Vision Toolbox for MATLAB

Manual

Embedded Target for the S32V234 Family of Automotive Vision Processors

Version 1.1.0

Target Based Automatic Code Generation Tools
For MATLAB™ working with Mathworks Image Processing and Computer Vision Toolboxes

Summary

1 Introduction

In this document, the NXP Vision Toolbox for S32V234 is described. The NXP Vision Toolbox can be used only in conjunction with the NXP S32V234 Vision SDK that support the Linux OS runtime environment.

The first part of this document covers the Vision Toolbox overview, installation and setup of required prerequisites. This includes necessary software packages and any collateral parts of the SW.

The second part then describes the main functionalities which are part of this Vision Toolbox for MATLAB. This part aims to provide an understanding of the basic functionality, such as using MATLAB toolbox wrappers with the Vision SDK.

1.1 Purpose

The purpose of this document is to present the NXP Vision Toolbox for S32V234 and help users to bring up examples quickly.

1.2 Audience

This document is intended to:

- MATLAB Computer Vision System users that wish to evaluate the NXP HW&SW solutions starting from existing applications written in m-scripts;
- S32V234 Vision SDK users that may wish to simulate the kernels in MATLAB for a better understanding and visualization capabilities;
- NXP S32V234 buyers that need to have a quick start-up into vision applications and ready to run examples;

1.3 References

This document does not cover the computer vision theory nor subjects related with how specific kernels and functions work. For more details about these please refer to:

ID	Title	Location
[1]	OpenCV Library	https://opencv.org/
[2]	The Modern History of Object Recognition—Infographic	https://medium.com/@nikasa1889/the-modern-history-of-object-recognition-infographic-aea18517c318
[3]	MATLAB Computer Vision Documentation	https://www.mathworks.com/help/vision/
[4]	NXP S32V234 Vision Processor	https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/s32-automotive-platform/vision-processor-for-front-and-surround-view-camera-machine-learning-and-sensor-fusion-applications:S32V234

1.4 Definitions, Acronyms and Abbreviations

Acronym	Description
ACF	APEX Core Framework
APEX	A parallel image processing accelerator HW block part of NXP S32V234 SoC.
APEX COMPILER	Set of tools (NXP APU compiler) that allow compilation of code for APEX subsystem
ARM	Family of RISC architectures
ISP	Image Signal Processor subsystem of the S32V234 SoC
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision
SDK	Software Development Kit

2 Vision Toolbox

This chapter describes the main NXP Vision Toolbox features. Note that everything needed to run and build the demos is installed with the help of an additional toolbox created for this purpose called NXP Support Package for S32V234.

The majority of applications included in NXP Vision Toolbox are demos, which demonstrates all possible uses of the toolbox and how to use MATLAB scripting to program and test applications on NXP S32V234 Vision Processor.

2.1 Programming modes

The NXP Vision Toolbox is targeted mainly for the vision processing algorithms on S32V234, which is aimed for fast, massively parallel image operations (APEX) and Image Signal Processing of the camera input (ISP).

Within the NXP Vision Toolbox there are two ways of programming the APEX cores available from MATLAB scripting:

- **APEX Core Framework (ACF)**: this method consists in writing special m-functions called graphs that are using dedicated APEX Kernels m-script wrappers which calls special routines from NXP Vision SDK optimized for performance. The following code snippet shows a simple rotate graph. The functions highlighted in bold are called APEX Kernels

```
function outImg = rotate_graph(inImg, inOffset)%#codegen
    nxpvt_set_chunk(1, 8, 8);
    outIndir = nxpvt.apu.indirect(inImg, inOffset);
    outImg = nxpvt.apu.rotate_180(outIndir);
end
```

  The code generated upon the call of `rotate_graph()` runs exclusively on the APEX core.

- **APEX Computer Vision**: this method consists in writing applications using special functions supported by the Vision SDK that mimics the OpenCV functionalities. These functions are special wrappers on top of Vision SDK classes that implement complex algorithms.

```
function rgb2gray_image_main()
    inImgPath = 'data/sobel.jpg';
    inImgUMat = nxpvt.imread(inImgPath);
    outImgUMatGray = nxpvt.apexcv.rgb2gray(inImgUMat);
    nxpvt.imshow(outImgUMatGray);
end
```

  The code generated upon the call of `APEXCV` functions may have code that is executed on both ARM and APEX cores.

The scope of this manual is to enable users to understand the existing examples, build and download the application to NXP target and not to describe the programming methods. In case you wish to become familiar with the APEX programming, please consult the Vision SDK documentation.

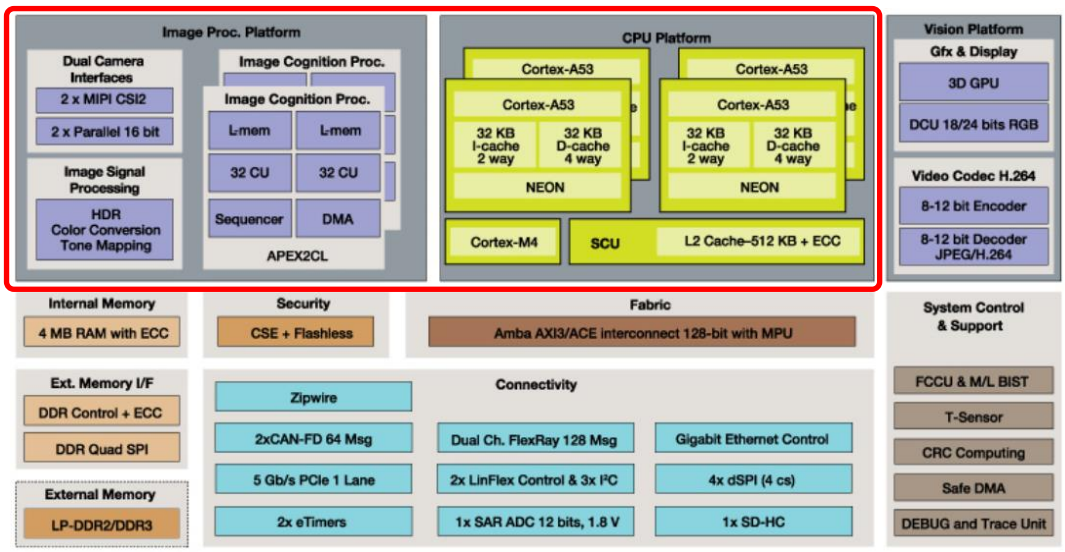
However, it's necessary to mention that all demos and examples provided as part of the NXP Vision Toolbox can be run as out-of-the-box software since the user is not forced to build any Vision SDK components. The NXP Vision Toolbox takes care of all setup necessary to run the applications shipped with the toolbox.

2.2 Main Features

The NXP Vision Toolbox for S32V234 is a prototype tool that helps you:

- Test vision algorithms using NXP Vision SDK functions in the MATLAB environment for a complete development, simulation and execution on the NXP targets by generating the C++ code directly from m-scripts using `nxpvt_codegen()`
- Use various I/O functions to control the NXP Evaluation Boards supported cameras and displays
- Program the NXP APEX cores directly from MATLAB environment using Apex Core Framework graphs
- Configure the NXP S32V Targets to enable code deployment directly from MATLAB environment and execute vision algorithm on NXP S32V Evaluation Boards
- Fast evaluation of NXP solutions using ready-to-run examples derived from MATLAB Computer Vision System Toolbox

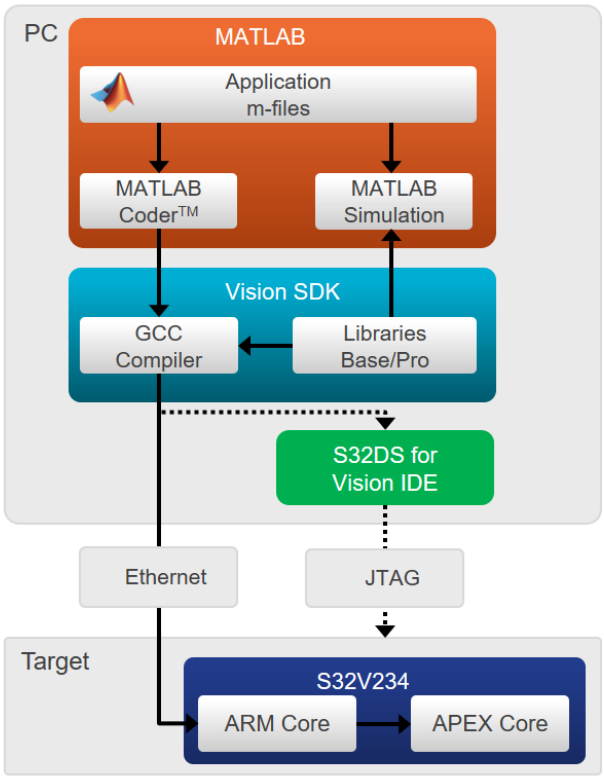
The NXP Vision Toolbox for S32V234 is designed to handle code generation based on NXP Vision Software Development Kit for CPU Platform (ARM A53 cores) and Image Processing Platform (APEX cores)



After the code generation, the NXP Vision Toolbox can be configured to download the application to the NXP target via TCP/IP. This mechanism requires to have a bootable SD-CARD configured with NXP u-boot and Linux images compatibles with the Vision SDK version used for code generation.

The NXP Vision Toolbox development flow is shown in the figure below. On the host-PC, running under MATLAB environment, one can start testing various algorithms using MATLAB simulation capabilities. Once the results satisfy the requirements, the MATLAB Coder can be employed to generate C++ code from m-scripts.

The C++ code is then cross-compiled on the host PC using the NXP build tools and Vision SDK libraries. The final application can then be loaded on the Target using dedicated MATLAB scripts available in NXP Vision Toolbox.



NXP offers additional tools like S32 Design Studio for Vision that can be used to import and debug the MATLAB generated code directly on the target.

For more information please check the Vision Toolbox quick start guide or watch this webinar

The NXP Vision Toolbox for S32V234 package contains:

- Optimized APEX kernel wrappers for the APEX image cognition processor and support for target code generation. For code generation the NXP Vision Toolbox works in conjunction with the NXP Vision SDK. It provides a mechanism to move from MATLAB to APEX quickly and easily. The user can prove their vision algorithm within MATLAB environment first, before moving to the target APEX processor. The toolbox eliminates lot of time consuming tasks like development of graph, process description, data descriptors etc. since they are automatically generated by the tool based on m-scripts.

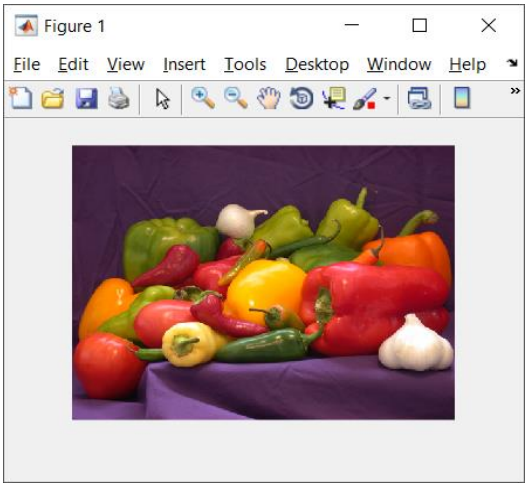
  Below is an example of Sobel Graph written in MATLAB using the Vision SDK Kernels highlighted in bold.

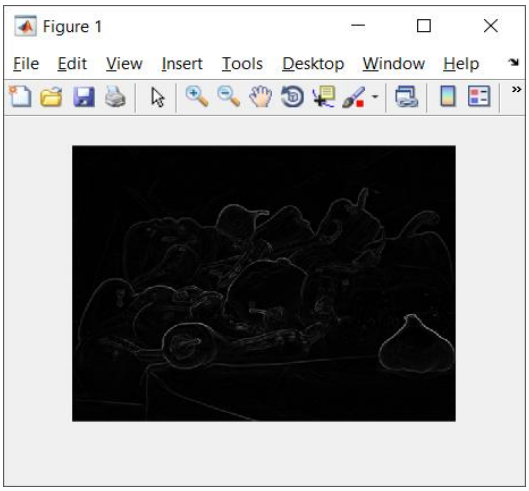
```
function [out, imgEdge] = sobel_graph(img)  %#codegen
    coder.inline('never');
    nxpvt_set_chunk(1, 8, 2);
    % Convert RGB image to grayscale
    imgGray = nxpvt.apu.rgb_to_grayscale(img);
    % Sobel edge detection method
    imgEdge = nxpvt.apu.filtering_sobel_3x3(imgGray);
    % Add two images
    out = nxpvt.apu.add(imgEdge, imgGray);
end
```

  During simulation these kernels act as any other MATLAB function allowing users to speed up the software development by checking the data at each processing step. To next code snipped shows how one can call such graph function to check the outcome.

```
function sobel_main()  %#codegen
    inImgPath = 'data/sobel.jpg';
    inImgUMat = nxpvt.imread(inImgPath);
    %% Sobel filter
    [~, imgEdgeUMat] = sobel_graph(inImgUMat);
    %% Output
    nxpvt.imshow(imgEdgeUMat);
end
```

  After running the `sobel_main()` you can simply check for results (input vs. output):





- Optimized APEX CV functions for the APEX image cognition processor and support for target code generation. These functions implement complex algorithm that can be used AS-IS from MATLAB m-scripts.

- Code generation utilities for transform m-scripts into C++ code that can be executed on ARM A53 or APEX cores depending on the type of data and kernels invoked in m-script files. By default, generic m-script code is converted into the C++ counterpart that is executed on the ARM core. In case you wish to take the full benefits of using the NXP Vision Accelerator, then you should use the available kernels exposed by the NXP Vision Toolbox to write your application.

- Target support utilities designed to configure the SD-Card with a bootable Linux OS image and capabilities to download and run the vision applications directly from MATLAB

- Ready-to-run examples that exercises various functionalities based on:
    - MATLAB Computer Vision System Toolbox demos
    - NXP Vision SDK examples

2.3 Processors Supported

The NXP Vision Toolbox 1.1.0 supports the NXP S32V234 Vision Processor. Testing and validation has been completed on production qualified parts mounted on:

- S32V234 Evaluation Board equipped with S32V-SonyCam
- SBC-S32V234 Evaluation Board equipped with S32V-SonyCam

2.4 MATLAB Required and Recommended Products

The NXP Vision Toolbox requires the following MathWorks products:

Product	Version Compatibility	Required or Recommended
MATLAB	R2018a/b	Required
MATLAB Coder	R2018a/b	Required
Embedded Coder	R2018a/b	Required
Image Processing Toolbox	R2018a/b	Required
Computer Vision System Toolbox	R2018a/b	Required
Embedded Coder Support Package for ARM Cortex-A Processors	R2018a/b	Required
Computer Vision System Toolbox OpenCV Interface	R2018a/b	Required
MATLAB Support Package for USB Webcams	R2018a/b	Recommended
Image Acquisition Toolbox Support Package for OS Generic Video Interface	R2018a/b	Recommended

Due to code generation performance issues the NXP Vision Toolbox uses a special feature `row-major` that has been introduces in MATLAB Coder 2018a. This feature allows better code generation that is compatible with embedded systems designed to store the arrays in row-major format avoiding this way unnecessary copies or transposes between MATLAB and Vision SDK APIs.

