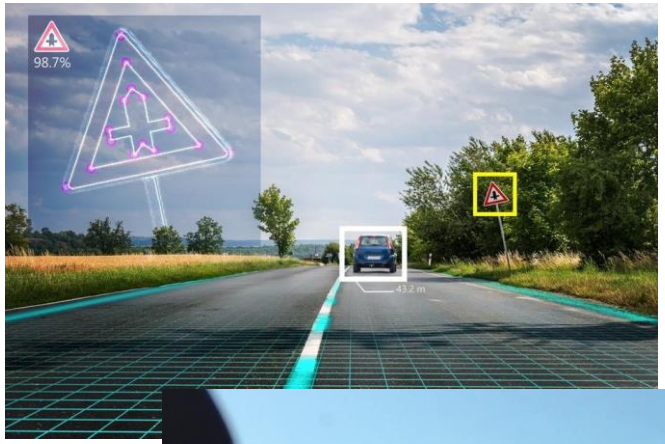
Real world example: Quality control

Real world example: Traffic management

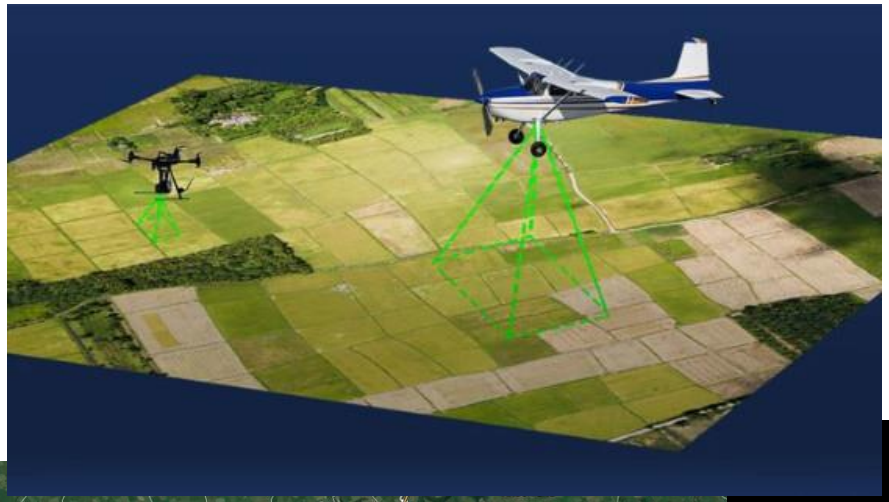


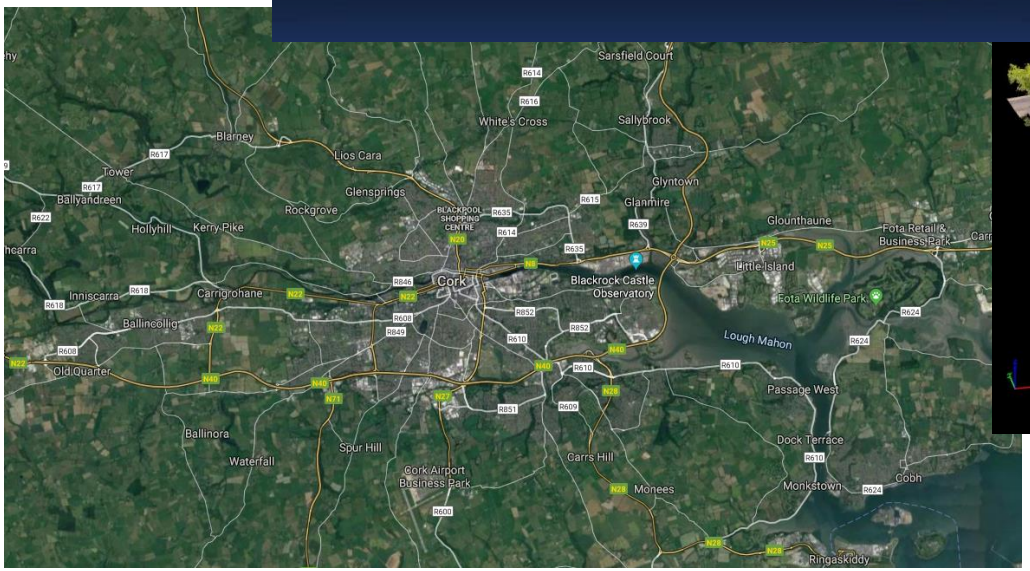


Real world example: Driving assistants

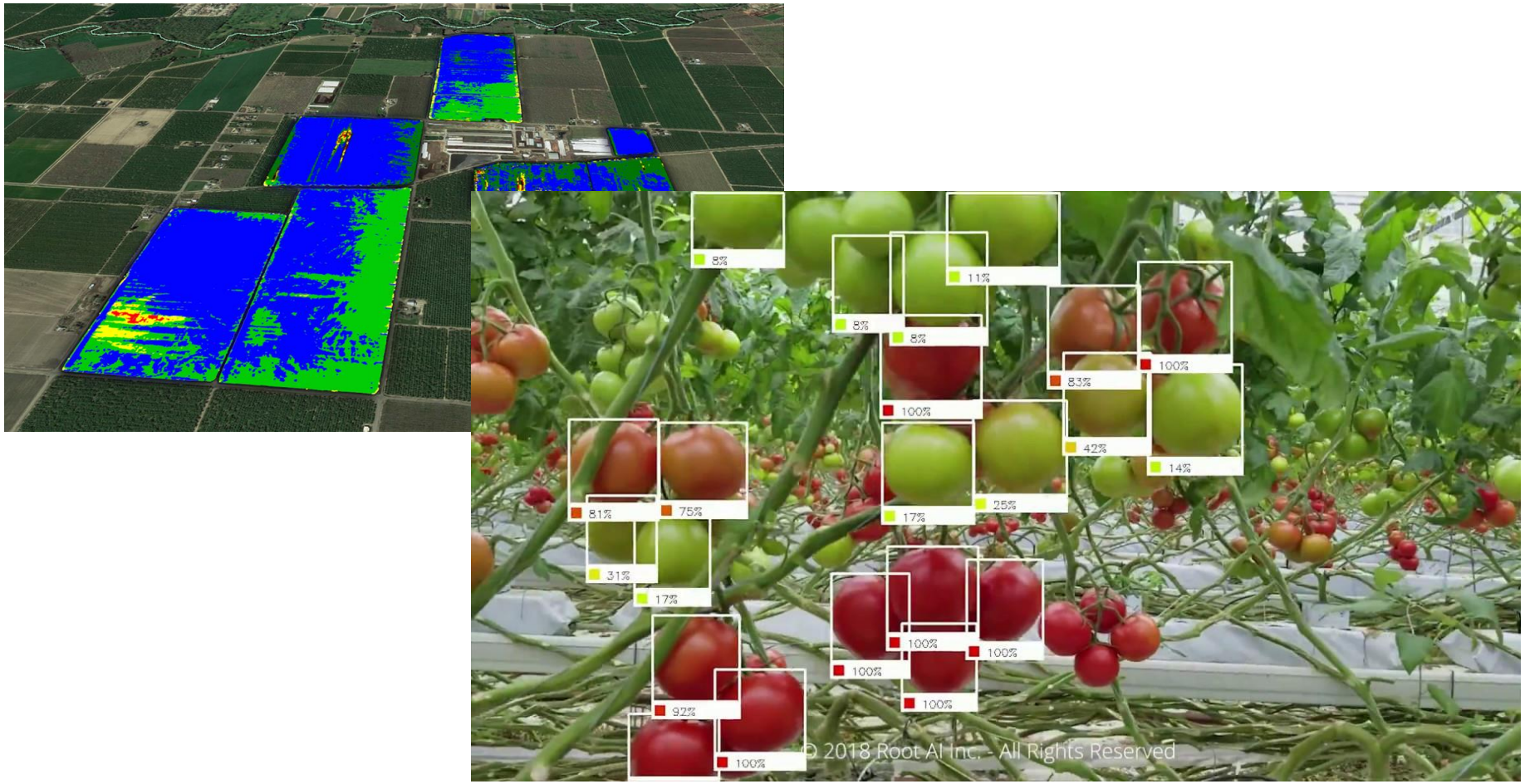


Real world example: Mapping

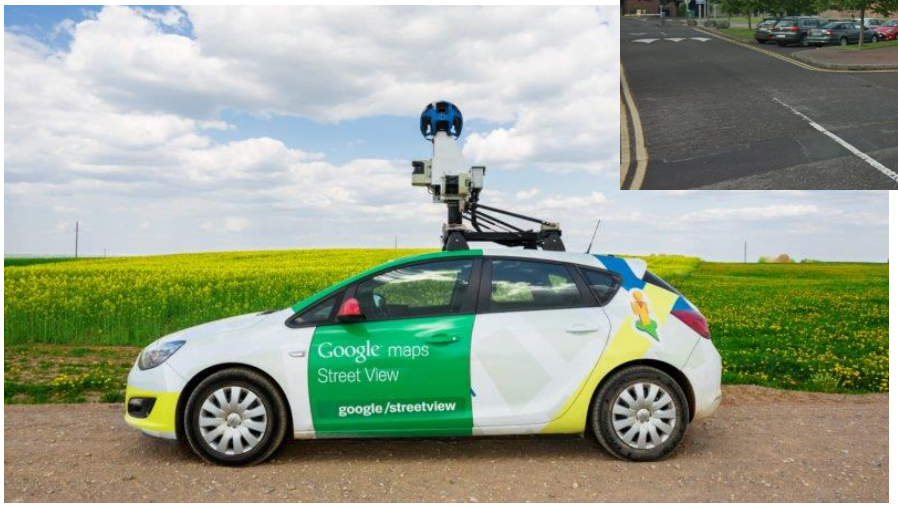




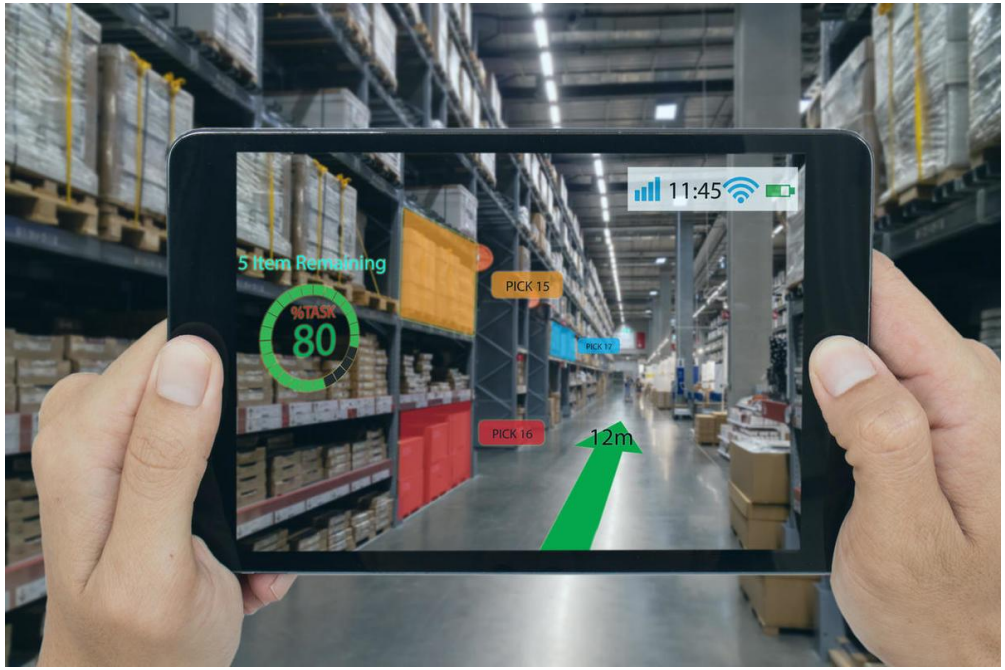
Real world example: Agriculture



Real world example: Panoramic images



Real world example: Augmented Reality



Real world example: Goal line technology

Real world example: Virtual TV studios

Real world example: Motion Capture

Real world example: Space exploration

Learning outcomes

Module Descriptor:
https://courses.cit.ie/index.cfm/page/module/moduleId/14577

- Apply computer vision methodologies to facilitate capturing, filtering and pre-processing of image and video data.
- Select and apply appropriate computer vision algorithms to solve real-world problems involving image and video data.
- Implement computer vision algorithms to extract and track relevant features from image and video data.
