**CHAPTER 2**

**DIGITAL VIDEO STABILIZATION**

1. **Introduction**

Digital Video Stabilization aims to remove the unwanted movement of the video by repositioning and rotating the frames so that the video looks stable. Although there is existed numerous different methods for achieving this effect, video stabilization frameworks usually implement these 3 stages in sequence: motion estimation, motion smoothing and, ﬁnally, motion compensation. Due to the interest in deploying video stabilization algorithms in power-constrained devices (including hand-held devices such as smartphones, for example, It’s imperative to develop methods that are not prohibitively computationally intensive but that still yield reasonable results. Hardware based methods to augment the camera lens and sensors’ capabilities in acquiring images have been proposed. However, these solutions are too expensive and too cumbersome to add on to ordinary cameras. Therefore, a completely software-based solution is preferred. The stabilization flow is shown in Figure 1.

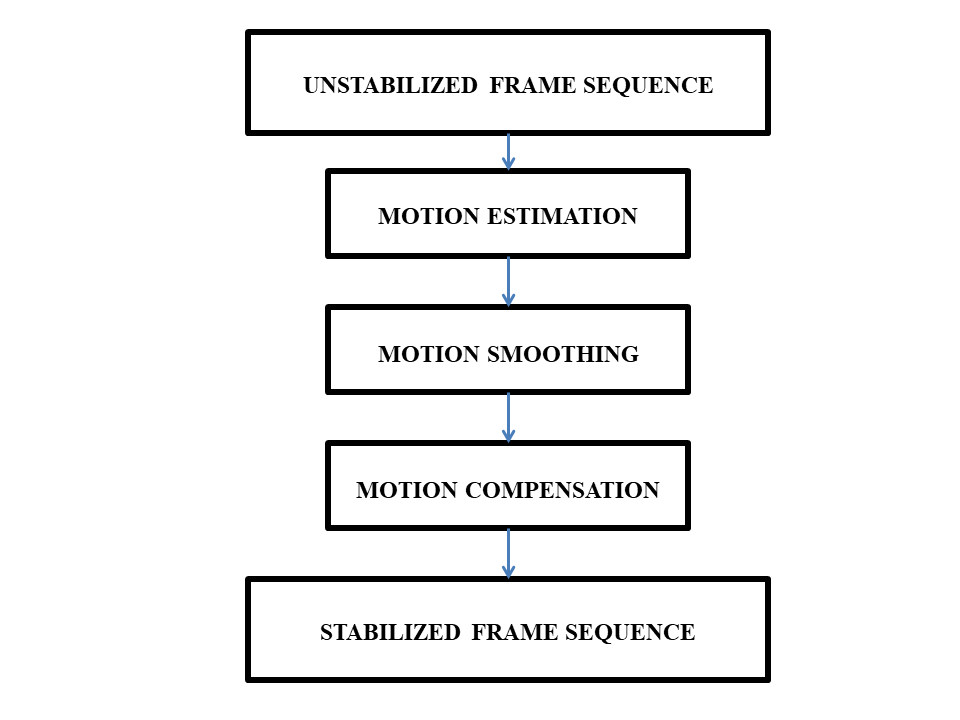


Figure 2.1: General Digital Video Stabilization Flow

* 1. **MOTION ESTIMATION**

Motion estimation is an important step for video stabilization algorithms. Motion estimation is the process of determining motion vectors that describe the transformation from one 2D image to another; usually from adjacent frames in a video sequence. It is an ill-posed problem as the motion is in three dimensions but the images are a projection of the 3D scene onto a 2D plane. The motion vectors may relate to the whole image (global motion estimation) or specific parts, such as rectangular blocks, arbitrary shaped patches or even per pixel. The motion vectors may be represented by a translational model or many other models that can approximate the motion of a real video camera, such as rotation and translation in all three dimensions and zoom. It is the attempt for estimating the displacement of points between two successive video frames. In video frame’s motion is manifested as alteration in pixels intensity values which can be used to determine motion of objects.

Equation 2.1 presents a simple representation of the problem where (𝑡) and (𝑡 +∆𝑡) are two consecutive video frames. As depicted in Figure 2.2 ∆𝑥 and ∆𝑦 are the motion vector components.