2.5 Build Tools

The NXP Vision Toolbox supports code generation for the NXP ARM GNU and NXP APU compilers.

Compiler	Versions Tested
NXP ARM GNU Compiler	NXP GCC 6.3.1
NXP APU Compiler	V1.0 build 530

The target compilers used for NXP Vision Toolbox needs to be configured. Use the notation below to setup these compiler environment or user variables. Ensure that such variables are defined to compiler path value as shown below:

```
APU_TOOLS= C:/NXP/APU_Compiler_v1.0
S32V234_SDK_ROOT = C:/NXP/VisionSDK_S32V2xx_RTM_1_2_0_HF1/s32v234_sdk
```

2.6 Installation

Installing the NXP Vision Toolbox is your first step to getting up and running on the NXP S32V234 Automotive Vision Processor. Please follow the installation steps below, and then explore the examples.

2.6.1 System Requirements

The NXP Vision Toolbox is supported only on PC with Windows OS. For a flowless development experience the minimum recommended PC platform is:

- Windows® 7/10 64bit Operating System
- At least 2 GHz CPU Speed
- At least 4 GB of RAM
- At least 20 GB of free disk space.
- Internet connectivity for web downloads.

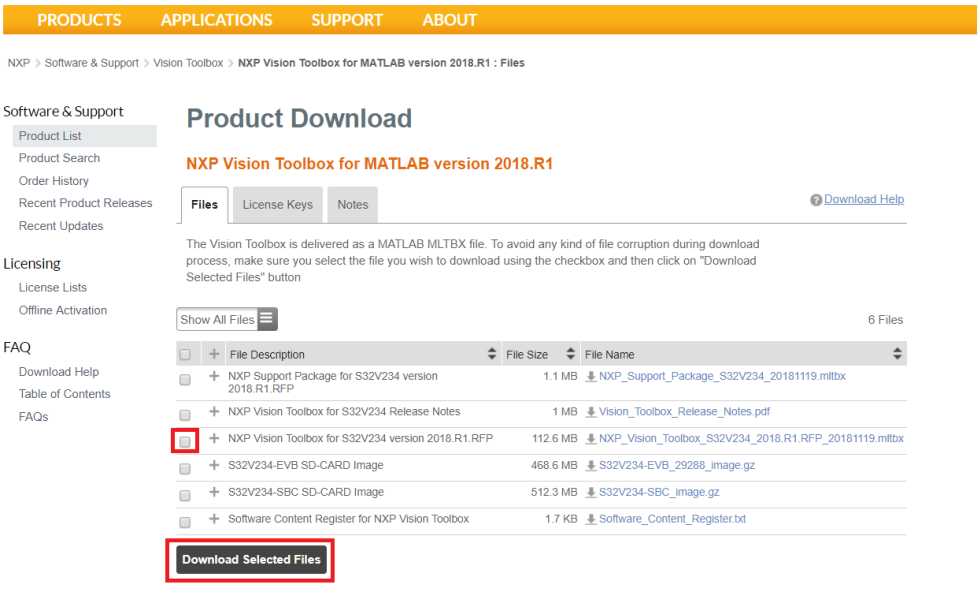
2.6.2 Vision Toolbox Installation

The complete installation procedure with detailed step-by-step screenshots is described in the `Vision_Toolbox_Quick_Start.pdf` located in `docs` folder or available via MATLAB Help. In this manual only, the summary of the installation steps is shown with details on specific dependencies.

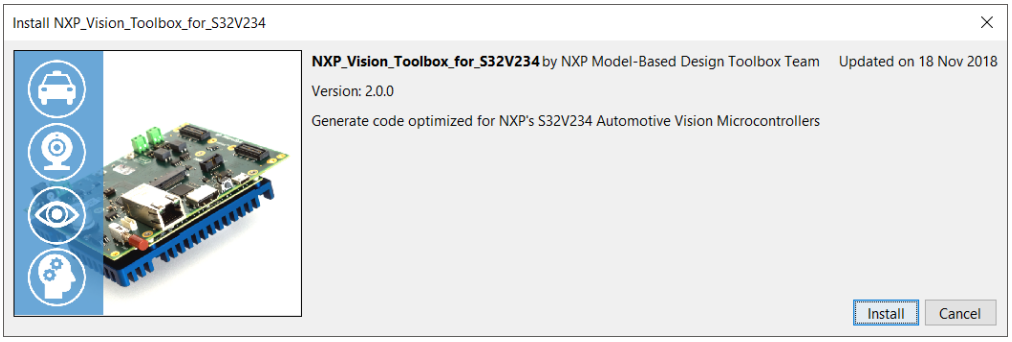
The NXP Vision Toolbox was designed to be installed as a MATLAB Add-on using Mathwork's Toolbox Installer technology. For this purpose, the NXP Vision Toolbox is delivered as a MLTBX file which is automatically recognized by MATLAB.

To install the NXP Vision Toolbox for S32V234, the following steps are required:

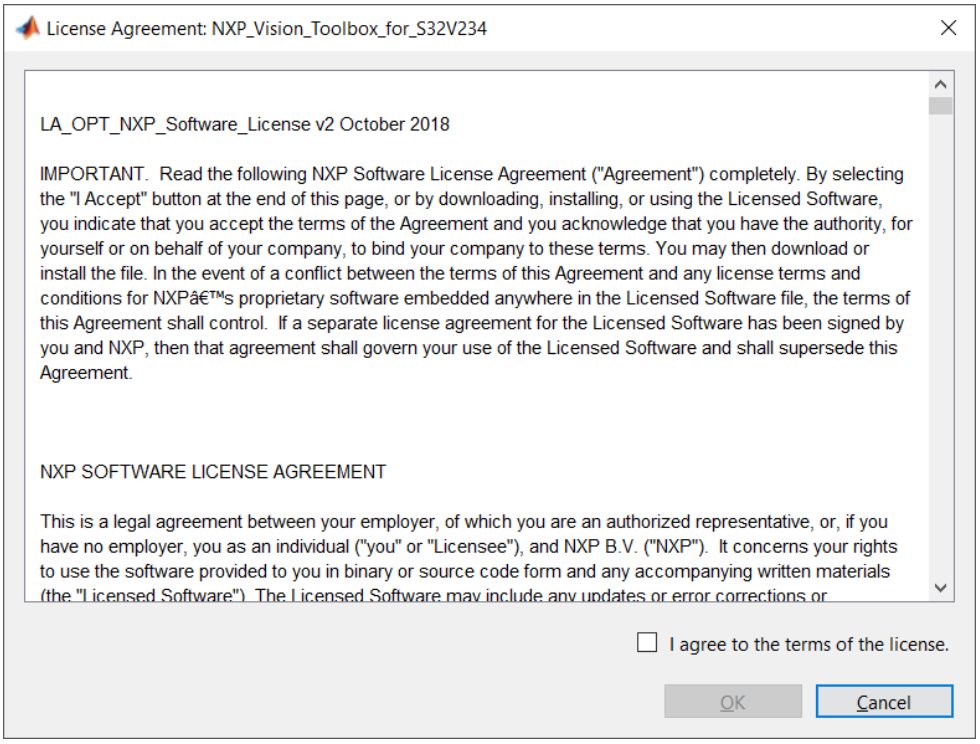
1. Go to NXP website and log-in into your account;
2. Use this link to access the NXP Vision Toolbox for S32V2334 MLTBX file;



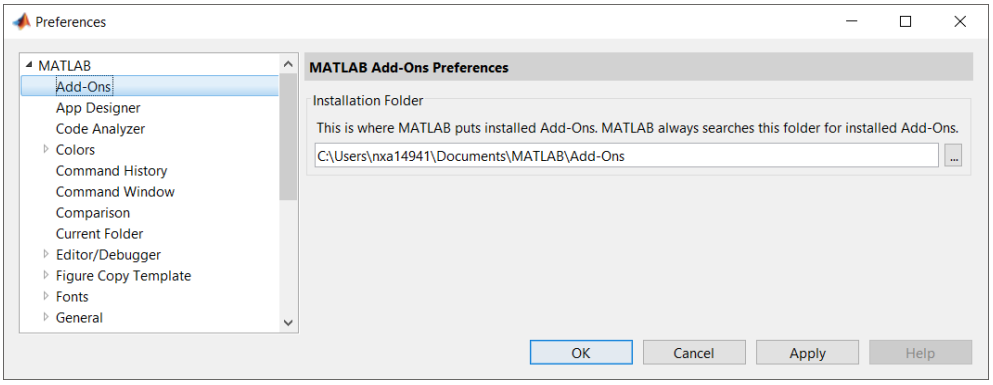
3. Make sure you download the NXP Vision Toolbox in MLTBX file format using `Download Selected Files` option.
4. Open the MLTBX file in MATLAB.



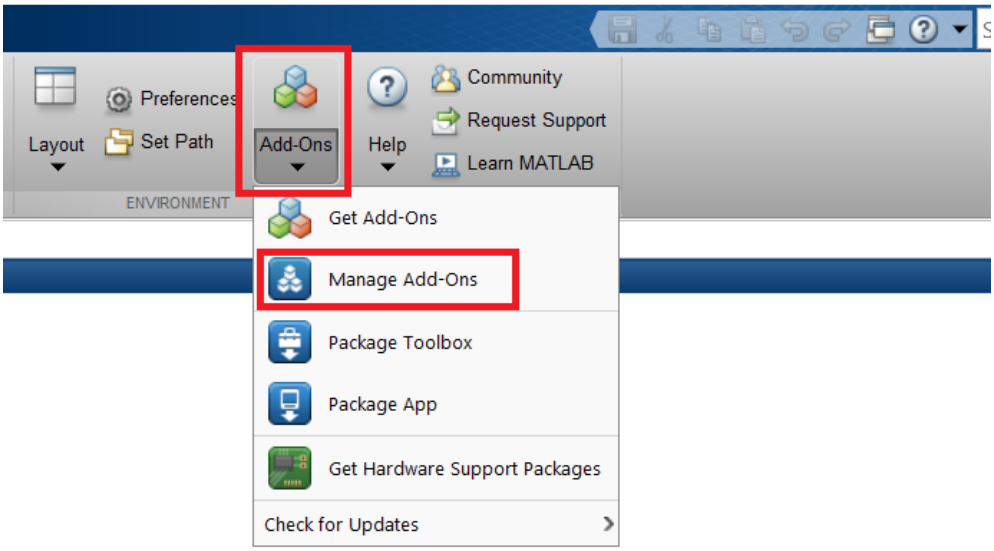
5. Indicate acceptance of the NXP Software License Agreement by selecting "`I agree to the terms of the license`" to proceed

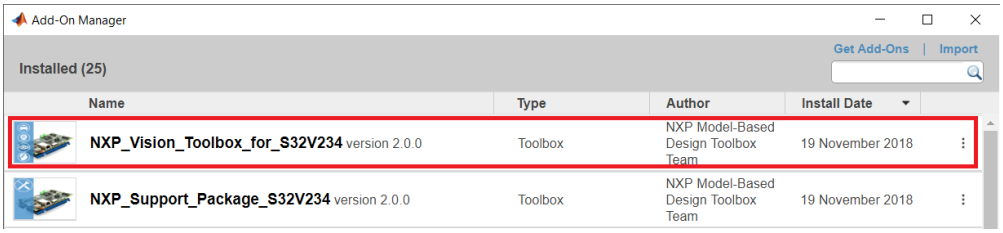


6. Click "OK" to start the MATLAB installation process. The rest of the process is silent and under MATLAB control. All the files will be automatically copied into default Add-Ons folder within the MATLAB. The default location can be changed prior to installation by changing the Add-Ons path from MATLAB Preferences