- Apply projective modelling techniques to implement parameter estimation algorithms for inferring and measuring scene geometry from image and video data.
- Analyse and evaluate the performance of computer vision and photogrammetry algorithms.

Module breakdown & assessment

Module Descriptor:
https://courses.cit.ie/index.cfm/page/module/moduleId/14577

- 2h lecture / week (**Mon 6pm-8pm**)
  - Presentation of theory and concepts
  - Every lecture will be accompanied by a lab exercise for you to apply the concepts

- 2h online support / week
  - …for your labs applying concepts seen in class
  - …for your work on the assignments

  - 1h live discussion on Zoom every **Mon afternoon (TBD)**
  - There is a Quiz on Canvas to determine your preferred timeslot
  - Use the **Canvas Discussion Board** for all your questions

  (please do not send questions to me by mail, as other students will not benefit in this case)

Module breakdown & assessment

Module Descriptor:
https://courses.cit.ie/index.cfm/page/module/moduleId/14577

- This module is 100% continuous assessment
  - 50% assignment due in week 8 (20/03/2020)
  - 50% assignment due in week 13 (08/05/2020)

- Assignment 1:

For a given case-study apply computer vision methodologies and implement algorithms to capture, process, and extract relevant information from image or video data.

- Assignment 2:

For a given case-study implement algorithms to extract and measure features from image or video data, estimate scene geometry parameters, and evaluate the accuracy of the results.

Module content

Module Descriptor:
https://courses.cit.ie/index.cfm/page/module/moduleId/14577

- L01-L03: Image capturing & processing
  - linear filters, Fourier transformation, Low-pass/high-pass filters, image pyramids, edge detection, colour representation, histograms, discretisation

- L04-L06: Image content extraction
  - Feature detection, local descriptors, feature matching, image segmentation, binary image descriptors, shape from shading, optical flow, feature tracking

  => Assignment 1

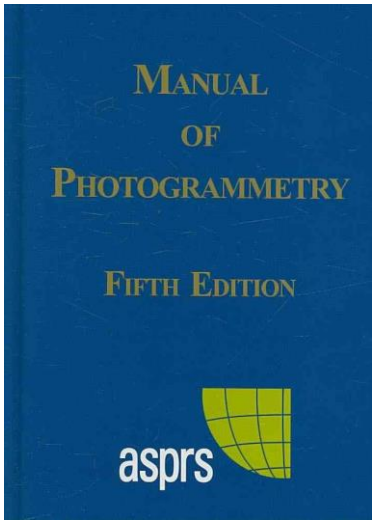
Module content

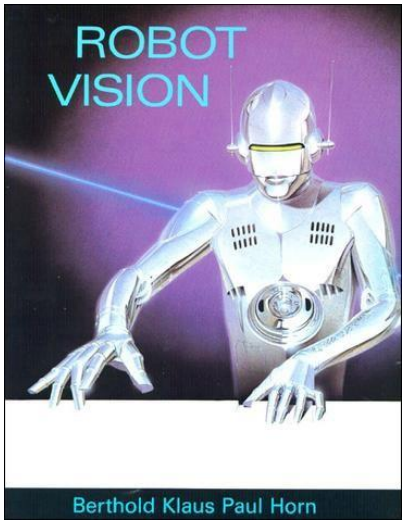
Module Descriptor:
https://courses.cit.ie/index.cfm/page/module/moduleId/14577