𝐼 (𝑥, *y, t*) = 𝐼 (𝑥 +∆𝑥, 𝑦 +∆𝑦, 𝑡 +∆𝑡) (2.1)

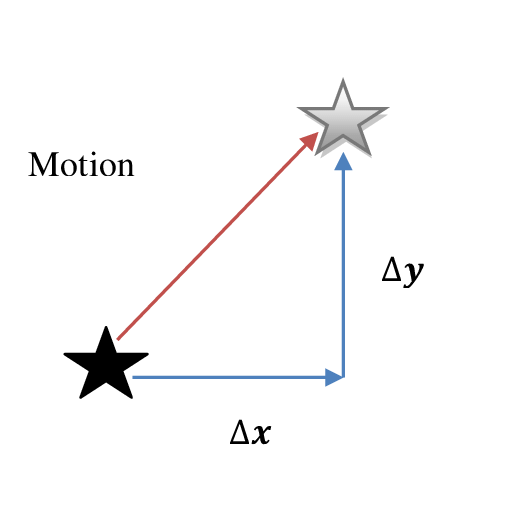


Figure 2.2: Motion Vector Component

In order to find ∆𝑥 and ∆𝑦 the following equation should be solved.

*I*(𝑥, *y*, t) − *I*(𝑥 +∆𝑥,𝑦 +∆𝑦,𝑡 +∆𝑡) = 0 (2.2)

However the existence of noise, camera displacements and light alterations can prevent the zero difference. Direct and Indirect motion estimation techniques are two different approaches to the problem. After introducing different motion models for two dimensional images, direct and indirect motion estimation techniques are discussed.

1. **Principle Types of Motion Models**

Mathematical equations describing the mapping procedure of pixel coordinates between two images are referred to as motion models. Any pixel coordinate in an image can be described as; 𝒙 = (𝑥, *y*) ∈ 𝑅2. For most transformations nonhomogeneous coordinates are sufficient however for perspective or projective transformations homogeneous transformations are needed. In what follows we give examples for various transformation types.

* + - 1. **Translation Transformation**

Translational motion is the motion in which all points of a moving body move uniformly in the same line or direction. If an object is executing translational motion then there is no change in its orientation relative to a fixed point. One example of translational motion is the motion of a bullet fired from a gun. For example, a train is moving in its track, a man walking on the road, birds flying in the sky, etc.

When a body is shifted or moved from one point to another point, then the body said to be experienced translational motion. It is the motion in which all points of a moving body move uniformly in the same line or direction. If an object is executing translational motion then there is no change in its orientation relative to a fixed point. So, if the object moves such that all the particles in the object move parallel to each other then the motion is called pure translational motion.

In this type of motion, all points of the body have velocities and accelerations that are the same in magnitude and direction at every instant of time. All points describe identical trajectories. By this, we mean that trajectories would coincident when they are placed one above the other. Basically, the orientation of the body remains fixed relative to a fixed axis.

Also from the definition of translatory motion for an object to execute Translational motion, there is no change in its orientation relative to a fixed point and all points of a moving body move uniformly in the same line or direction.

Equation 2.3 describes a two dimensional translations. This transformation preserves the orientation.

𝑥′ = 𝑥 +𝑡 (2.3)

or

𝑥′ = [*I* *t*] 𝑥̅ , where I is a (2×2) identity matrix

* + - 1. **Euclidean Transformation**

Euclidean transformation is the union of Translation and Rotation transformations. In mathematics, a Euclidean transformation is a geometric transformation of a Euclidean space that preserves the Euclidean distance between every pair of points. The Euclidean transformations include rotations, translations, reflections, or their combination. Sometimes reflections are excluded from the definition of a Euclidean transformation by imposing that the transformation also preserve the handedness of figures in the Euclidean space (a reflection would not preserve handedness; for instance, it would transform a left hand into a right hand). To avoid ambiguity, this smaller class of transformations is known as rigid motions or proper rigid transformations' (informally, also known as roto-translations).

In general, any proper Euclidean transformation can be decomposed as a rotation followed by a translation, while any Euclidean transformation can be decomposed as an improper rotation followed by a translation (or as a sequence of reflections). Any object will keep the same shape and size after a proper rigid transformation. Euclidean transformation can be expressed as the following equation [14].