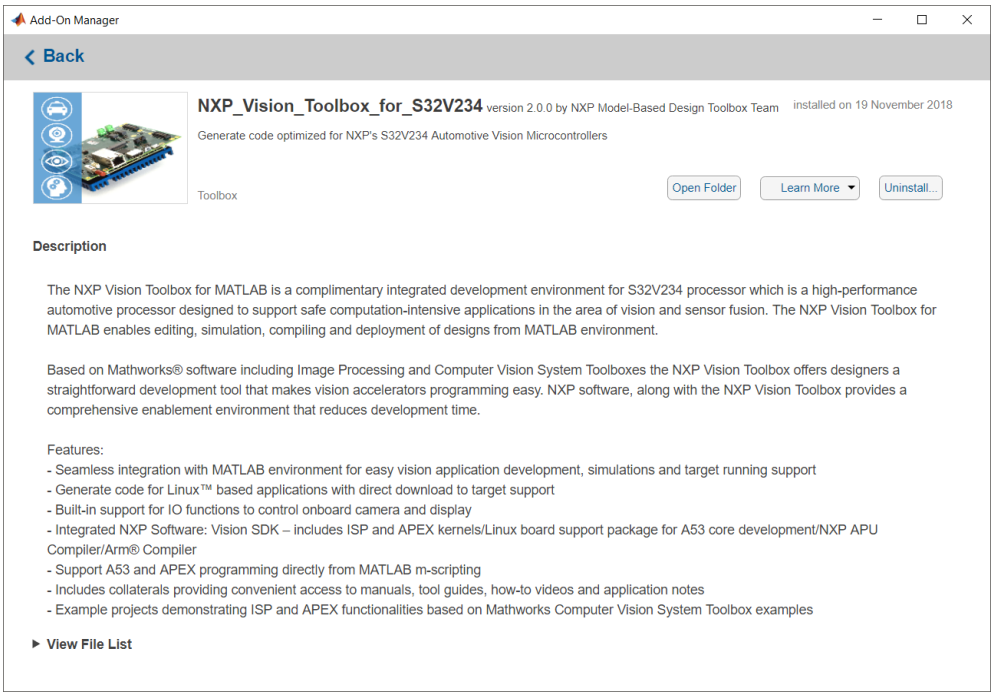


7. After a couple of seconds, the NXP's Vision Toolbox should be visible as a new Add-ons.





8. More details about the NXP's Vision Toolbox can be found by clicking on `View Details`

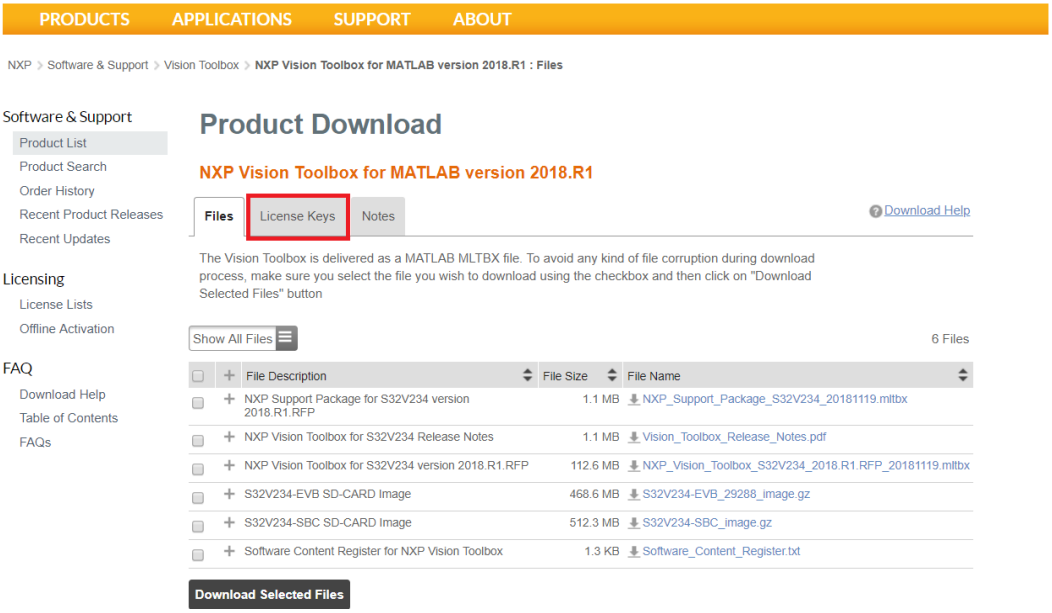


2.6.3 License Generation and Activation

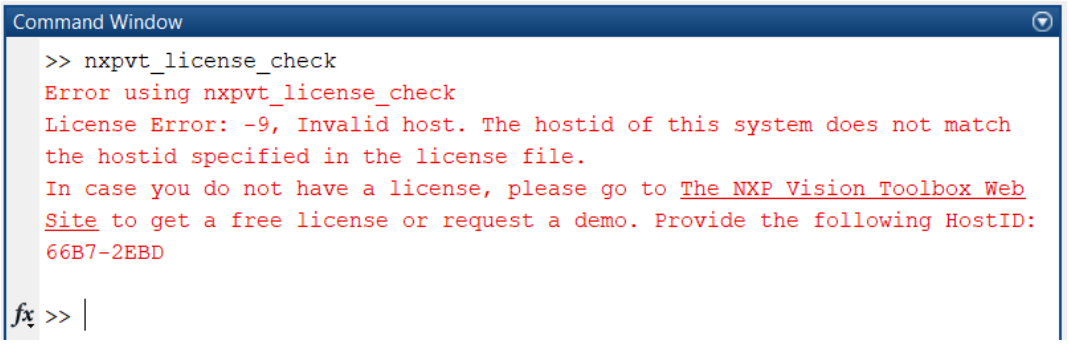
The NXP Vision Toolbox for S32V234 is available free of charge, however, a valid license is required to activate the Vision Toolbox.

The license can be generated for free. For more details and step-by-step guide please check the dedicated manual on this subject `Vision_Toolbox_License_Activation.pdf`

Log-in into your NXP Software Account and Generate the license using this link



To validate the license activation, run the command `nxpvt_license_check`. If there are issues with the license, this command will return the root-cause.

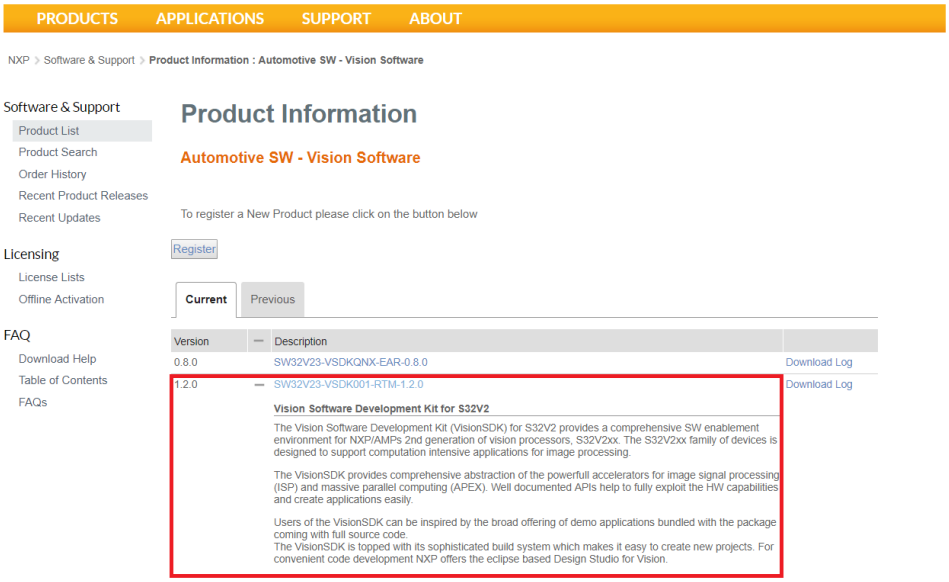


2.6.4 Vision SDK and Build Tools

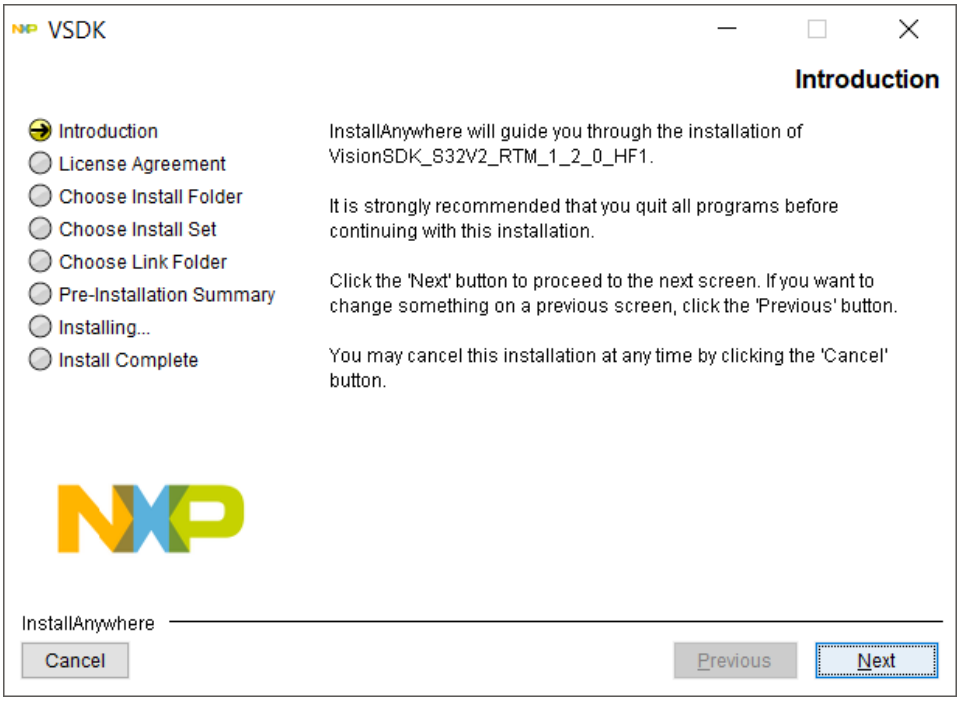
All the code generated by NXP Vision Toolbox is based on S32V234 Vision SDK package. This software package is also free of charge and apart of optimized kernels and libraries for the S32V automotive vision processors, it also contains the build tools to cross-compile the MATLAB generated code to ARM A53 and APEX cores.

You can obtain the S32V234 Vision SDK free of charge directly from NXP website. Perform the following steps to obtain and install the S32V234 Vision SDK and NXP Build Tools:

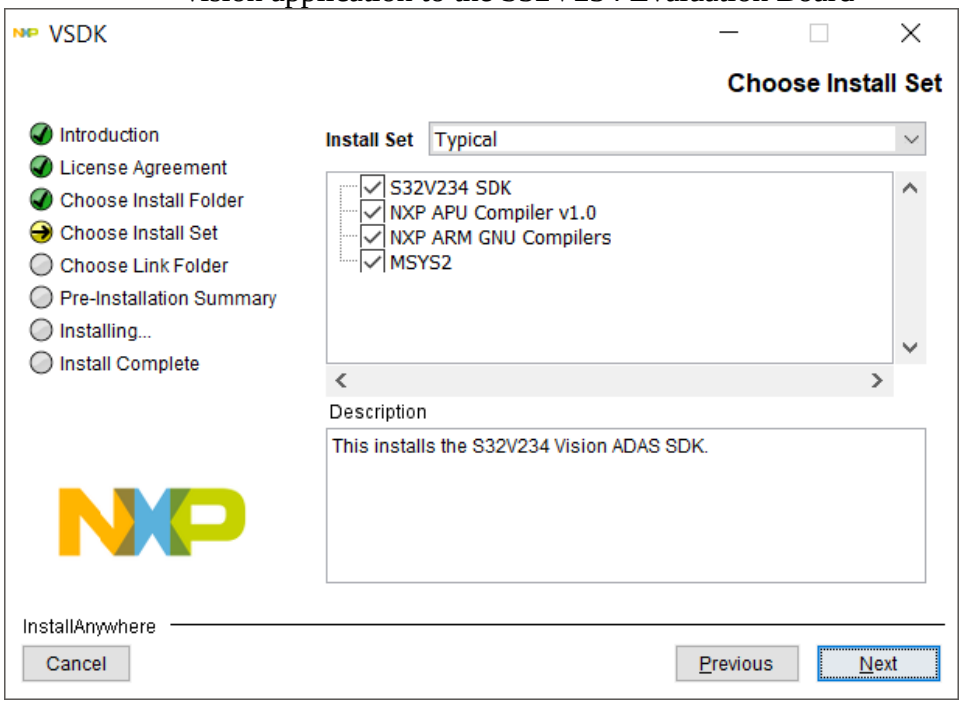
1. Download the Vision SDK RTM v1.2.0 (with all HotFixes) on your PC. Due the size of the package this might take a while.



2. Open the exe file and wait for the Vision SDK Install Anywhere to start.



3. Make sure you follow all the steps and install the:
    - NXP APU Compiler v1.0 – used to compile the generated code for APEX Vision Accelerator
    - NXP ARM GNU Compilers – used to compile the generated code for ARM A53
    - MSYS2 – used to configure the bootable Linux image and to download the actual vision application to the S32V234 Evaluation Board

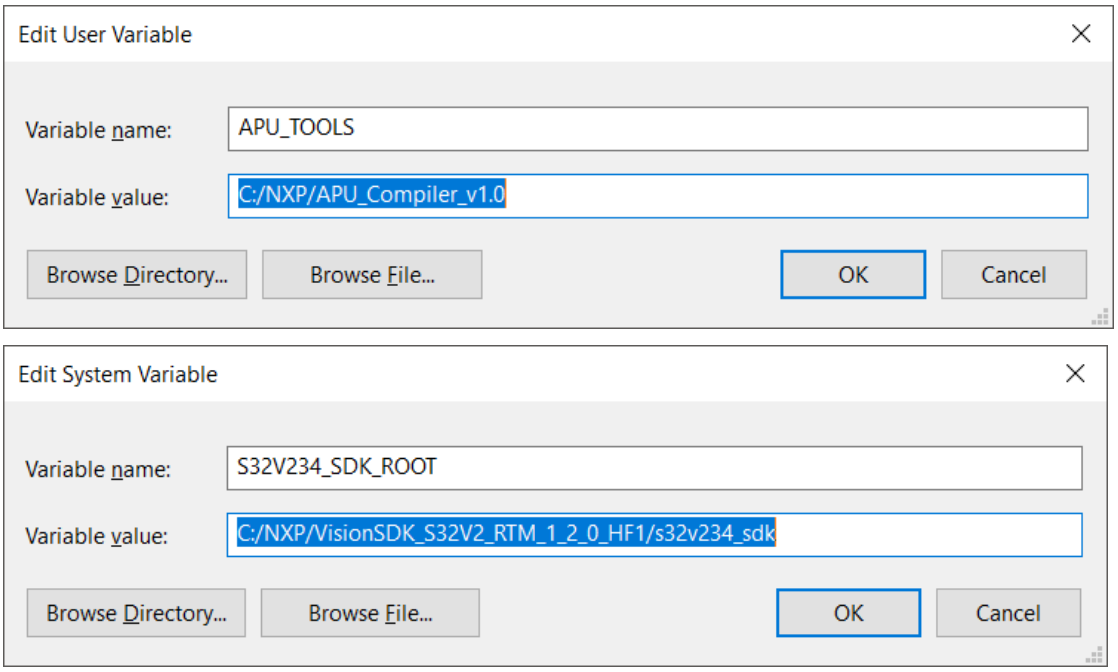


2.6.5 Setting up the Environment

The last step required for software configuration is to set two system or user environmental variables APU_TOOLS and S32V234_SDK_ROOT that points to:

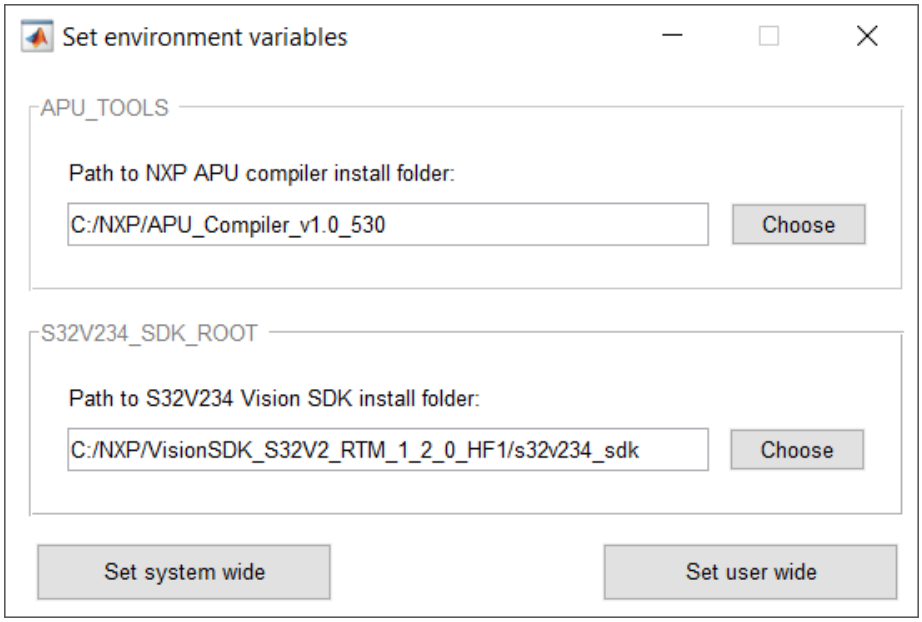
```
APU_TOOLS= C:/NXP/APU_Compiler_v1.0
S32V234_SDK_ROOT = C:/NXP/VisionSDK_S32V2xx_RTM_1_2_0_HF1/s32v234_sdk
```

Ensure system or user environment variables, corresponding to the compiler(s) you have installed, are defined to compiler path value as shown below:



Note: Paths shown are for illustration, your installation path may be different. Once environmental variables are setup you will need to restart MATLAB to use these variables.