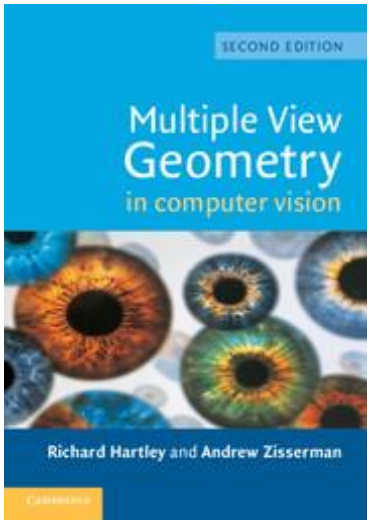
- L07-L08: Projective geometry
  - Projective camera model, planar homographies, image stitching, multi-view geometry, parameter estimation techniques
- L09-L12: Scene reconstruction
  - Camera calibration, metric scene reconstruction, bundle adjustment, simultaneous localisation and mapping

=> Assignment 2

Recommended resources





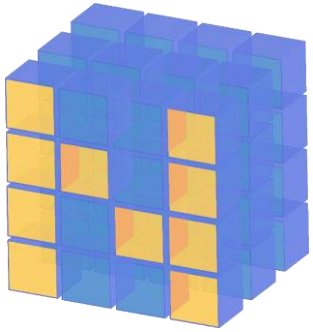


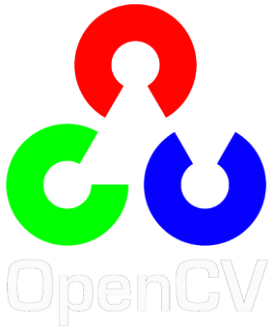
https://opencv.org/

http://homepages.inf.ed.ac.uk/rbf/CVonline/

Development environment

- All lab exercises and assignments will be developed in Python3
- The recommended IDE is Spyder and is part of the Anaconda package: https://www.anaconda.com/
- The Anaconda distribution already includes NumPy: https://numpy.org/
- In addition you will need to install the OpenCV library into your Python environment: https://pypi.org/project/opencv-python/