𝑥′ = [ *R* *t* ]*̅*  (2.4)

𝑅 =

𝑅𝑅𝑇 = 𝐼

|𝑅| = 1

* + - 1. **Similarity Transformation**

A similarity transformation is one or more rigid transformations (reflection, rotation, translation) followed by a dilation. When a figure is transformed by a similarity transformation, an image is created that is similar to the original figure. In other words, two figures are similar if a similarity transformation will carry the first figure to the second figure.

Equation 2.9 [14] describes similarity transform also known as scaled rotation. In this transformation angles between lines are preserved.

𝑥′ = [*sR* ]̅ = 𝑥̅ (2.5)

Where 𝑠 is an arbitrary scale factor.

* + - 1. **Affine Transformation**

Affine transformation is an automorphism of an affine space. More specifically, it is a function mapping an affine space onto itself that preserves the dimension of any affine subspaces (meaning that it sends points to points, lines to lines, planes to planes, and so on) and also preserves the ratio of the lengths of parallel line segments. Consequently, sets of parallel affine subspaces remain parallel after an affine transformation. An affine transformation does not necessarily preserve angles between lines or distances between points, though it does preserve ratios of distances between points lying on a straight line. Affine transformation described by the following equation preserves the parallelism between lines [14]. The parameter 𝐴 is an arbitrary 2×3 matrix.

𝑥′ = 𝐴𝑥̅ = 𝑥̅ (2.6)

* + - 1. **Homography Transformation**

The Homography transformation is a popular geo-referencing technique used worldwide. It is based on quite complex geometric and mathematic concepts, known as "homogeneous coordinates" and "projective planes". The familiar Cartesian plane is composed by a set of points which have a one-to-one correlation to pairs of real numbers, i.e. X-Y on the two axis. The "projective plane" instead is a superset of that real plane where for each point we also consider all possible (infinite) straight lines towards space.

Homography transformation which is also referred to as perspective or projective transformation operates on homogenous coordinate and can be described by equation 2.7 [14].

(2.7)

𝐻 ̃ is an arbitrary 3×3 homogenous matrix. The resulting homogenous 𝑥 ̃ should be normalized in order to obtain an inhomogeneous result x [14].

(2.8)

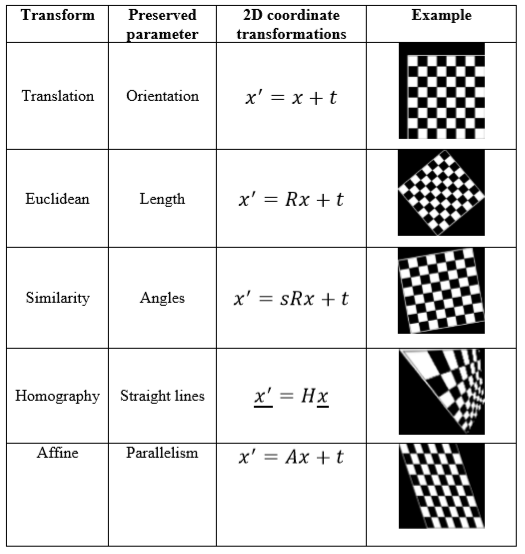
Table 2.1 represents an organized summary of different motion models.

* + 1. **Direct Motion Estimation Technique**

In direct approach to estimate motion, all the pixels in the frame are in use to estimate the motion. Unlike feature based methods which are adopted in this research, in direct motion estimation methods unknown parameters are recovered directly from measurable image quantities such as intensity.

The advantage of using direct methods is increasing the sub-pixel accuracy. These methods are also more efficient in handling the data which does not fit the model (outliers). In direct methods spatial derivatives in each frame are compared using an affine model.