An alternative for settings the system paths manually is the “Set the environment variables” option from the NXP Vision Toolbox support package installer:



Note: If the MATLAB is open with Administrator rights, then the "Set system wide" can be used to set the system variables. Othervise (most of the cases) use "Set user wide" to setup the environment variables.

2.6.6 Setting the MATLAB Path

By default, the MATLAB environment is configured during the NXP Vision Toolbox Add-On installation process. In special cases (for other MATLAB installations, Add-on Management, Restore to default Paths) the NXP Vision Toolbox might need to be re-added to the MATLAB path.

In case you need to add the toolbox to the MATLAB path, navigate to the Vision Toolbox installation directory and run the "`nxpvt_install_toolbox`" script.

```
>> nxpvt_install_toolbox
NXP Vision Toolbox: (c) 2018 NXP https://www.nxp.com/visiontoolbox
Successful.
>>
```

2.7 Connecting to the board

In order to provide an easy-to-use interface, the NXP Vision Toolbox supports a direct connection to an S32V234-SBC/S32V234-EVB board which gives the user a way to effortlessly interact with the board. The connection object only needs the IP address of the board:

```
>> s32Obj = nxpvt.s32v234('134.27.168.171')
s32v234.elf      |    4 kB |    4.0 kB/s | ETA: 00:06:35 |   0%
s32v234.elf      | 1586 kB | 1586.6 kB/s | ETA: 00:00:00 | 100%

s32Obj =

  s32v234 with properties:

      CONNECTED: 1
    NOTCONNECTED: 0
         status: 1
      ipAddress: '134.27.168.171'
           port: 9898
     connection: [1×1 tcpclient]
    cameraInUse: 0
```

After the creation of the object there are a series of commands that can be issued to complete different tasks and operations as described below.

Syntax:

s32Obj.shell() - opens a shell in the Matlab Command Window tothe s32v234 board

s32Obj.system(command) - runs the command on the s32v234 connected board

s32Obj.getFile(remoteFilename, localFilename) - copies the remote file from the s32v234 connected board

s32Obj.putFile(localFilename, remoteFilename) - copies the local file to the s32v234 connected board

s32Obj.disconnect() - disconnects from the s32v234 connected board

Examples of usage:

```
s32Obj = nxpvt.s32v234('192.168.1.1');

s32Obj.system('ls -l');

s32Obj.getFile('/a.out', 'C:\a.out'); - copies /a.out from the
s32v234 connected board to the local file

s32Obj.putFile('C:\a.out",'/a.out'); - copies C:\a.out to the
s32v234 connected board remote file
```

s32.disconnect() – disconnects from the board.

2.7.1 Using the MIPI-CSI attached camera

The NXP Vision Toolbox contains a way to get the input from the MIPI-CSI cameras attached to either one of the S32V234-EVB or S32V234-SBC boards, directly in MATLAB. The way to do that is by creating a connection object and a cameraboard object on top of it. Then you can simply get a stream or a single image which can then be handled as a normal MATLAB image. The syntax for doing this is straightforward.

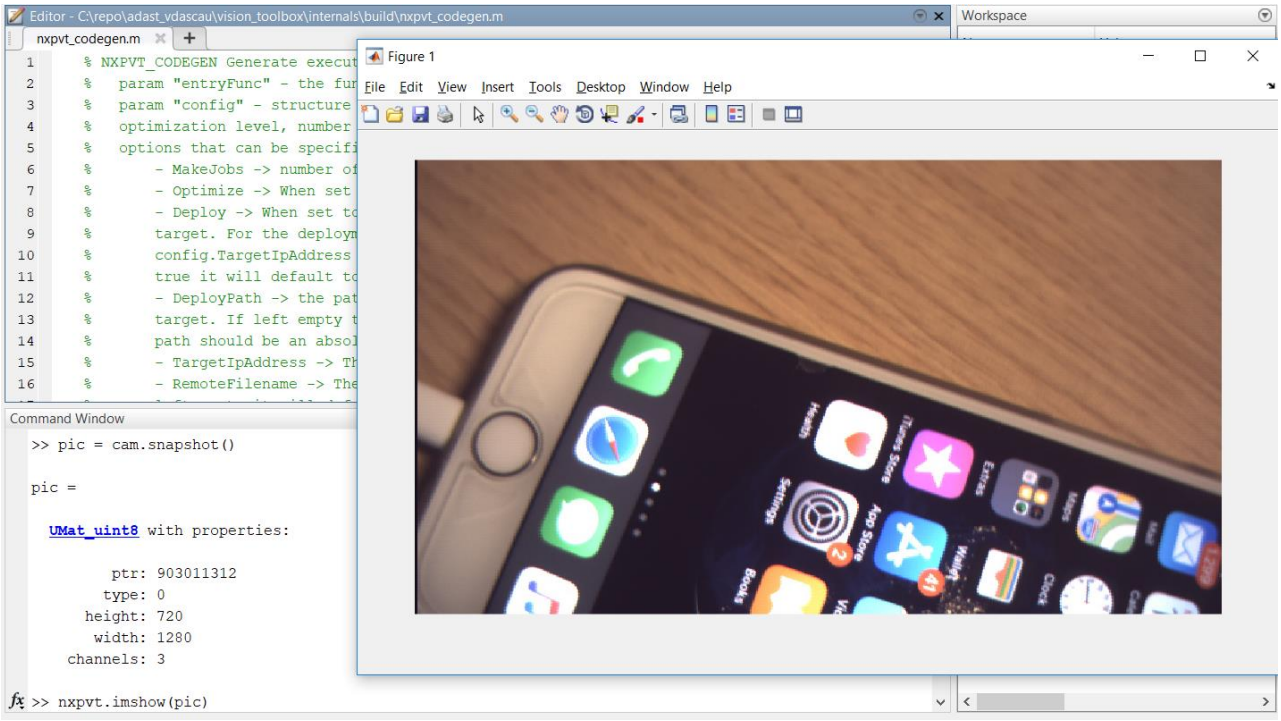
To create the cameraboard object the following syntax should be used:

*camObj = **nxpvt.cameraboard**(s32, cameraIndex, 'Resolution', supportedResolution)*

```
>> cam = nxpvt.cameraboard(s32Obj, 1, 'Resolution', '720x1280')

cam =

  cameraboard with properties:

        height: 720
         width: 1280
     cameraIdx: 0
    s32v234obj: [1×1 nxpvt.s32v234]
```

The *cameraIndex* parameter should be either 1 (MIPI-CSI A port) or 2 (MIPI-CSI B port) depending on the MIPI-CSI port used. The only supported resolution at this moment is '720x1280'. Also, at the moment, we are supporting just one camera at a time. After creating the cameraboard object, it can be used in the following way:

```
>> pic = cam.snapshot()

pic =

  UMat_uint8 with properties:

         ptr: 903011312
        type: 0
      height: 720
       width: 1280
    channels: 3

>> nxpvt.imshow(pic)
```

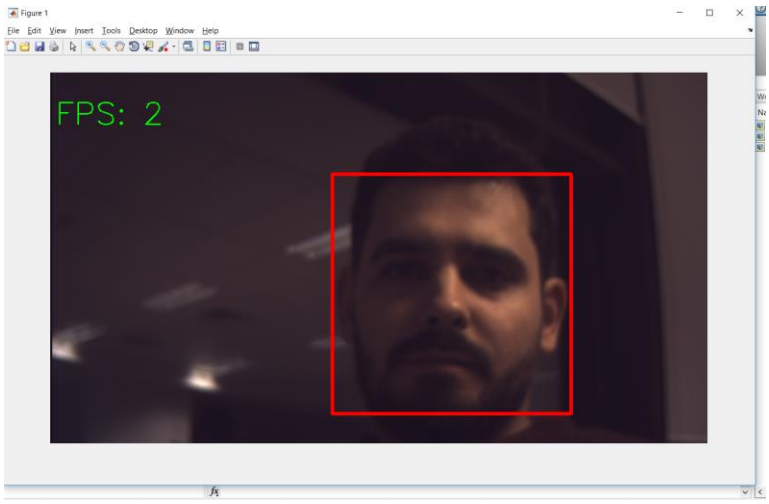


The cameraboard object also supports streaming from the camera. This can be achieved by running the following command:

```
>> cam.stream()
```

The NXP Vision Toolbox also contains an example of face recognition using the onboard camera in MATLAB. This can be found in the '/examples/apps/face_detection' folder. To run you need to specify the board's IP address as an input:

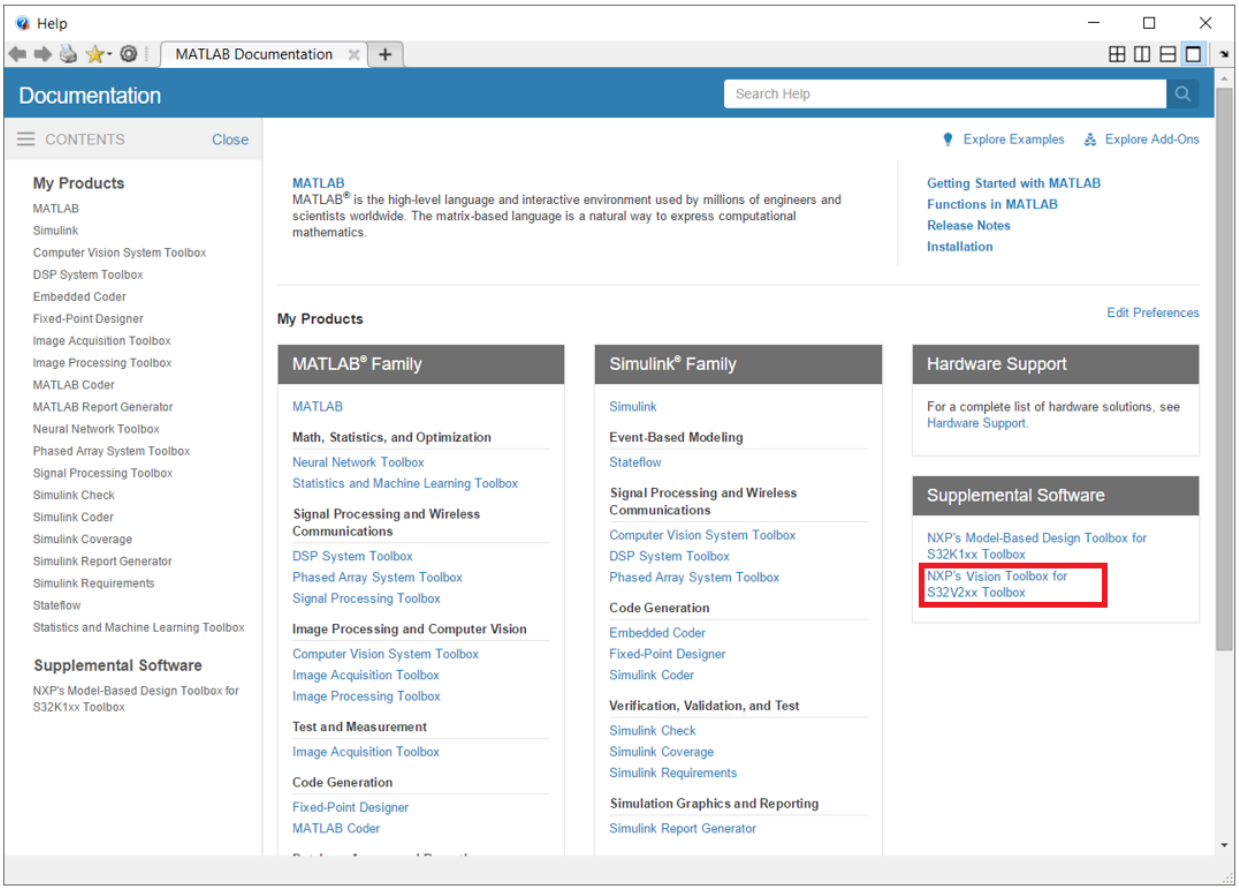
```
>> face_detection_s32v234_camera_main('134.27.168.171')

s32v234.elf                 | 4 kB |    4.0 kB/s | ETA: 00:06:35 |   0%
s32v234.elf               | 1586 kB | 1586.6 kB/s | ETA: 00:00:00 | 100%
[1] FPS: 1, Faces detected: 1,
[2] FPS: 2, Faces detected: 0,
```



2.8 Examples

The NXP Vision Toolbox includes many demonstration models showing many different uses of the kernel functions. To access these examples, go to “examples” folder at your Vision Toolbox install path.

NXP’s Vision Toolbox comes with an Examples Library that let you test and run multiple applications. To open the library, go to `MATLAB Help` (or simply press F1) and select the `NXP Vision Toolbox for S32V234` Supplemental Software link as shown below:



The S32V234 Examples Library represents a collection of MATLAB models that let you test and run complex applications in simulation and on real hardware.

There are four groups of examples that highlight four different types of functionalities supported by NXP Vision Toolbox for S32V234:

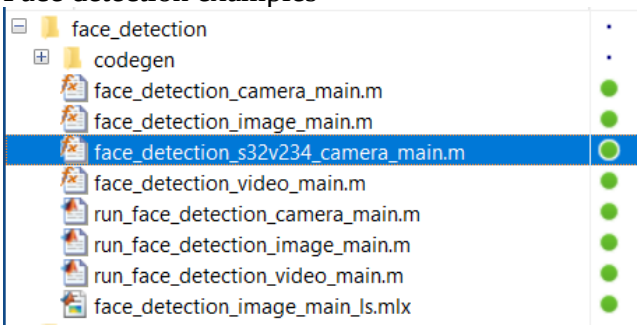
- Vision Applications;
- APEX Kernels;
- APEX Computer Vision Examples;
- S32V234 IO Examples;

2.8.1 Applications

The toolbox contains a series of application examples in the ‘examples/apps’ folder. All application examples can be ran using the simple run_*.m provided in each application folder. The only prerequisite to using these examples is the setting of the global TARGET_IP_ADDRESS variable to the IP address of the board.

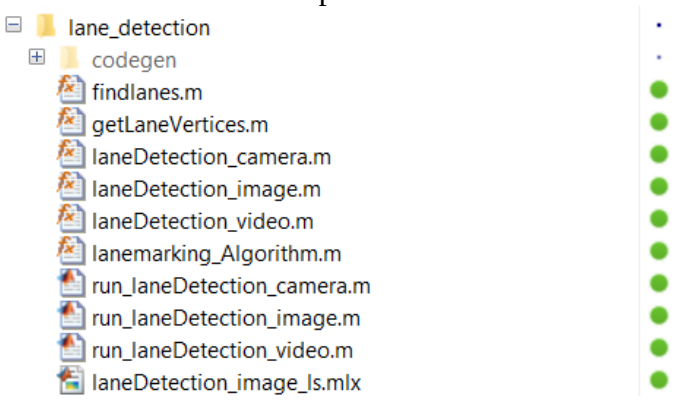
```
>> global TARGET_IP_ADDRESS
>> TARGET_IP_ADDRESS= '134.27.168.171'

TARGET_IP_ADDRESS =

    '134.27.168.171'
```

- Face detection examples

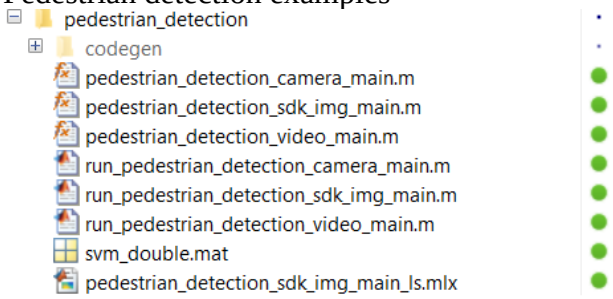


After setting the global variable mentioned above, all of the run scripts will deploy the application onto the board. The only notable running method exceptions are the .m scripts that require continuous communication with the S32V234-EVB / S32V234-SBC hardware boards, those being ‘face_detection_s32v234_camera_main.m’ described above and the ‘examples/io/ s32v234_camera_main.m’ which should get the IP address as an input parameter.