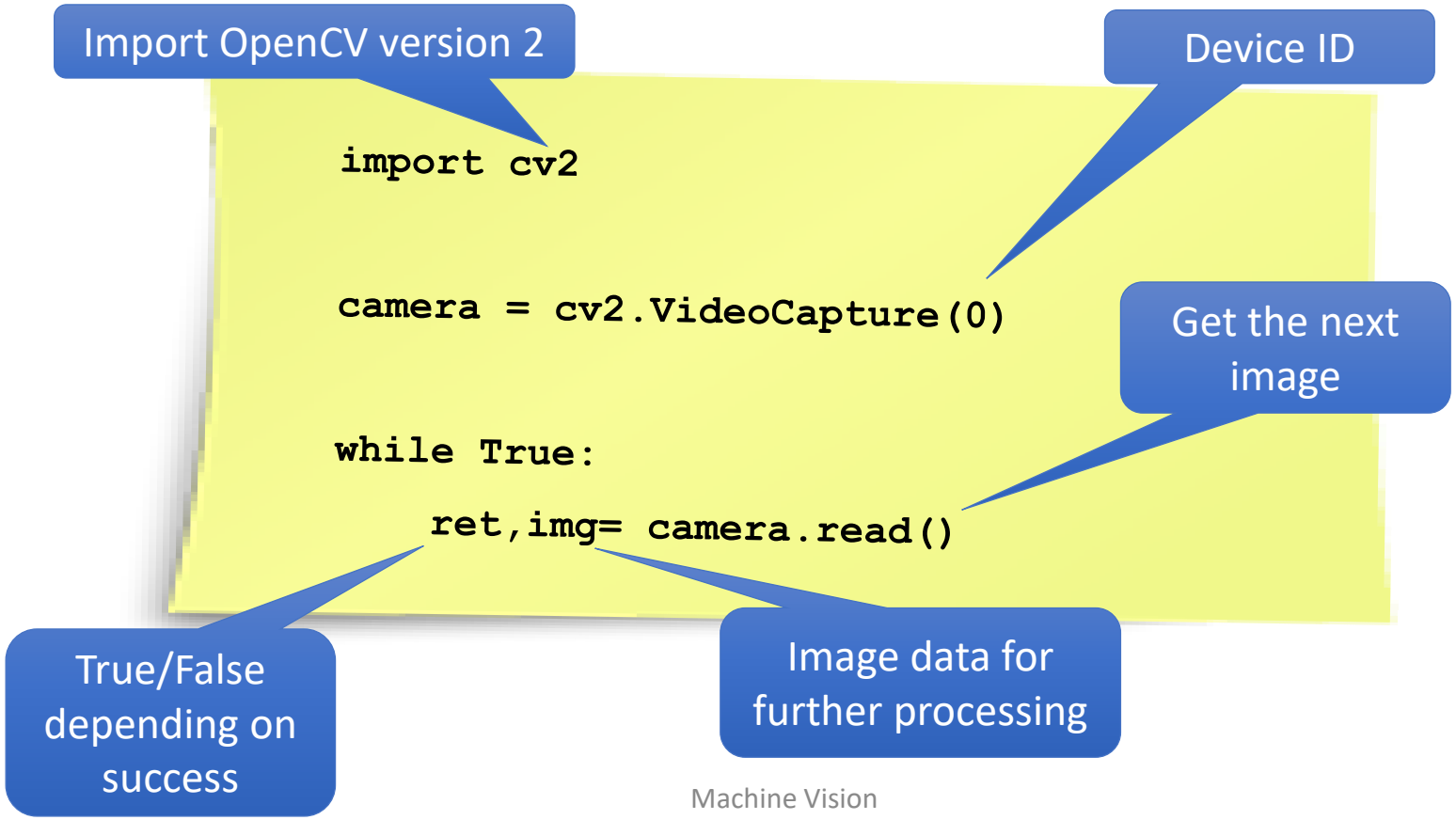


Introduction to OpenCV

- OpenCV is an open-source collection of algorithms for image processing and computer vision
- It contains functions for
  - Capturing images and videos
  - Loading/saving images and videos
  - Converting and processing images and videos
  - A lot of computer vision and machine learning algorithms
  - Some basic GUI functions for displaying images and videos
- It has interfaces for C++, Java, MATLAB and Python
- This link provides good tutorials on getting started with OpenCV in Python

Capturing images from a camera

- We use **`cv2.VideoCapture`** to get images from a capturing device like a camera



Processing a video file

- The function **`cv2.VideoCapture`** can also be used to process a video file

```
import cv2


video = cv2.VideoCapture("input.avi")
while True:
    ret,img= camera.read()
    if not ret:
        break
```

Filename instead of device ID

Stop processing at the end of the video

Recoding a video

- The function **`cv2.VideoWriter`** is used to encode and record a video into a file

```
import cv2


fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('output.avi', fourcc, 20.0, (640,480))


camera = cv2.VideoCapture(0)
while True:
    ret,img= camera.read()
    out.write(img)
```

Create a video codec

Set output file name

Set a frame rate

Record camera images into a file

Loading/saving an image file

- The functions **`cv2.imread`** and **`cv2.imwrite`** are used to load images from files and save images to files

```
import cv2


img = cv2.imread("input.jpg")
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
cv2.imwrite("output.jpg", gray)
```

Load image from file

Save image to file

Convert RGB colour image into a grayscale image

Displaying images

- Some basic GUI functions for displaying images and getting reading input are part of OpenCV

```
import cv2
camera = cv2.VideoCapture(0)
while True:
    ret,img= camera.read()
    cv2.imshow("camera", img)
    k = cv2.waitKey(1)
    if k%256==27:
        break
cv2.destroyWindow("camera")
```

Keyboard events on the windows can be processed

Named windows can be created to display images

Windows should be closed again

Introduction to NumPy

- NumPy is an open-source add-on module to Python that provide common mathematical and numerical routines in **pre-compiled**, fast functions.

- The NumPy (Numeric Python) package provides basic routines for **manipulating large arrays** and matrices of numeric data.

- At the core of the NumPy package, is the **Ndarray** object.
  - This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.
- NumPy arrays facilitate mathematical and other types of operations on large numbers of data.
  - Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
  - Minimize dependency on **loops**
- This link provides a good tutorial on getting started with NumPy.

NumPy – Slicing Arrays

- Array can be sliced just as with lists using the ‘**:** ‘notation within the square brackets

```
import numpy as np


arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = arr1[1:3]
print (arr2)
```

[45.6 3.2]

It is <u>important</u> to understand that a slice represents a view of the original array and references the data items in the original array.

NumPy – Slicing Arrays

```
arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = arr1[1:3]
print (arr2)
arr2[0] = 12
print (arr1)
```

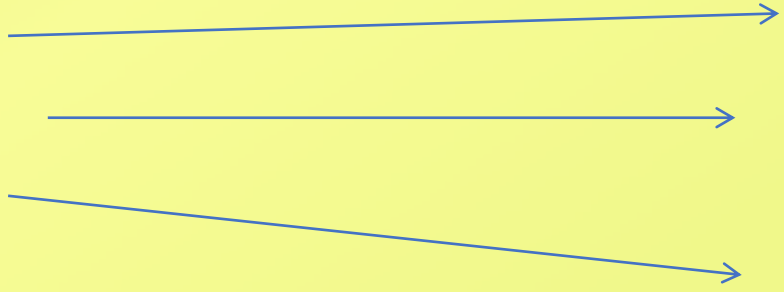
```
[ 45.6  3.2]

[ 5.5 12.   3.2]
```

- Notice that the change made to the sliced NumPy array (arr1) is reflected in the original NumPy array (arr2).