Table 2.1: Motion Models



The first step in most of direct methods is to determine brightness constancy constraint. Assuming I and J as two consecutive video frames we can write:

𝐽(𝑥,𝑦)= I (𝑥 +𝑢(𝑥,𝑦),𝑦 +𝑣(𝑥,𝑦)) (2.9)

Where (𝑢,𝑣) represent pixel displacement between the frames. If (𝑢,𝑣) are small enough and I is linearized around (𝑥,𝑦) the following constraint can be obtained

𝐼𝑥𝑢 + 𝐼𝑦𝑣 +𝐼𝑡 = 0 (2.10)

𝐼𝑡 = 𝐼 −𝐽

In this equation 𝐼𝑥 and 𝐼𝑦 denote spatial derivatives of the brightness. There will be one such equation for every pixel in the frame.

In the second step of direct motion estimation methods another constraint describing the image motion variations in the total image is also defined. In most of the direct methods the affine motion model is described as follows.

𝑢(𝑥,𝑦) = 𝑎1 +𝑎2𝑥 +𝑎3𝑦 (2.11)

𝑣(𝑥,𝑦) = 𝑎4 +𝑎5𝑥 +𝑎6𝑦

This model gives better results when the image depicts a distant scene. Substituting equation 2.11 in equation 2.10 we have.

𝐼𝑥(𝑎1 +𝑎2𝑥 +𝑎3𝑦)+ 𝐼𝑦(𝑎4 +𝑎5𝑥 +𝑎6𝑦)+𝐼𝑡 = 0 (2.12)

For each pixel of the image we have one constraint containing six parameters which are identical for all pixels so six constrains are adequate to solve the equation.

* + 1. **Indirect Motion Estimation Technique**

In indirect motion estimation methods, image features are used with the purpose of estimating motion between frames. In these methods the first step is to find strong features of each frame. There are several methods to find feature points in an image. Harris, FAST and SUSAN corner detection are some examples. Generally corner points have higher chance to be in the next frame as well.

As each feature will have a distinct vector, a filter is required in indirect algorithms to filter out the outliners. RANSAC is a popular example. The following steps constitute the indirect algorithm to compute a two dimensional homographic transformation between two frames.

1. Corner point features are computed in sub-pixel precision.
2. Considering the similarity and proximity of the neighborhood point intensity, a set of corner points matches is computed.
3. RANSAC robust assessment for N samples

* Selection of four random correspondences based on which the homography H is calculated.
* For all the assumed correspondences a geometric distance error should be computed.
* Choosing Correspondences with the geometric distance error less than a threshold value based on which number of inliers consistent with H is computed.

1. Optimal re-estimation of H from the inliers.
2. Determination of more corner point correspondences based on the H calculated in the previous step with the purpose of defining a search region around the transferred point position.
   1. **Field Programmable Gate Array(FPGA)**

A Field-Programmable Gate Array is an integrated circuit silicon chip which has array of logic gates and this array can be programmed in the field i.e. the user can overwrite the existing configurations with its new defined configurations and can create their own digital circuit on field. The FPGAs can be considered as blank slate. FPGAs do nothing by itself whereas it is up to designers to create a configuration file often called a bit file for the FPGA. The FPGA will behave like the digital circuit once it is loaded with a bit file.

Whereas in microcontrollers, this is not the case as microcontrollers cannot be programmed or restructured in the field. The user is neither allowed to overwrite its existing configurations nor can they create any digital circuit on field. The microcontrollers are easy to program and the community is also wide. The microcontrollers are custom built mini computers which comes in IC form while FPGAs are only contains logic blocks that can again be rewired electrically. Also in terms of microcontrollers, it consumes less power than FPGAs. The FPGAs is known to be costly and it requires more cost than microcontroller when it comes to building any device. FPGAs takes considerably much more time to set-up while the microcontrollers are available readily built for specific applications. Example of FPGA families include the following-

* Altera Stratix II and Cyclone II families
* Atmel AT6000 and AT40K families
* Terasic DE10 Standard(SoC)
* Intel Cyclone V SoC Develoment Board
* PYNQ-Z1 Xilinx Zynq 7000 EEP
* Lattice LatticeEC and LatticeECP families
* Xilinx Spartan-3 and Virtex-4 families
* Actel SX and Axcelerator families
* Quicklogic Eclipse II family
* Actel SX and Axcelerator families
* Quicklogic Eclipse II family
* Actel SX and Axcelerator families
* Quicklogic Eclipse II family
  + 1. **FPGA Architecture**

An FPGA has a regular structure of logic cells or modules and interlinks which is under the developers and designers complete control. The FPGA is built with mainly three major blocks such as Configurable Logic Block (CLB), I/O Blocks or Pads and Switch Matrix/ Interconnection Wires. Each block will be discussed below in brief.