- Lane detection examples



- Pedestrian detection examples



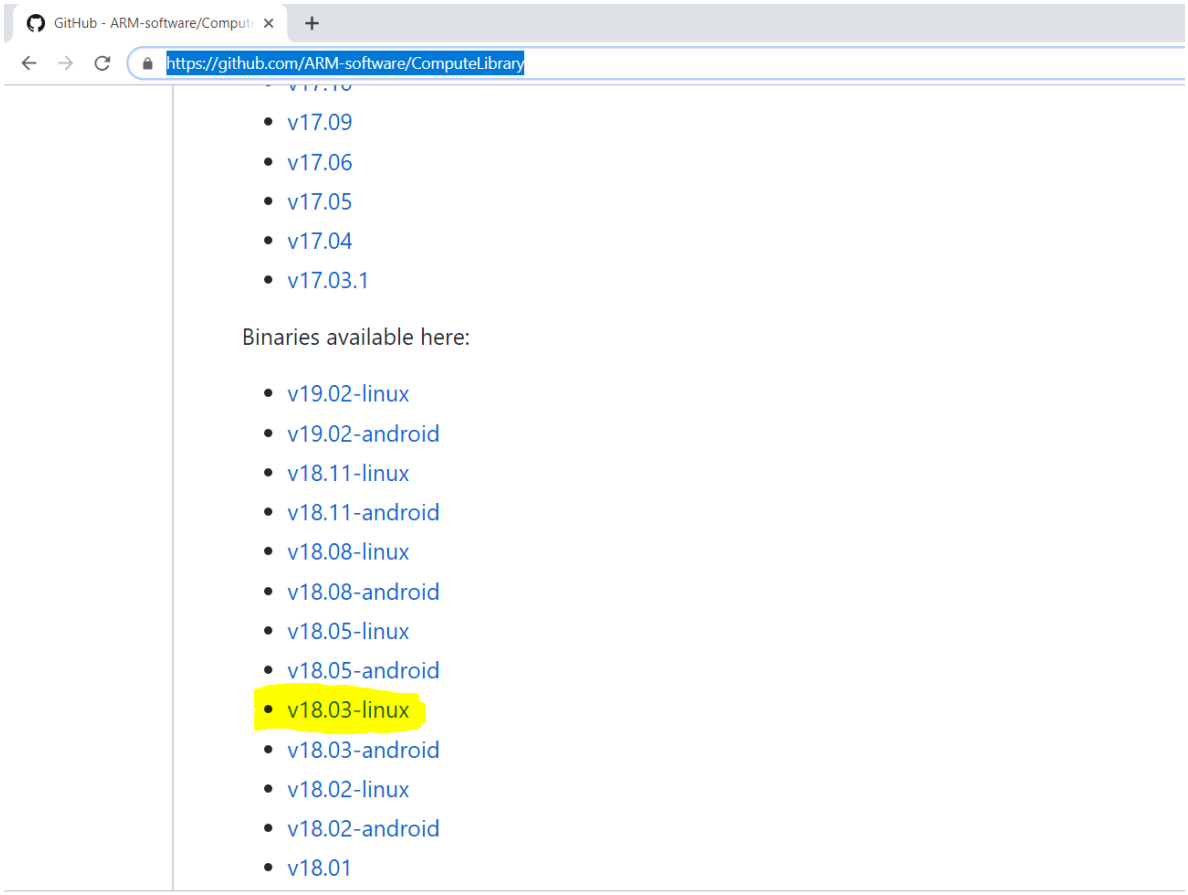
2.8.2 Convolutional Neural Networks

MATLAB provides a series of pretrained neural networks with its Deep Learning Toolbox™. In turn the NXP Vision Toolbox also provides functionality to deploy these networks to the S32V234 board. The user can simply use the provided wrappers to classify objects using one of the pretrained networks or using a custom MATLAB network. To be able to do this, the arm-compute library should be installed on the computer and the ARM_COMPUTELIB environmental variable should be set to point to the root of this installation. To showcase the ease-of-use of this procedure, the toolbox provides a few examples using AlexNet, GoogLeNet and SqueezeNet. These examples classify objects in an image and using frames taken from the MIPI-CSI attached.

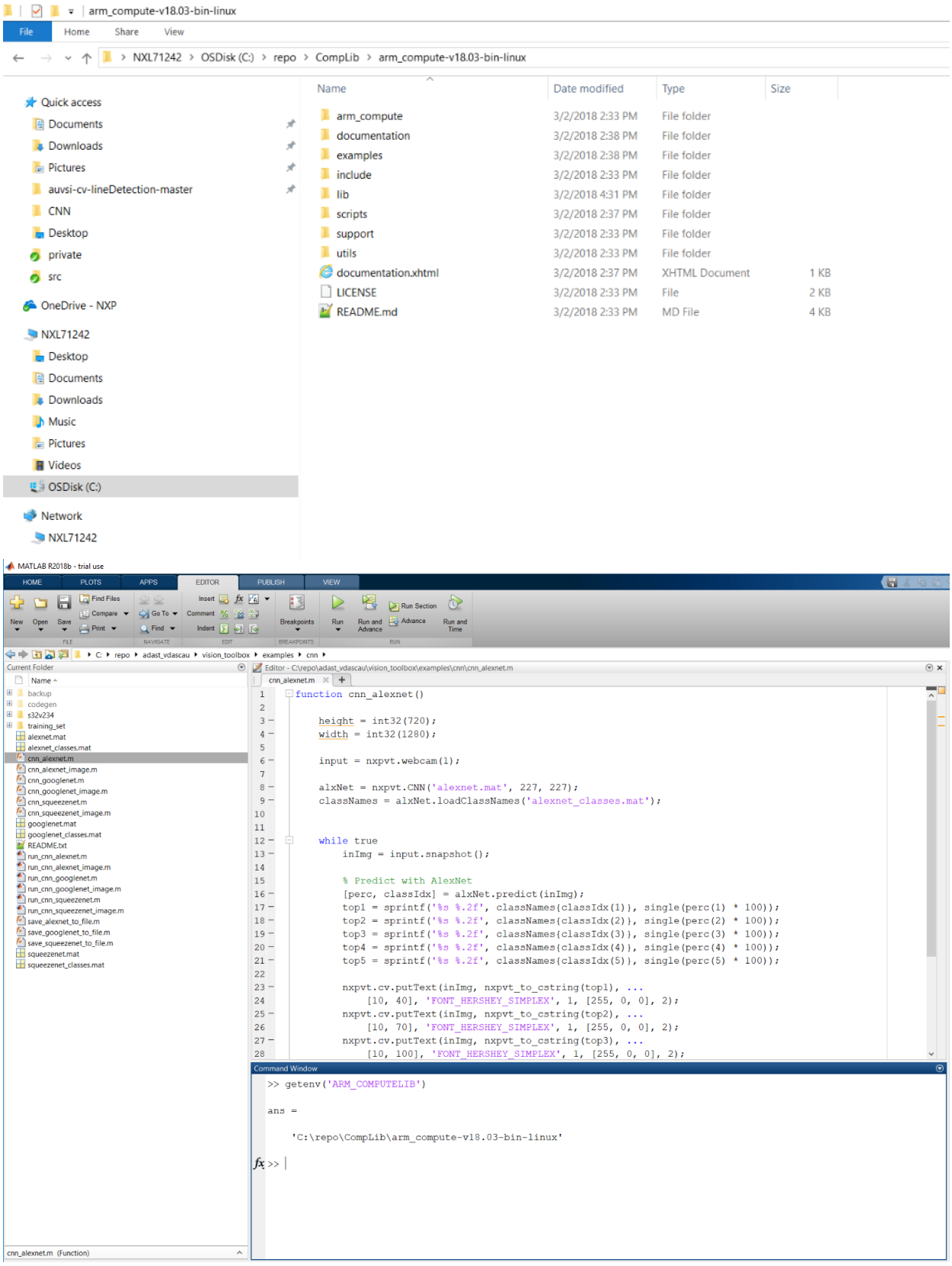
Requirements:

- Deep Learning Toolbox™
- Deep Learning Toolbox™ Model for GoogLeNet Network
- Deep Learning Toolbox™ Model for AlexNet Network.
- Deep Learning Toolbox™ Model for SqueezeNet Network.
- MATLAB Coder Interface for Deep Learning Libraries
- arm_compute library (v18.03)
- ARM_COMPUTELIB environmental variable set to point to the arm_compute library

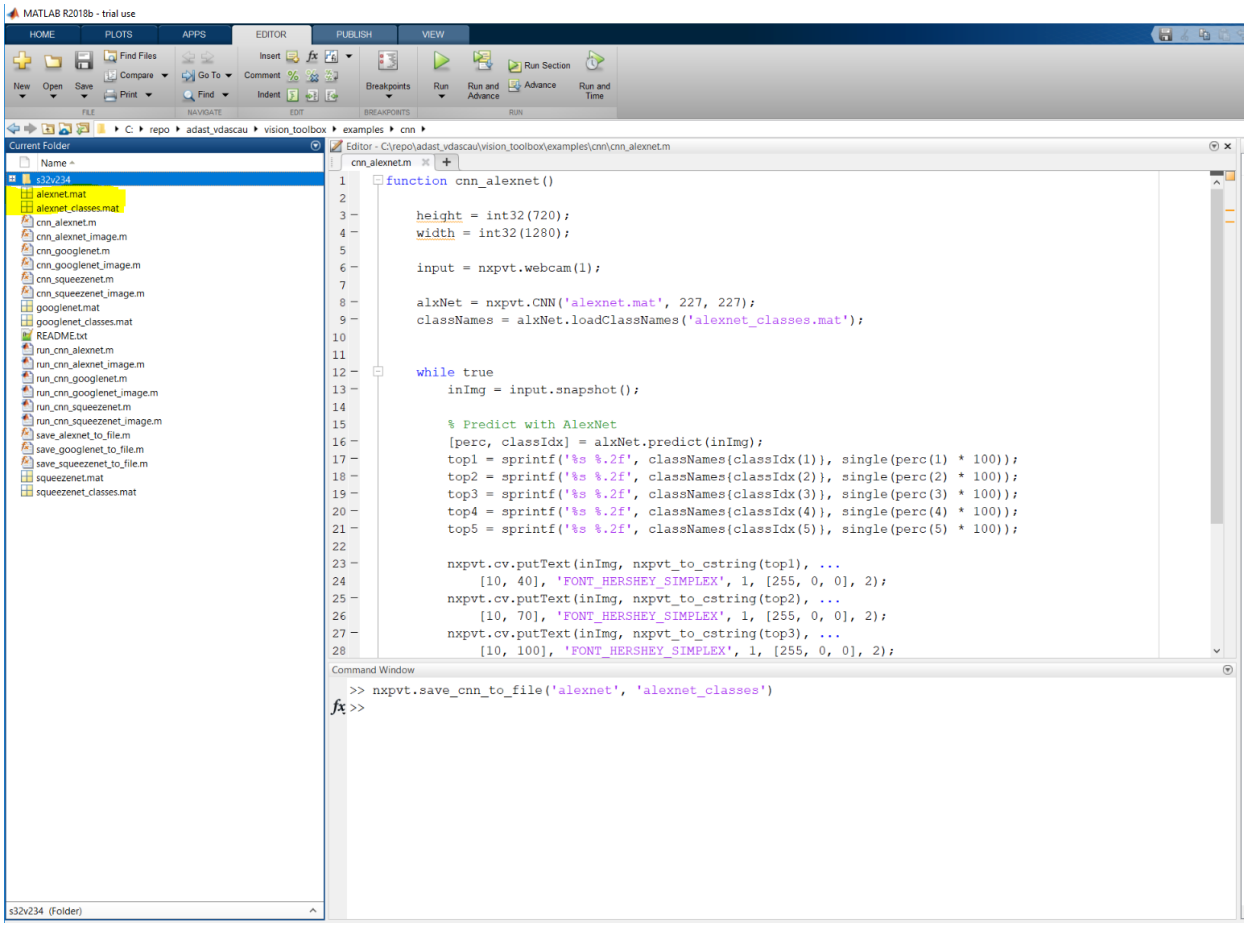
To download arm_compute library, one should download it from https://github.com/ARM-software/ComputeLibrary . The version that was used for running these examples is v18.03:



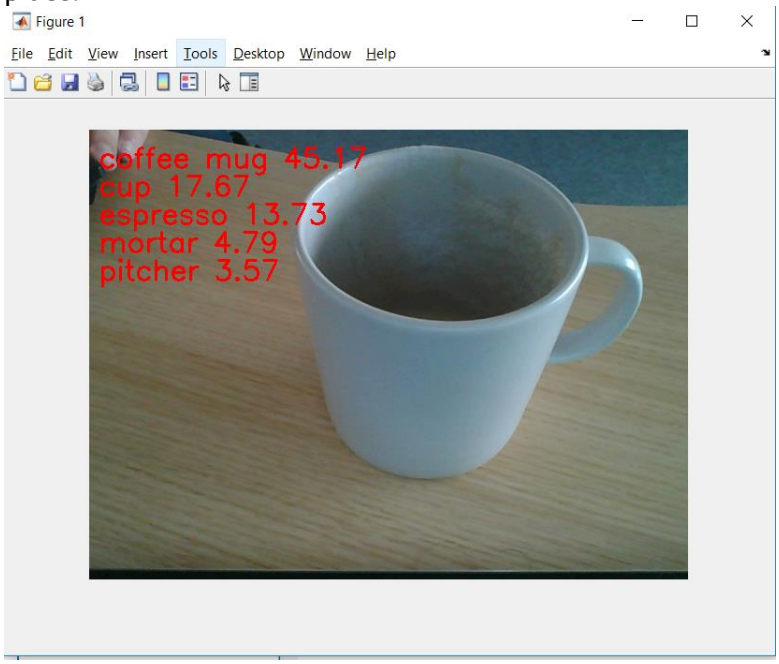
Set the ARM_COMPUTELIB variable to point to the installation folder:



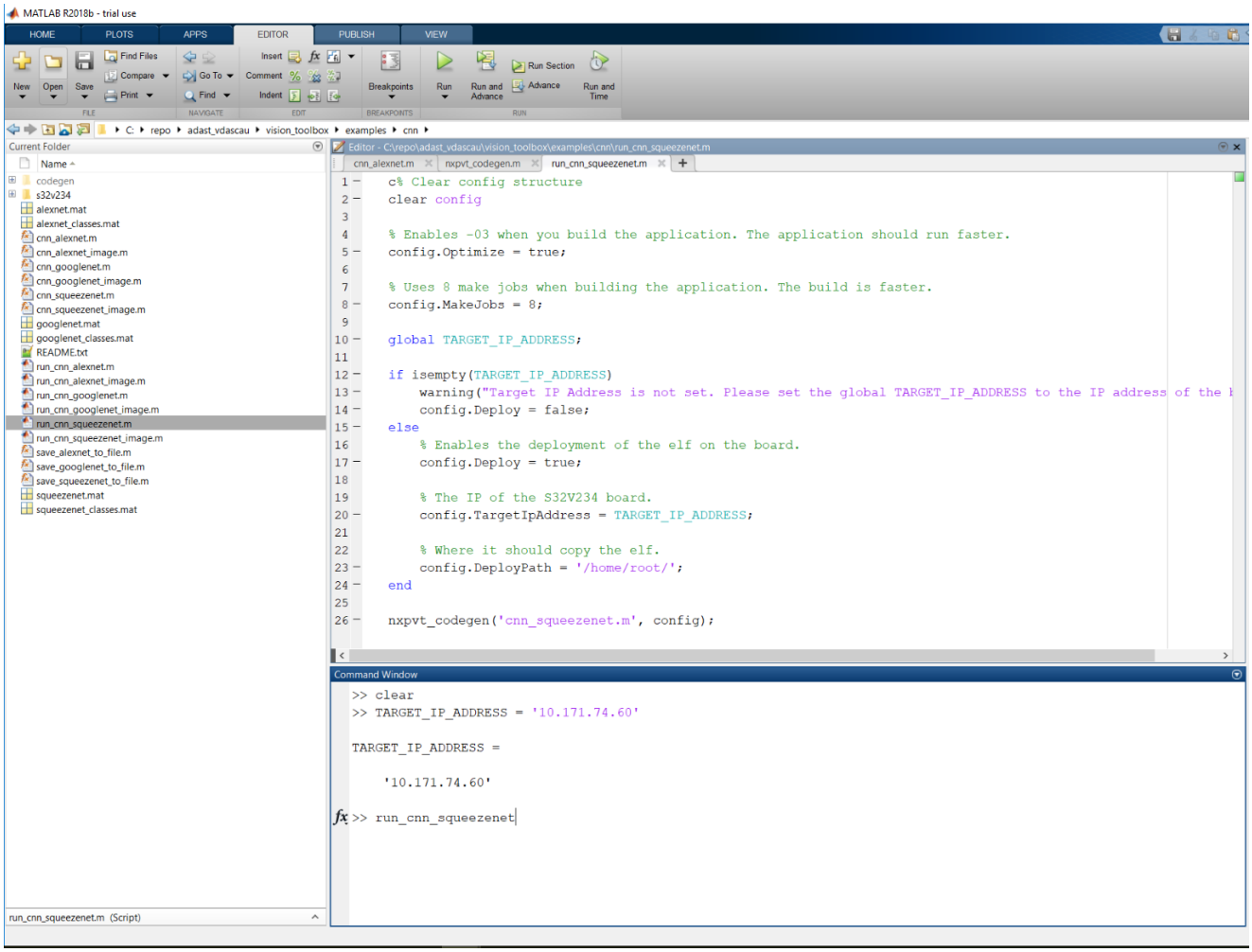
Running the examples in simulation is straightforward and one should be able to do it out of the box, provided that the correct MATLAB toolboxes are installed. However, deploying a network on the hardware is a 2-step procedure. In order to do that, the network and the associated class names should be first saved as .mat files:



Simulation uses the webcam to acquire the video frames on which the classification is taking place:



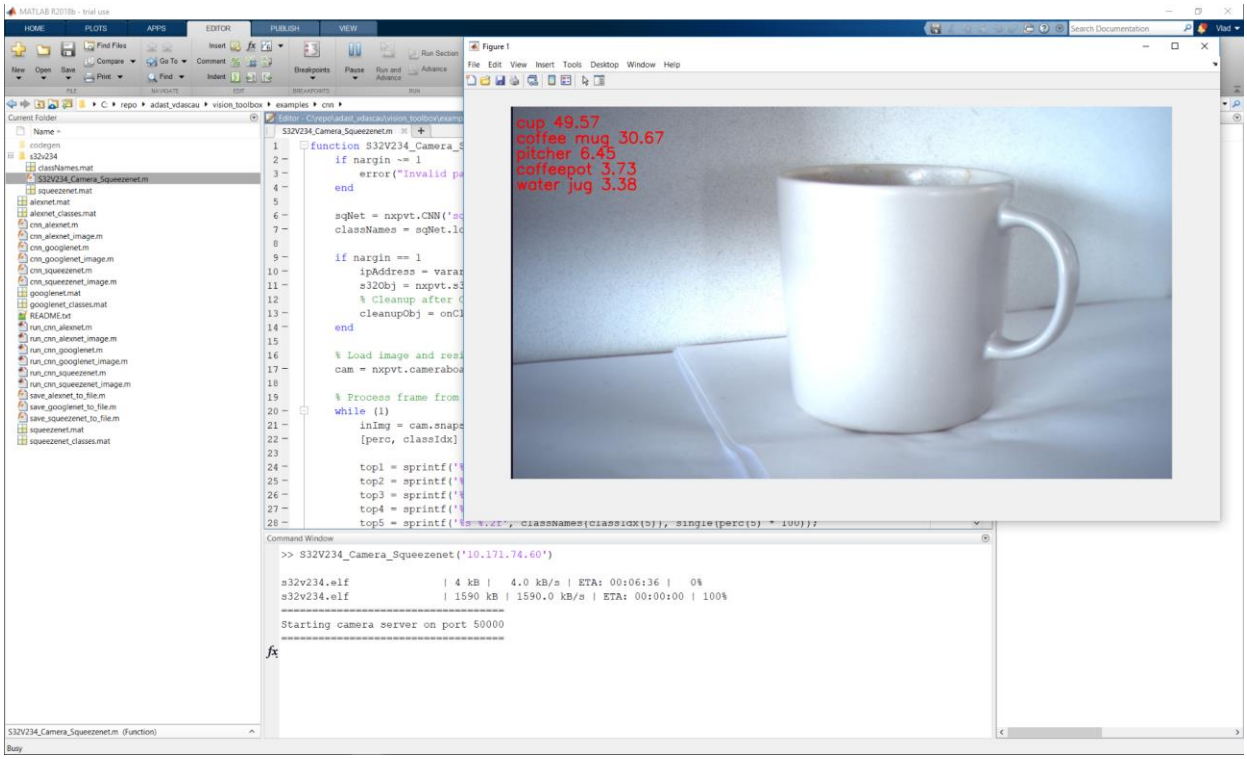
Deploying the CNN to the board is done using the same nxpvt_codegen command as for the regular scripts or by using the **run_[cnn_name].m** script after configuring the TARGET_IP_ADDRESS global variable.



There is also the possibility of using the S32V234 camera to classify the images in MATLAB using the script inside the s32v234 folder. This way the classification algorithm and the neural network will be ran in MATLAB and the images will be taken from the MIPI-CSI S32V234 attached camera.

Usage:

```
>> S32V234_Camera_Squeezenet('10.171.74.60')

s32v234.elf          | 4 kB |   4.0 kB/s | ETA: 00:06:36 |   0%
s32v234.elf          | 1590 kB | 1590.0 kB/s | ETA: 00:00:00 | 100%
====================================
Starting camera server on port 50000
====================================
```



3 Kernels

This chapter describes the main

This section is meant as a starting point for the implementation of computing kernels for the S32V234 using the NXP Vision Toolbox.

The NXP Vision Toolbox for S32V234 kernels organization follows the Vision SDK implementation and supports several component libraries:

- Arithmetic kernels - provide basic operators for element-wise addition, subtraction, multiplication, division and arithmetic shifting
- Comparison kernels - provide basic element-wise comparison operators like less than, less-than-or-equal, binary AND operator and binary descriptor matches
- Conversion kernels - support conversion from 16 to 8 bit and from RGB format to grayscale
- Display kernels - provide examples of marking an image at certain points as overlay or in a certain color channel.
- Feature detection - provides two corner detection algorithms FAST9 and Harris corner detection
- Filtering - offers kernels for general purpose filtering, and also the most used filters like Gaussian filtering, gradient computation, non-maximum suppression and saturation
- Geometry - provides geometric transformations, like rotations and bilinear interpolation and also a replacement for indirect inputs, called offset selection
- Morphology - example of a morphological dilation operator.

- Object detection - two object detection algorithms: Haar cascade and LBP (local binary pattern) cascade
- Optimization - implementation of the Integral Image (SAT) kernel and a SAT-based box filter.
- Resizing - provides downsampling and upsampling kernels (gives examples of size changes inside a filter)
- Statistics - provides kernels for statistics computations, such as a Histogram kernel, a vector-to-scalar reduction kernel and an accumulation kernel.

To use any of these kernels in the MATLAB m-script functions use:

```
nxpvt.apu.<kernel_name>(args1, …)
```

3.1 Arithmetic

3.1.1 nxpvt.apu.add

Description : add two values.

Prototype : [outImg, varargout] = *nxpvt.apu.add*(inImg1, inImg2, varargin)

Inputs/Outputs : Inputs: unsigned 8bit, unsigned 8bit
Output: unsigned 16bit

Inputs: signed 16bit, signed 16bit
Output: signed 32bit

Inputs: signed 32bit, signed 32bit
Output: signed 32bit

Inputs: signed 32bit(high) unsigned 32bit(low), signed 32bit(high) unsigned 32bit(low)
Output signed 32bit(high), unsigned 32bit(low)

3.1.2 nxpvt.apu.diff

Description : minus two values.

Prototype : [out, varargout] = *nxpvt.apu.diff*(in1, in2, varargin)

Inputs/Outputs : Inputs: unsigned 8bit, unsigned 8bit
Output: unsigned 16bit

Inputs: signed 16bit, signed 16bit
Output: signed 16bit

Inputs: signed 16bit, signed 16bit
Output: signed 32bit

Inputs: signed 32bit, signed 32bit
Output: signed 32bit

Inputs: signed 32bit(high), unsigned 32bit(low), signed 32bit(high), unsigned 32bit(low)
Outputs: signed 32bit(high), unsigned 32bit(low)

3.1.3 nxpvt.apu.dot_division

Description : divide two values

Prototype : [out] = *nxpvt.apu.dot_division*(in1, in2)

Inputs/Outputs : Input: signed 32bit
Output: signed 32bit

3.1.4 nxpvt.apu.dot_log2

Description : base 2 log

Prototype : [out] = *nxpvt.apu.dot_log2*(in1)

Inputs/Outputs : Input: signed 32bit
Output: unsigned 8bit

3.1.5 nxpvt.apu.dot_lsh1

Description : Multiply by 2

Prototype : [out, varargout] = *nxpvt.apu.dot_lsh1*(in1, varargin)

Inputs/Outputs : Input: signed 32bit
Output: signed 32bit multiple result: nxpvt.apu.lsh1(value)

Input: signed 32bit
Output: signed 64bit multiple result: nxpvt.apu.lsh1(value, 'int64')

3.1.6 nxpvt.apu.dot_mult_scalar

Description : Multiplies pixelwise with scalar. Does not check out of range

Prototype : [out] = *nxpvt.apu.dot_mult_scalar*(in1, scalar)

Inputs/Outputs : Inputs: unsigned 8bit, Scalar signed 32bit
Output: signed 16bit

Inputs: signed 32bit, Scalar signed 32bit
Output: signed 32bit

3.1.7 nxpvt.apu.dot_multiplic

Description : Multiplies pixelwise two images. Doesn't check out of range

Prototype : [out, varargout] = *nxpvt.apu.dot_multiplic*(in1, in2, varargin)

Inputs/Outputs : Inputs: signed 16bit
Output: signed 32bit

Inputs: signed 32bit
Outputs: signed 32bit(High) unsigned 32bit(low)

Inputs: signed 32bit
Output: signed 32bit

Inputs: signed 32bit, signed 16bit
Output: signed 32bit

3.1.8 nxpvt.apu.dot_sqr

Description : Computes pixelwise the square of an input

Prototype : [out, varargout] = *nxpvt.apu.dot_sqr*(in1, varargin)

Inputs/Outputs : Input: signed 16bit
Output: signed 32bit

Input: signed 32bit
Output: unsigned 32bit

Input: signed 32bit
Output: unsigned 64bit

3.1.9 nxpvt.apu.left_shift

Description : Shifts to the left each pixel of an unsigned 16bit image by a scalar shift value

Prototype : out = *nxpvt.apu.left_shift*(in1, scalar)

Inputs/Outputs : Inputs: unsigned 16bit, scalar signed 32bit
Output: signed 16bit

3.1.10 nxpvt.apu.max

Description : Largest element from the both arrays

Prototype : out = *nxpvt.apu.max*(in1, in2)

Inputs/Outputs : Inputs: unsigned 8bit
Output: unsigned 8bit

3.1.11 nxpvt.apu.right_shift

Description : Shifts to the right each pixel of a signed 64bit image by a scalar shift value

Prototype : [out, varargout] = *nxpvt.apu.right_shift*(in1, in2, shiftFact, varargin)

Inputs/Outputs : Inputs: signed 32bit(Hight), unsigned 32bit(Low)
Outputs: signed 32bit(High), unsigned 32bit(Low)

Inputs: signed 32bit(Hight), unsigned 32bit(Low)
Outputs: signed 32bit : nxpvt.apu.right_shift(in1, in2, shiftFact) or nxpvt.apu.right_shift(in1, in2, shiftFact, 'int32')

Notes: shiftFact internally typecasts into the uint8 value and checks on 0.

3.2 Comparison

3.2.1 nxpvt.apu.abslower

Description : Compares pixelwise two images. Outputs unsigned 8bit comparison result.

Prototype : outImg = *nxpvt.apu.abslower*(inImg1, inImg2)

Inputs/Outputs : param "inImg1" - Input image.
param "inImg2" - Input image or scalar value.
param "outImg" - Output image.

3.2.2 nxpvt.apu.and

Description : Pixelwise "AND" operator between two images. Outputs unsigned 16bit comparison result. Is true if (INPUTA != 0) && (INPUTB != 0)

Prototype : outImg = *nxpvt.apu.and*(inImg1, inImg2, varargin)

Inputs/Outputs : param "inImg1" - Input image.
param "inImg2" - Input image.
param "outImg" - Output image.

3.2.3 nxpvt.apu.lower

Description : Compares pixelwise two images. Outputs unsigned 8bit comparison result. Is true if inImg1 < inImg2

Prototype : outImg = *nxpvt.apu.lower*(inImg1, inImg2, varargin)

Inputs/Outputs : param "inImg1" - Input image.
param "inImg2" - Input image.
param "outImg" - Output image.