NumPy – Multi-Dimensional Arrays

- Arrays can be multidimensional. Elements are accessed using [row, column] format inside bracket notation.
- Most of the time we will be working with 2D arrays

```
arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)
print (arr[0, 0])
print (arr[1, 2])
```



```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

1.0

6.0
```

Slicing in 2D Arrays

- A single index value provided to a multi-dimensional array will refer to an entire row.
- We can use slicing to access an individual column by accessing all rows and specific columns

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)

print (arr)
print (arr[1])

print (arr[:, 0])

print (arr[:, [0,2]])
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

[ 4.  5.  6.]

[ 1.  4.]

[[ 1.  3.]
 [ 4.  6.]]
```

Slicing in 2D Arrays

Of course we can also provide a start and stop index for a slice (just as we did with lists previously).

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], float)
print (arr)

arr2 = arr[1:3, 0:2]
print (arr2)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]

[[ 4.  5.]
 [ 7.  8.]]
```

Exact everything from row 1 and 2 for the column 0 and 1

NumPy – Broadcasting

- NumPy has a useful broadcasting functionality that allows us to apply operations to a entire subset of data items.

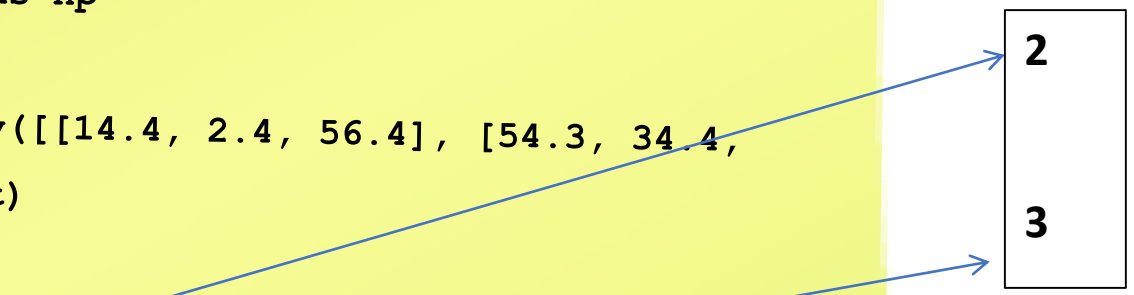
```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr[0, 1] = 12.2
print (arr)

arr[0] = 12.2
print (arr)

arr[:,0] *= 2
print (arr)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

[[ 1.  12.2  3. ]
 [ 4.   5.   6. ]]

[[ 12.2  12.2  12.2]
 [ 4.   5.   6. ]]

[[ 24.4  12.2  12.2]
 [ 8.   5.   6. ]]
```

Use of len Function in M-D Arrays

- len function can be used to obtain the number of rows or the number of columns
  - len of 2D array will return the **number of rows**
  - len of 2D row will return the **number of columns within that row**

```
import numpy as np

arr = np.array([[14.4, 2.4, 56.4], [54.3, 34.4,
98.22]], float)

print (len(arr))
print (len( arr[0] ) )
```



NumPy – Appending to MD Arrays

- We can add elements using append to MD arrays in NumPy
  - numpy.**append**(*arr*, *values*, ***axis=None***)
    - arr - Values are appended to a copy of this array.
    - values - These values are appended to a copy of arr. It must be of the correct shape
    - axis = The axis along which values are appended. **If axis is not given, both arr and values are flattened before use.**
      - In Numpy dimensions are called axes.

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [7, 8, 9])

print (arr1)
```

Notice the output array has been flattened. This is because no axis was specified

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
[ 1.  2.  3.  4.  5.  6.  7.  8.  9.]
```

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis= ?)

print (arr1)
```

axis = 0 refers to the vertical axis

axis = 1 refers to the horizontal axis

Dimension of values being added must be same as the specific axis we are adding to

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis= 0)

print (arr1)
```

Add a row containing the values [7, 8, 9] to axis = 0

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis= 1)

print (arr1)
```

Add a column containing the values [7, 8, 9] to axis = 1

Generates an **error** specifying array dimensions don't match because each column only contains two values (not three)

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
```

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print arr

arr1 = np.append(arr, [[7],[8]], axis = 1)


print arr1
```

Notice we use two [] brackets, that is because we are adding a single column element to each row

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

[[ 1.  2.  3.  7.]
 [ 4.  5.  6.  8.]]
```

Reshaping Arrays

- The ***arange*** function is similar to the range function but returns a NumPy array
- Only possible to create 1D array with arange
- Arrays can be reshaped by specifying new dimensions with reshape

```
import numpy as np


arr1 = np.arange(0,100, dtype=float)
arr2 = arr1.reshape((10, 10))
print (arr2)
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
 [20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]
 [30. 31. 32. 33. 34. 35. 36. 37. 38. 39.]
 [40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]
 [50. 51. 52. 53. 54. 55. 56. 57. 58. 59.]
 [60. 61. 62. 63. 64. 65. 66. 67. 68. 69.]
 [70. 71. 72. 73. 74. 75. 76. 77. 78. 79.]
 [80. 81. 82. 83. 84. 85. 86. 87. 88. 89.]
 [90. 91. 92. 93. 94. 95. 96. 97. 98. 99.]]
```