* **CLB (Configurable Logic Block):** These are the basic cells of FPGA. It consists of one 8-bit function generator, two 16-bit function generators, two registers (flip-flops or latches), and reprogrammable routing controls (multiplexers). The CLBs are applied to implement other designed function and macros. Each CLBs have inputs on each side which makes them flexile for the mapping and partitioning of logic.
* **I/O Pads or Blocks:** The Input/Output pads are used for the outside peripherals to access the functions of FPGA and using the I/O pads it can also communicate with FPGA for different applications using different peripherals.
* **Switch Matrix/ Interconnection Wires:** Switch Matrix is used in FPGA to connect the long and short interconnection wires together in flexible combination. It also contains the transistors to turn on/off connections between different lines.

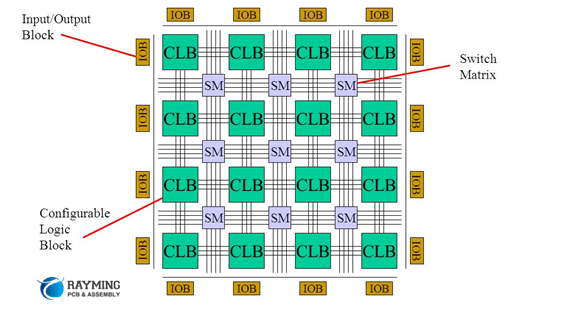


Figure 2.3: FPGA Architecture

* + 1. **Usage of FPGA**

As mentioned above that microcontrollers have some limitation and cannot be used to perform task in parallel as microcontroller and microprocessors runs on sequential execution of programs which makes it a bit slow in some applications, in this scenario the FPGAs has an advantage and can be effectively used. Also microcontroller can perform limited tasks because they come with instructions and their circuitry. A programmer has to abide by the restrictions while developing code. So in this scenario also, the FPGAs have advantage.

However, in the case of microcontrollers, the processor switches from one code to another to achieve some level of parallelism. It is easier to write codes on microcontrollers than FPGAs. The parallel processing capability of FPGAs enables to control interruptions effectively by using Finite State Machines (FSMs).

In the case of microcontrollers, FPGA can be rewired easily just by reprogramming it. The configuration in an FPGA is loaded on the configurable logic cells when the power is switched on.

There is no needed to make any changes in the hardware to reprogram the FPGA. FPGAs are suitable for high-speed processing of parallel data and comes with a high degree of customizability. However, they also have the drawbacks of prototype operation and complexity of configuration. So, the FPGAs can be chosen with these advantages over microcontrollers. Let’s start the FPGA programming and emphasize more on FPGA programming.

* + 1. **Programming Steps of FPGA**

Programming of FPGAs is done by HDLs (Hardware Description Languages). There are several HDLs are available but the VHDL and Verilog are widely used HDLs. Even though there are some similarity between HDL code and high-level software programming language but the two are fundamentally different. Software codes are a sequence of operations and perform the processing in sequence whereas HDL code is a schematic that uses text to introduce components and create interconnections with parallel processing.

To understand it in a better way, notice the difference between steps involved in Hardware and Software Design Flow in FPGAs and Microcontrollers respectively.