3.2.4 nxpvt.apu.lowerequal

Description : Compares pixelwise two images. Outputs unsigned 8bit comparison result. Is true if inImg1 <= inImg2

Prototype : outImg = *nxpvt.apu.lowerequal*(inImg1, inImg2)

Inputs/Outputs : param "inImg1" - Input image.
param "inImg2" - Input image.
param "outImg" - Output image.

3.2.5 nxpvt.apu.match

Description : Matches binary descriptors from group A to binary descriptors from group B. Matches with hamming distance greater than provided threshold is rejected.

Prototype : [matchA, matchB] = *nxpvt.apu.match*(binDataA, binDataB, config)

Inputs/Outputs : param "binDataA" - Input array A of binary descriptors
param "binDataB" - Input array B of binary descriptors
param "config" - Input Configuration:
number of descriptors in binDataA (signed 16-bit)
number of descriptors in binDataB (signed 16-bit)
uint8 - matching threshold (max Hamming distance) (unsigned 8-bit)
uint8 - range check (min Hamming distance between the closest and the second closest descriptors found) (unsigned 8-bit)
param "matchA" - Output First elements of match pairs array. Must be preassigned array of 512 values;
param "matchB" - Output Second elements of match pairs array. Must be preassigned array of 512 values;

3.3 Conversion

3.3.1 nxpvt.apu.low16_to_8

Description : Extracts lower parts of the 16-bit image pixels. Extracts lower parts of the 16-bit image pixels into 8-bit image.

Prototype : outImg = *nxpvt.apu.16low_to_8*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.3.2 nxpvt.apu.rgb_to_grayscale

Description : Convert RGB image to grayscale. Converts the truecolor image to the grayscale intensity image.

Prototype : outImg = *nxpvt.apu.rgb_to_grayscale*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.4 Definitions

The following functions are available to be modified.

3.4.1 nxpvt.apu.accumulation_defs

Definitions for nxpvt.apu.accumulation_defs.
return "chunkWidth" - Chunk width.
return "chunkHeight" - Chunk height.

3.4.2 nxpvt.apu.col_defs

Definitions for nxpvt.apu.col_filter.
return "filterCols" - The number of columns of the column filter.
return "filterQ" - The number of fractional bits for the fixed point coefficients.

3.4.3 nxpvt.apu.cu_defs

Definitions for CU count.
return "nCU" - CU count.

3.4.4 nxpvt.apu.harris_defs

Definitions for nxpvt.apu.harris.
return "window" - window size.

3.4.5 nxpvt.apu.histogram_defs

Definitions for nxpvt.apu.histogram.
return "chunkWidth" - Chunk width.

3.4.6 nxpvt.apu.lbp_defs

Definitions for nxpvt.apu.lbp_defs
return "lbp_window" - Size of window used by LBP
return "chunkW" - Chunk width

3.4.7 nxpvt.apu.match_defs

Definitions for nxpvt.apu.match.
return "chunkX" - Chunk width.
return "chunkY" - Chunk height.
return "cuCounts" - Number of Computation Units or APEX CU.
return "matches" - Number of matches.

3.4.8 nxpvt.apu.rotate_180_defs

Definitions for nxpvt.apu.rotate_180.
return "chunkWidth" - Chunk width.
return "chunkHeight" - Chunk height.

3.4.9 nxpvt.apu.row_defs

Definitions for nxpvt.apu.row_filter.
return "filterRows" - The number of rows of the row filter.
return "filterQ" - The number of fractional bits for the fixed point coefficients.

3.4.10 nxpvt.apu.sat_box_filter_defs

Defines constant for the nxpvt.apu.sat_box_filter function.

3.5 Display

3.5.1 nxpvt.apu.mark

Description : Marking with greyscale

Prototype : [outImg] = *nxpvt.apu.mark*(imgIn, imgMap)

Inputs/Outputs : param “imgIn” - greyscale image
param “imgMap” - markers map
param “outImg” - output image

3.5.2 nxpvt.apu.mark_color

Description : Marking with color on image

Prototype : [outImg] = *nxpvt.apu.mark_color*(imgIn, imgMap, channel)

Inputs/Outputs : param “imgIn” - color image
param “imgMap” - markers map, channel - color
param “channel” - to mark (0-Red 1-Green 2-Blue)
param “outImg” - output image

3.6 Feature Detection

3.6.1 nxpvt.apu.fast9

Description : FAST9 feature point detection. Finds the corners in the input data using the FAST9 algorithm. Outputs corner scores or 0 if not a corner. For each input pixel a 16-pixel circle centered at the processed pixel is considered. The circle pixels are classified as darker, brighter or similar to the central pixel depending on the provided threshold. The central pixel is considered as a corner if and only if there is a contiguous segment of 9 pixels which are all classified as brighter or darker in the circle.
See http://www.edwardrosten.com/work/fast.htm

Prototype : [outImg] = *nxpvt.apu.fast9*(inImg, threshold)

Inputs/Outputs : param "inImg" - Input image.
param "threshold" - Threshold used for classifying ring pixels (brighter/darker/similar).
param "outImg" - Output image.

3.6.2 nxpvt.apu.harris

Description : Harris Corner Detector. Finds the corners in the input data using the Harris algorithm. Outputs a Harris response value for each pixel.

Prototype : outResponse = *nxpvt.apu.harris*(inGrX, inGrY, inKRbsWin)

Inputs/Outputs : param "inGrX" - Image gradient X component (int16).
param "inGrY" - Image gradient Y component (int16).
param "inKRbsWin" - Harris detector free parameter (uint16).
- Harris detector window size (uint16).
- Bit shift for output response (uint16).
param "outResponse" - output Harris response image (uint16).

3.6.3 nxpvt.apu.sad

Description : Sum of Absolute Differences. Calculate minimum SAD & location given a 4x4 template in an 8x8 window.

Prototype : [out] = *nxpvt.apu.sad*(inImg1, inImg2)

Inputs/Outputs : param "inImg1" - First input image (template 4x4 uint8).
param "inImg2" - Second input image (window 8x8 uint8).
param "out" - Minimum SAD & location given:
0 - Low byte Minimum SAD
1 - High byte Minimum SAD
2 - Location X
3 - Location Y

3.7 Filtering

3.7.1 CorrelationSize

Description : Filter size to select correlation type.

3.7.2 nxpvt.apu.col_filter

Description : 1 dimensional column filter. A column filter is a 1-dimension filter applied to an image where each pixel becomes a weighted sum of itself and neighboring pixels in the same row. The weighted sum is determined by a set of filter coefficients. The filter are pixel-centered. The filter has 3 columns and a single row of coefficients.

Prototype : [outImg] = *nxpvt.apu.col_filter*(inImg, coeffs)

Inputs/Outputs : param "inImg" - The padded source image.
param "coeffs" - The column filter coefficients.
param "outImg" - The destination image.

3.7.3 nxpvt.apu.correlation

Description : General correlation. Correlation of input image with a filter.

Prototype : outImg = *nxpvt.apu.correlation*(inImg, filterCoefs, scale, corrSize, corrType)

Inputs/Outputs : param "inImg" - Input image.
param "filterCoefs" - Signed 16 bit filter coefficients.
param "scale" - The scalar value of the normalization factor used for the filter.
param "corrSize" - Filter size to select correlation type.
param "corrType" - Flags for the different possible shapes of a filter.
param "outImg" - Output image, signed 16bit correlation result.

3.7.4 nxpvt.apu.filter_median_3x3

Description : 2-D median filtering. Performs median filtering of the image inImg in two dimensions. Each output pixel contains the median value in the 3-by-3 neighborhood around the corresponding pixel in the input image.

Prototype : outImg = *nxpvt.apu.filter_median_3x3*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.5 nxpvt.apu.filtering_sobel_3x3

Description : Sobel filter. Performs Sobel filtering of the image inImg in two dimensions.

Prototype : outImg = *nxpvt.apu. filtering_sobel_3x3*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.6 nxpvt.apu.gauss_3x3

Description : Blurs an image using a Gaussian filter. Convolves the image with the 3x3 Gaussian kernel

Prototype : outImg = *nxpvt.apu.gauss_3x3*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.7 nxpvt.apu.gauss_5x5

Description : Blurs an image using a Gaussian filter. Convolves the image with the 5x5 Gaussian kernel

Prototype : outImg = *nxpvt.apu.gauss_5x5*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.8 nxpvt.apu.gradient

Description : Directional gradients of an image. Returns the directional gradients using the Sobel method.

Prototype : [outGx, outGy] = *nxpvt.apu.gradient*(inImg)

Inputs/Outputs : param "inImg" - Input image (8-bit)
param "outGx" - Gradient X component output image (16-bit)
param "outGy" - Gradient Y component output image (16-bit)

3.7.9 nxpvt.apu.gradient_x

Description : Gradient in X direction. Convolution of input unsigned 8bit image with a [-1 0 1] row-filter. Outputs signed 16bit convolution result.

Prototype : outImg = *nxpvt.apu.gradient_x*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.10 nxpvt.apu.gradient_y

Description : Gradient in Y direction. Convolution of input unsigned 8bit image with a [-1 0 1] column-filter. Outputs signed 16bit convolution result.

Prototype : outImg = *nxpvt.apu.gradient_y*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.11 nxpvt.apu.nms

Description : Non-maximum suppression. Sets values which are not maximal in their 3x3 neighborhood (8 pixels) to 0. 8-bit/16-bit version.

Prototype : [outImg] = *nxpvt.apu.nms*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.12 nxpvt.apu.row_filter

Description : 1 dimensional row filter. A row filter is a 1-dimension filter applied to an image where each pixel becomes a weighted sum of itself and neighboring pixels in the same column. The weighted sum is determined by a set of filter coefficients. The filter are pixel-centered. The filter has 5 rows and a single columns of coefficients.

Prototype : [outImg] = *nxpvt.apu.row_filter*(inImg, coeffs)

Inputs/Outputs : param "inImg" - The padded source image.
param "coeffs" - The row filter coefficients.
param "outImg" - The destination image.

3.7.13 nxpvt.apu.saturate_nonzero

Description : Non-zero pixel saturation. Changes non-zero pixel values to maximal values.

Prototype : outImg = *nxpvt.apu.saturate_nonzero*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.14 nxpvt.apu.scharr_x

Description : Gradient in X direction for Scharr_X filter. Convolution of input image with a scharr_x filter. Outputs signed 16bit convolution result.

Prototype : outImg = *nxpvt.apu.scharr_x*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.7.15 nxpvt.apu.scharr_y

Description : Gradient in Y direction for Scharr_Y filter. Convolution of input image with a scharr_y filter. Outputs signed 16bit convolution result.

Prototype : outImg = *nxpvt.apu.scharr_y*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.8 Geometry

3.8.1 nxpvt.apu.rotate_180

Description : Rotates image by 180 degrees. Rotates image inImg by 180 degrees in a counterclockwise direction around its center point.

Prototype : outImg = *nxpvt.apu.rotate_180*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.9 Indirect

3.9.1 nxpvt.apu.indirect

Description : Indirect Inputs. Kernel for Indirect Inputs emulation.

Prototype : outImg = *nxpvt.apu.indirect*(inImg, inOffset)

Inputs/Outputs : param "inImg" - input Image (uint8).
param "inOffset" - offset array (byte offsets relative to the source data region starting point) (uint32).
param "outImg" - output Image (uint8).

3.10 Morphology

3.10.1 nxpvt.apu.dilate_diamond

Description : Diamond dilation. Dilates the image using 5x5 diamond structure element.

Prototype : outImg = *nxpvt.apu.dilate_diamond*(inImg)

Inputs/Outputs : param "inImg" - Input image (uint8).
param "outImg" - Output image (uint8).

3.11 Object Detection

3.11.1 nxpvt.apu.harr_cascade

Description : Detects objects using Haar-like feature cascades. This algorithm searches for 20x20-pixel objects using a Haar-like classifier provided by the user. For each input pixel, it outputs 255 if the pixel is a lower left corner of an object and 0 otherwise.

Prototype : [out] = *nxpvt.apu.haar_cascade*(sat, satSquared, stageSize, features, stages, pixelShift, pixelOffsets)

Inputs/Outputs :

sat	- summed area table	- usigned 32bit
satSquared	- squared summed area table	- unsigned 32bit
cascadeFeatures	- feature structure	
stageSize	- number of cascades	- unsigned 16bit
cascadeStages	- cascade structure	
pixelShifts	- Required for code generation only	- unsigned 8bit array
pixelOffsets	- Required for code generation only	- unsigned 8bit array

3.11.2 nxpvt.apu.lbp_cascade

Description : This function is detects faces uses LBP cascade algorithm. LBP_WINDOWS_SIZE constant defined in the lbp_definitions.m file.

Prototype : outImg = *nxpvt.apu.lbp_cascade*(sat, cascadeSize, cascadeFeatures, cascadeStages, pixelShift, pixelOffsets)

Inputs/Outputs :

sat	- summed area table - uint32
cascadeSize	- [cascade_feature size, cascade_stages size, 0] - uint16
cascadeFeatures	- feature structure
cascadeStages	- cascade structure
pixelShifts	- Required for code generation only - uint8 array
pixelOffsets	- Required for code generation only - uint8 array
imgOut	- uint8

3.12 Optimization

3.12.1 nxpvt.apu.sat

Description : Summed area table.

Prototype : [outImg, varargout] = *nxpvt.apu.sat*(inImg)

Inputs/Outputs : Input unsigned 8bit - Output unsigned 32bit
Input signed 8bit - Output signed 32bit, unsigned 32bit
Input signed 32 bit - Output signed 32bit(high) unsigned 32bit(low)

3.12.2 nxpvt.apu.sat_box_filter

Description : Applies a box filter (== sum over that patch) to the image using its summed area table (integral image). Constants defined in sat_box_filter_definitions function.

Prototype : outImg = *nxpvt.apu.sat_box_filter*(inImg)

Inputs/Outputs : param "inImg" - Input image.
param "outImg" - Output image.

3.13 Resizing

3.13.1 nxpvt.apu.downsample

Description : x2 downsampling. Downsamples the image by two.

Prototype : outImg = *nxpvt.apu.downsample*(inImg)

Inputs/Outputs : param "inImg" - Input image (uint8/uint16).
param "outImg" - Output image (uint8/uint16).

3.13.2 nxpvt.apu.downsample_gauss

Description : x2 downsampling using Gaussian blur. Downsamples the image by two using Gaussian blur.

Prototype : outImg = *nxpvt.apu.downsample_gauss*(inImg)

Inputs/Outputs : param "inImg" - Input image (uint8).
param "outImg" - Output image (uint8).

3.13.3 nxpvt.apu.upsample

Description : x2 upsampling. Upsamples the image by two.

Prototype : outImg = *nxpvt.apu.upsample*(inImg)

Inputs/Outputs : param "inImg" - Input image (uint8).
param "outImg" - Output image (uint8).