Obtain Max or Min in an Array

- For multi-dimensional arrays we can specify the axis.
  - If we don't specify the axis it will determine the maximum for the entire array
- The way to understand it is whichever axis you are using will be 'collapsed' into the shape of the array. If axis is 0 the collapse is down to rows.

```
import numpy as np

arr1 = np.array([[10,20,30],[50, 60, 10]], float)
print (arr1)

print (np.amax(arr1))

print (np.amax(arr1, axis=0))

print (np.amax(arr1, axis=1))
```

```
[[ 10.  20.  30.]
 [ 50.  60.  10.]]

60.0

[ 50.  60.  30.]

[ 30.  60.]
```

Basic Array Operations

- Many functions exist for extracting whole-array properties.
- The items in an array can be summed or multiplied:
- These functions can be performed on multi-dimensional arrays
  - We can also provide an additional element of the axis we wish to access
- http://docs.scipy.org/doc/numpy/reference/routines.math.html
- http://docs.scipy.org/doc/numpy/reference/routines.statistics.html

```
import numpy as np

arr1 = np.array([[1, 2, 4],[3, 4, 2]], float)
print (np.sum(arr1))
print (np.product(arr1))
print (np.mean(arr1, axis = 0))
print (np.std(arr1, axis = 1))
```

```
16.0

192.0

[ 2.  3.  3.]

[ 1.24721913
0.81649658]
```

Vector products

- Vectors operations, such as inner and outer products can be computed on arrays

```
import numpy as np

a = np.array([1.,2.,3.])
b = np.array([4.,5.,6.])

print(np.dot(a,b))
print(np.outer(a,b))
```

```
32.0

[[ 4.  5.  6.]
 [ 8. 10. 12.]
 [12. 15. 18.]]
```

Matrix/vector products

- Matrix/vector and Matrix/Matrix products need to be computed using the **`np.matmul`** function (**not** *)

```
import numpy as np

A = np.array([[1.,2.],[3.,4.],[5.,6.]])
b = np.array([7.,8.])

print(np.matmul(A,b))

B = np.array([[1.,2.,3.,4.],[5.,6.,7.,8.]])
print(np.matmul(A,B))
```

```
[23. 53. 83.]

[[11. 14. 17. 20.]
 [23. 30. 37. 44.]
 [35. 46. 57. 68.]]
```

Deleting a column/row from a NumPy array

- To delete a column or row from an existing array we can used numpy.delete(arr, index, axis=None)

```
arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
arr1 = np.delete(arr, 1, axis=0)

print (arr1)
```

In the code above we remove a row from the vertical axis (the row with index 1).

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[[ 1  2  3  4]
 [ 9 10 11 12]]
```

Array Mathematical Operations

- When **standard mathematical operations** are used with arrays, they are applied on an element by-element basis.
  - This means that the arrays should be the **same size** during addition, subtraction, etc
  - NumPy arrays support the typical range of operators +,-, *, /, %, **

```
import numpy as np

arr1 = np.array([[10,20], [30, 40]], float)
arr2 = np.array([[1,2], [3,4]], float)

print (arr1+arr2)
print ((arr1+arr2)*2)
```

```
[[ 11.  22.]
 [ 33.  44.]]

[[ 22.  44.]
 [ 66.  88.]]
```

Array Selectors

- We have already seen that, like lists, individual elements and slices of arrays can be selected using bracket notation. **Arrays also permit selection using other arrays**.

- That is, we can use **an array to filter** for specific subsets of elements of other arrays.
  - Before we look at this we must look at the result of using relational operators on NumPy arrays.

Comparison operators

- Boolean comparisons can be used to compare members element-wise on arrays of equal size.
- These operators (<,>, >=, <=, ==) return a **Boolean array** as a result

```
import numpy as np

arr1 = np.array([1, 3, 0], float)
arr2 = np.array([1, 2, 3], float)

resultArr = arr1>arr2
print (resultArr)
print (arr1== arr2)
```

[False True False]

[True False False]

Array Selectors

- We can use a Boolean array to **filter** the contents of another array.
- Below we use a Boolean array to select a subset of element from the NumPy array

```
import numpy as np

arr1 = np.array([45, 3, 2, 5, 67], float)

boolArr1 = [True, False, True, False, True]

print (arr1[boolArr1])
```

[45. 2. 67.]

Notice the program only returns the elements in arr1, where the corresponding element in the Boolean array is true

Comparison operators – Using relational operators to filter arrays

```
import numpy as np

arr1 = np.array([1, 3, 20, 5, 6, 78], float)

arr2 = np.array([1, 2, 3, 67, 56, 32], float)

resultArr = arr1>arr2
print (arr1[resultArr])
```

[3. 20. 78.]

Notice here we combine comparison operators and Boolean selection. This will print out all those values in arr1 that are greater than the corresponding value in arr2

Selecting Columns from 2D Array

```
import numpy as np

arr2D = np.array([[45, 3, 67],[12, 43, 73]], float)

boolArr4 = np.array([True, False, True], bool)
print (arr2D[:,boolArr4])
```

```
[[ 45.  67.]
 [ 12.  73.]]
```

Here we use booleans to select particular columns from a 2D array. We specify all rows using : and we select the first and last column for selection

Comparison Operators

- The following code applies a conditional operator to the column with index 1 and returns the rows that satisfy this condition.

```
[[ 1.  2.  3.]
 [ 2.  4.  5.]
 [ 4.  5.  7.]
 [ 6.  2.  3.]]