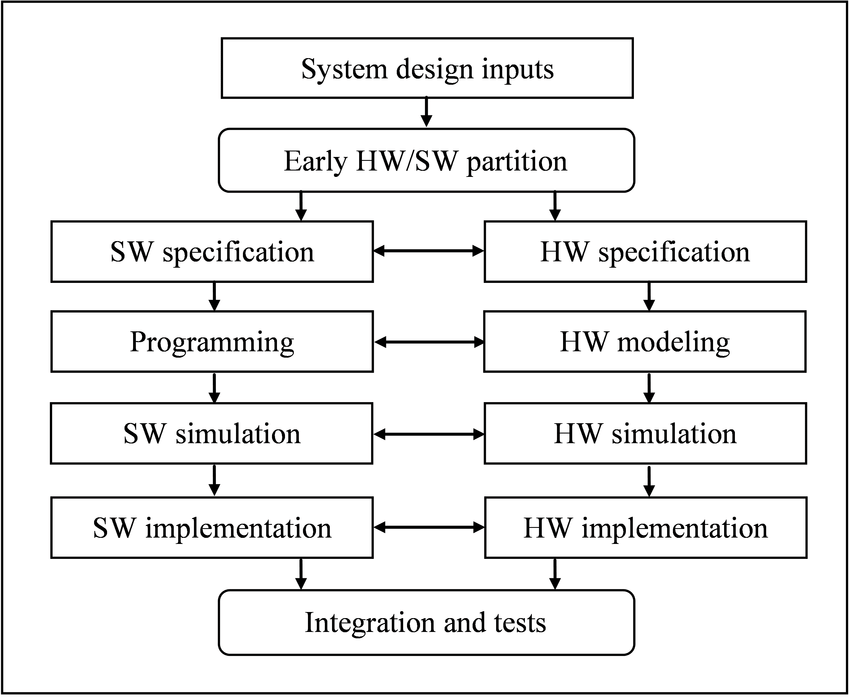


Figure 2.4 : Hardware and Software Design Flow

The typical Hardware and Software Flow is shown above. The Hardware design flow is used to program FPGAs whereas Software Design Flow is used to program the typical Microcontrollers and Microprocessors. The important steps involve in programming FPGAs are as follows.

* Synthesis: The First step is the synthesis which takes HDL code and translate into netlist which is a textual description of a circuit diagram or schematic.
* Simulation: After synthesis, the next step involves the simulation which is used to verify if the design specified in the netlist functions correctly.
* Convert netlist into Binary Format: Once the design is verified, the next is convert netlist into binary format. The components and connections are mapped to CLBs and the design is placed and routed to fit onto the target FPGA (i.e Place and Route).
* Perform Second Simulation: To see the design quality, a second simulation is performed.
* Generate Bit File: Finally a bit file is generated to load the design onto FPGA (A .bit file is a configuration file which is used to program all of the resources within FPGA).
* Verify and Debug: At last, using different tools the design is verified and debugged while it is running on the FPGA.
  + 1. **Programming Language and Tools**

As mentioned above there are several programming languages and tools available to program and debug FPGAs but most widely used are VHDL and Verilog. Both VHDL and Verilog are well established and wide support HDLs. In terms of program FPGA, one needs to forget the software coding behaviour and start thinking about logic gates and circuits to implement the functionality that one wants to run on FPGAs.

There are many FPGA dev tools available such as:

**VHDL/Verilog:** Both languages provide structures to describe the inherently parallel nature of FPGA / ASIC development. Due to their initial use to describe the behaviour of the circuits prior to the generation of synthesis tools, these languages also support test benches to test the design being implemented.

**LabVIEW FPGA:** The LabVIEW is a graphical language which gives a completely different way of programming a FPGA. LabVIEW FPGA is the FPGA compilation uses a cloud-based option, which speeds up the compilation time significantly.

**MATLAB:** The MATLAB is the language which can play a vital role and should be studied. The MATLAB is generally used to generate filters for signal processing, develop image processing algorithms and almost any other algorithm. But apart from this, it is possible to go from MATLAB model to FPGA using the HDL coder. The traceability enables the high integrity applications can be developed using this approach. HDL coder enables to perform hardware (FPGA) in the loop testing and co-simulation to see the difference between the original algorithm and the implemented hardware algorithm, which helps to explore the design space.

**C/C++/System C:** High-Level Synthesis (HLS) is a recent technique for utilizing programmable logic without using the traditional hardware definition languages (Verilog / VHDL) and with no need for prior knowledge of FPGA/VLSI design practices. The HLS tools compile a C/C++ function into logic elements, aiming to utilize the programmable device efficiently for speedy operation and economic resource usage. This opens an opportunity for regular programmers to write custom coprocessing logic without being FPGA experts.

Apart from this there are other tools similar to this such as **MyHDL, JHDL, BSV, System Verilog, SPINAL HDL, CHISEL etc.**