3.14 Statistics

3.14.1 nxpvt.apu.accumulation

Description : Accumulates all values in a chunk. Builds the sum of all elements of a chunk and writes out a vector of sum values.

Prototype : outAccum = *nxpvt.apu.accumulation*(inImg, inOffsX, inOffsY, inWidthX, inHeightY)

Inputs/Outputs : param "inImg" - Input image (int32).
param "inOffsX" - X Offset where to start accumulation (int16).
param "inOffsY" - Y Offset where to start accumulation (int16).
param "inWidthX" - Width inside block for which accumulation has to be performed (int16).
param "inHeightY" - Height inside block for which accumulation has to be performed (int16).
param "outAccum" - Output accumulation value (int32).
(inImgWidth/chunkWidth) columns X
(inImgHeight/chunkHeight) rows.

3.14.2 nxpvt.apu.histogram

Description : Histogram implementation for APEX. Histogram computation of an input image.

Prototype : outHist = *nxpvt.apu.histogram*(inImg)

Inputs/Outputs : param "inImg" - Input image (uint8).
param "outHist" - Histogram row vector output, CU count X 256 (uint32).

3.14.3 nxpvt.apu.reduction

Description : Reduction from 256 vectors on each CU to 256 scalars. Reduce an input vector/image by summing up the corresponding elements. ! Use this kernel only with *nxpvt.apu.histogram*.

Prototype : outSclrHist = *nxpvt.apu.reduction*(inVecHist)

Inputs/Outputs : param "inVecHist" - Histogram row vector, CU count X 256 (uint32).
param "outSclrHist" - Histogram scalar, row 1 columns 256 (uint32).

4 Functions

4.1 Code Generation

4.1.1 nxpvt_codegen

Description : Generate executable based on m-script passed as input

Prototype : *nxpvt_codegen*(entryFunc, config)

Inputs/Outputs : param "entryFunc" - the function for which to generate the executable.
param "config" - structure holding various configurations (e.g. optimization level, number of make jobs, deployment on target).

The options that can be specified in a config are:
- MakeJobs -> number of CPU jobs
- Optimize -> When set to true it will be using O3 optimization
- Deploy -> When set to true it deploys the application on to the target. For the deployment to work the user needs to configure config.TargetIpAddress as well. If this is not set explicitly to true it will default to false
- DeployPath -> the path where the executable will be copied on the target. If left empty the '/examples/' folders will be used. This path should be an absolute path.
- TargetIpAddress -> The ip address for the target
- RemoteFilename -> The name of the executable on the target. If left empty it will default to the entryFunc name with the elf extension instead of the .m extension
- ExtraFiles -> Files that are used by the elf (e.g videos,images). If left empty no extra files will be copied on the target. The paths for this file should be relative to the DeployPath.

4.2 Target Configuration

4.2.1 nxpvt_create_target

Description : Creates SD card bootable image.

Prototype : *nxpvt_deploy_on_target*(sourceImg, dstImgWin)

Inputs/Outputs : param "sourceImg" - the SD Card image to be written
param "dstImgWin" - the Windows drive letter for the SD card

4.2.2 nxpvt_deploy_on_target

Description : Copy and run the executable on the target.

Prototype : *nxpvt_deploy_on_target*(entry_name, config)

Inputs/Outputs : param "entry_name" - executable to be copied on the target
param "config" - config structure used in *nxpvt_codegen*

4.3 Toolbox Management

4.3.1 nxpvt_install_toolbox

Description : Install or uninstall the toolbox by setting the MATLAB paths.

Prototype : *nxpvt_install_toolbox*(varargin)

Inputs/Outputs : param "varargin" - if varargin is empty the function will add the toolbox into MATLAB path. If varargin is “remove” the function is going to remove the toolbox from MATLAB path

4.4 Core functionality

4.4.1 UMat

The nxpvt.UMat image container is the data structure used in the whole toolbox to wrap data buffers. This is a virtual data container allowing for manipulating data which can be used by the host ARM core as well as by the hardware accelerators.

4.4.1.1 Object Creation

`dataUMat = nxpvt.UMat(data)` creates an UMat object which holds the data array. Type of data can be: int8, uint8, int16, uint16, int32, uint32, single, and double. Dimension of data array can be maximum 3.

`dataUMatRoi = nxpvt.UMat(dataUMat, [x y w h])` creates an UMat object from a bigger one. A region of interest (often abbreviated ROI), are samples within a data set identified for a particular purpose. The data array of dataUMatRoi is data(y:y+h-1, x:x+w-1).

4.4.1.2 Methods

`data = dataUMat.data()` returns data array.

`m = dataUMat.rows()` returns number of rows.

`n = dataUMat.cols()` returns number of columns.

`k = dataUMat.channels_()` returns number of channels.

`dataUMat.isempty()` checks if empty.

`dataUMat.release()` releases UMat object.

4.5 Classifiers

4.5.1 Cascade object detector

The cascade object detector uses Haar-like features and LBP features to detect people's faces, noses, eyes, and mouth. To detect facial features or faces in an image:

1. Create the nxpvt.CascadeObjectDetector object and set its properties.

2. Call the object with arguments, as if it were a function.

4.5.1.1 Object Creation

`detector = nxpvt.CascadeObjectDetector(model)` creates a detector configured to detect objects defined by the input model name.

`detector = nxpvt.CascadeObjectDetector(XMLFILE)` creates a detector and configures it to use the custom classification model specified with the XMLFILE input.

`detector = nxpvt.CascadeObjectDetector(PropName,PropValue)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example:

```
detector = nxpvt.CascadeObjectDetector('ClassificationModel','FrontalFaceLBP');
```

4.5.1.2 Properties

Properties are non-tunable, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

Classification Model

Classification model can be:

- 'FrontalFaceLBP' - Detects faces that are upright and forward facing. This model is composed of weak classifiers, based on a decision stump. These classifiers use local binary patterns (LBP) to encode facial features. LBP features can provide robustness against variation in illumination. Image size used to train model is `Training_Size = [height width] = [24 24]`

- 'EyePairBig' - Detects a pair of eyes. It was trained on larger image than 'EyePairSmall' model. This model is composed of weak classifiers, based on a decision stump. This classifier uses Haar features to encode details. Image size used to train model is `Training_Size = [height width] = [11 45]`

- 'EyePairSmall' - Detects a pair of eyes. This model is composed of weak classifiers, based on a decision stump. This classifier uses Haar features to encode details. Image size used to train model is `Training_Size = [height width] = [5 11]`

- 'LeftEye' - Detects the left eye separately. This model is composed of weak classifiers, based on a decision stump. This classifier uses Haar features to encode details. Image size used to train the model is `Training_Size = [height width] = [12 18]`

- 'RightEye' - Detects the right eye separately. This model is similar to 'LeftEye' model. Image size used to train the model is `Training_Size = [height width] = [12 18]`

- 'Mouth' - Detects the mouth. This model is composed of weak classifiers, based on a decision stump, which use Haar features to encode mouth details. Image size used to train the model is `Training_Size = [height width] = [15 25]`

- 'Nose' - This model is composed of weak classifiers, based on a decision stump, which use Haar features to encode nose details. Image size used to train the model is `Training_Size = [height width] = [15 18]`

Size of smallest detectable object

Size of smallest detectable object, specified as a two-element vector MinSize = [width height]. Set this property in pixels for the minimum size region containing an object. The value must be greater than or equal to the image size used to train the model. Use this property to reduce computation time when you know the minimum object size prior to processing the image. When you do not specify a value for this property, the detector sets it to the size of the image used to train the classification model.

Size of largest detectable object

Size of largest detectable object, specified as a two-element vector `MaxSize = [width height]`. Specify the size in pixels of the largest object to detect. Use this property to reduce computation time when you know the maximum object size prior to processing the image. When you do not specify a value for this property, the detector sets it to size(I).

Scaling for multiscale object detection

Scaling for multiscale object detection, specified as a value greater than 1.0001. The scale factor incrementally scales the detection resolution between MinSize and MaxSize. The detector scales the search region at increments between MinSize and MaxSize using the following relationship: search_region = round((Training_Size)*(ScaleFactor)^N). N is the current increment, an integer greater than zero, and Training_Size is the image size used to train the classification model.

Skip odd

Skipping the odd rows and columns forces the application to not apply the algorithm if the current row index or the current column index is odd, which means that the search is done only on one quarter of all possibilities. By setting this option a speedup is done, but also the search can miss easily.

Detection threshold

Detection threshold, specified as an integer. The threshold defines the criteria needed to declare a final detection in an area where there are multiple detections around an object. Groups of collocated detections that meet the threshold are merged to produce one bounding box around the target object. Increasing this threshold may help suppress false detections by requiring that the target object be detected multiple times during the multiscale detection phase. When you set this property to 0, all detections are returned without performing thresholding or merging operation.

4.5.1.3 Method step

```
bbox = detector(img)
```

- `img` is the input image, specified as true-color (RGB)
- `bbox` contains the detections, returned as an M-by-4 element matrix. Each row of the output matrix contains a four-element vector, [x y width height], that specifies in pixels, the upper-left corner and size of a bounding box

4.5.1.4 Example

```
% Create a face detector object.
faceDetector    =    nxpvt.CascadeObjectDetector('FrontalFaceLBP',    'ScaleFactor',1.1,
'MinSize',[40 40], 'MaxSize',[70 70], 'SkipOdd',1, 'MergeThreshold',1);
% Read the input image.
img = nxpvt.imread('visionteam.jpg');
% Detect faces.
bboxes = step(faceDetector, img);
% Annotate detected faces.
IFaces = nxpvt.cv.rectangle(img, bboxes, [255 0 0], 2);
nxpvt.imshow(IFaces);
```

4.5.2 Convolutional Neural Networks

4.5.2.1 Object creation

`obj = CNN(cnnMatFileName, imgHeight, imgWidth)` – creates a CNN from the .mat saved network *cnnMatFileName* that is trained on images with imgHeight and imgWidth.
For example:
`alxNet = nxpvt.CNN('alexnet.mat', 227, 227);`

4.5.2.2 Method loadClassNames

`classNames = alxNet.loadClassNames(cnnClassNamesFile)` – loads saved .mat containing class names.
For example:
`classNames = alxNet.loadClassNames('alexnet_classes.mat');`

4.5.2.3 Method predict

`[percentages, classes] = cnnObj.predict(inputImage)` – returns the classes detected and the associated percentages.
For example:
`[percentage, classIdx] = alxNet.predict(inImg);`

4.6 OpenCV wrappers

4.6.1 Object tracking

4.6.1.1 Kalman filter

The Kalman filter is an algorithm that uses a noisy state-space representation and a series of noisy observation to estimate unknown variables.
The algorithm works in a two-step process:

1. Predict - Uses the old state and the model to predict the new state.
2. Correct - Uses the current observation to correct the predicted state.

4.6.1.2 Object Creation

`kf = KalmanFilter(dynamParams, measureParams, controlParams, datatype)` creates a Kalman filter object that has a state with dynamParams variables,

measureParams measured variables, controlParams control variables, and data type determined by datatype.

4.6.1.3 Methods

`kf.setControlMatrix(B)` sets control matrix, where B is the control matrix, specified as a UMat.

`kf.setMeasurementMatrix(H)` sets measurement matrix, where H is the measurement matrix, specified as a UMat.

`kf.setProcessNoiseCov(Q)` sets process noise covariance matrix, where Q is the process noise covariance matrix, specified as a UMat.

`kf.setMeasurementNoiseCov(R)` sets measurement noise covariance matrix, where R is the measurement noise covariance matrix, specified as a UMat.

`kf.setErrorCovPre(P)` sets priori error covariance matrix, where P is the priori error covariance matrix, specified as a UMat.

`kf.setErrorCovPost(P)` sets posteriori error covariance matrix, where P is the posteriori error covariance matrix, specified as a UMat.

`kf.setStatePre(x)` sets priori state, where x is the priori state, specified as a UMat.

`kf.setStatePost(x)` sets posteriori state, where x is the posteriori state, specified as a UMat.

`kf.getErrorCovPre(P)` gets priori error covariance matrix, where P is the priori error covariance matrix, specified as a UMat.

`kf.getErrorCovPost(P)` gets posteriori error covariance matrix, where P is the posteriori error covariance matrix, specified as a UMat.

`kf.setStatePre(x)` gets priori state, where x is the priori state, specified as a UMat.

`kf.setStatePost(x)` gets posteriori state, where x is the posteriori state, specified as a UMat.

`kf.predict()` predicts state having no control.

`kf.predict(u)` predicts state having control, where u is the control vector, specified as a UMat.

`kf.correct(x)` corrects state, where x is the measured vector, specified as a UMat.

4.6.1.4 Example

```
% Create a Kalman filter object
kf = nxpvt.cv.KalmanFilter(int32(2), int32(1), int32(0), int32(0));
```

```
% Initialize state with position 1 and speed 0
initialState = [1; 0];
initialStateUMat = nxpvt.UMat(single(initialState));
kf.setStatePost(initialStateUMat);

% Time elapsed between observations
T = 0.1;

% Set transition matrix
F = [1 T; 0 1];
FUMat = nxpvt.UMat(single(F));
kf.setTransitionMatrix(FUMat);

% Set process noise covariance matrix
sd2 = 14^2;
Q = [0.25*T^4*sd2 0.5*T^3*sd2; 0.5*T^3*sd2  T^2*sd2];
QUMat = nxpvt.UMat(single(Q));
kf.setProcessNoiseCov(QUMat);

% Set error covariance posteriori matrix
P = [0.05 0; 0 0.05];
PUMat = nxpvt.UMat(single(P));
kf.setErrorCovPost(PUMat);

% Set measurement matrix
H = [0 1];
HUMat = nxpvt.UMat(single(H));
kf.setMeasurementMatrix(HUMat);

% Set measurement noise covariance
R = [1];
RUMat = nxpvt.UMat(single(R));
kf.setMeasurementNoiseCov(RUMat);

% Use other 10 observations of position
for i = 1:10
    % Predict the new state
    estimPriTransUMat = kf.predict();

    % Correct the new state
    measurement = i+1;
    measurementUMat = nxpvt.UMat(single(measurement));
    estimPostTransUMat = kf.correct(measurementUMat);
end

% Estimated speed
estimPostTrans = estimPostTransUMat.data;
speed = estimPostTrans(2)
```

How to Reach Us:

Home Page:
www.nxp.com

Web Support:
www.nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

NXP Semiconductor reserves the right to make changes without further notice to any products herein. NXP Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. NXP Semiconductor does not convey any license under its patent rights nor the rights of others. NXP Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the NXP Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use NXP Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold NXP Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that NXP Semiconductor was negligent regarding the design or manufacture of the part.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.
Microsoft and .NET Framework are trademarks of Microsoft Corporation.
Flexera Software, Flexlm, and FlexNet Publisher are registered trademarks or trademarks of Flexera Software, Inc. and/or InstallShield Co. Inc. in the United States of America and/or other countries.
NXP, the NXP logo, CodeWarrior and ColdFire are trademarks of NXP Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Flexis and Processor Expert are trademarks of NXP Semiconductor, Inc. All other product or service names are the property of their respective owners