[[ 1.  2.  3.]
 [ 6.  2.  3.]]
```

```
import numpy as np

data = np.array([[1, 2, 3], [2, 4, 5], [4, 5, 7], [6, 2, 3]], float)
print (data)

# return all rows in array where the element at index 1 in a row equals 2
newdata = data[ data[:,1] == 2 ]
print (newdata)
```

Returns all rows in the 2D array such that the value of the column with index 1 in that row contains the value 2

Logical Operators

- You can combine multiple conditions using logical operators.
- Unlike standard Python the logical operators used are **&** and **|**

```
import numpy as np

data = np.array([[1, 2, 3], [2, 4, 5],
[4, 5, 7], [6, 2, 3]], float)

resultA = data[:,0]>3
resultB = data[:,2]>6

print ( data [ resultA & resultB ] )
```

Notice in the code we combine two conditions using & (we could chain as many conditions as we wish)

[[4. 5. 7.]]

Back to Array Selectors

- In addition to Boolean selection, it is possible to select using integer arrays.
- In this example the new array *c* is composed by selecting the elements from *a* using the index specified by the elements of *b*.

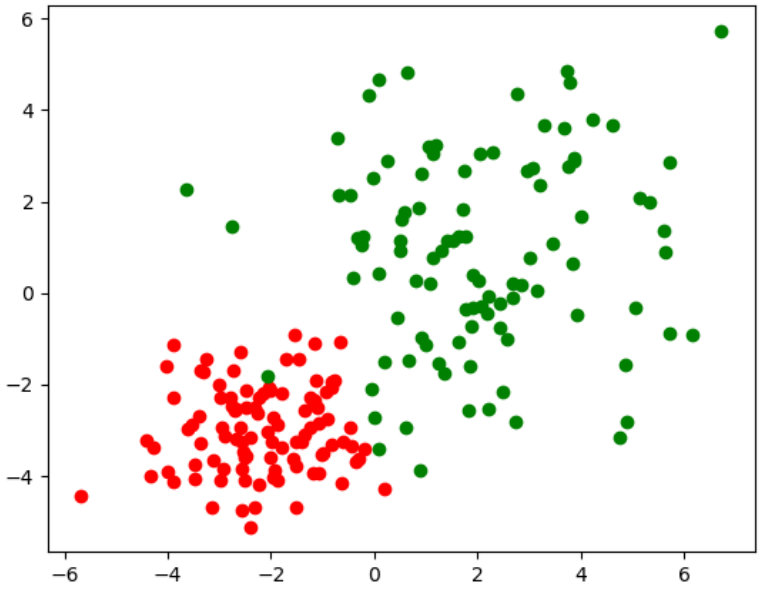
```
import numpy as np

a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int)
c = a[b]
print (c)
```

[2. 2. 4. 8. 6. 4.]

Notice the array c is composed of index 0,0, 1, 3, 2, 1 of the array a

Visualising data

- 2d scatter plots



Import matplotlib

```
import matplotlib.pyplot as plt

data1 = [-2,-3] + np.random.randn(100,2)
data2 = [2,1] + 2*np.random.randn(100,2)

plt.figure()
plt.scatter(data1[:,0], data1[:,1], color='r')
plt.scatter(data2[:,0], data2[:,1], color='g')
```

Create a figure

Add a scatter plot

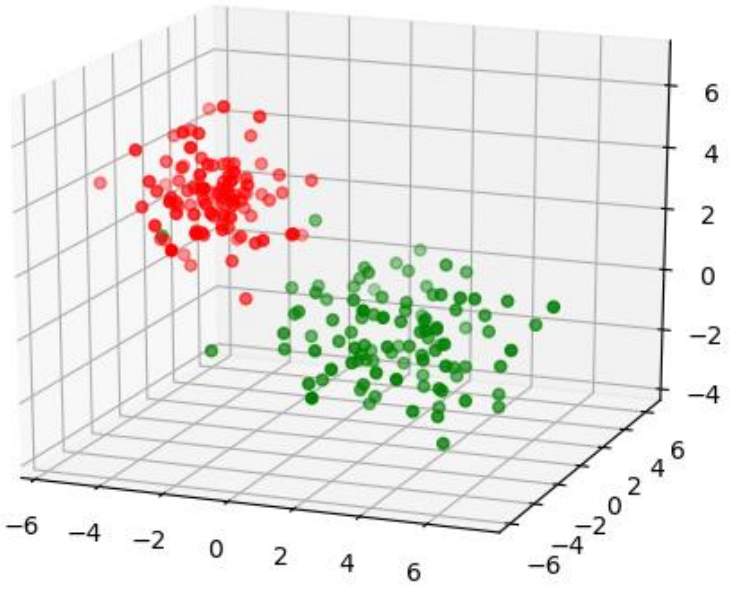
Colour of markers

Array of X-coordinates

Array of Y-coordinates

Visualising data

- 3d scatter plots



Import 3d axes

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(data1[0,:], data1[1,:], data1[2,:], color='r')
ax.scatter(data2[0,:], data2[1,:], data2[2,:], color='g')
```

Add 3d axes to plot

Call the scatter function on the 3d axes

Array of Z-coordiantes

Thank you for your attention!