**Introduction to SIFT (Scale-Invariant Feature Transform)**

**Goal**

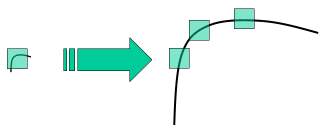
In this chapter,

We will learn about the concepts of SIFT algorithm

We will learn to find SIFT Keypoints and Descriptors.

**Theory**

In last couple of chapters, we saw some corner detectors like Harris etc. They are rotation-invariant, which means, even if the image is rotated, we can find the same corners. It is obvious because corners remain corners in rotated image also. But what about scaling? A corner may not be a corner if the image is scaled. For example, check a simple image below. A corner in a small image within a small window is flat when it is zoomed in the same window. So Harris corner is not scale invariant.



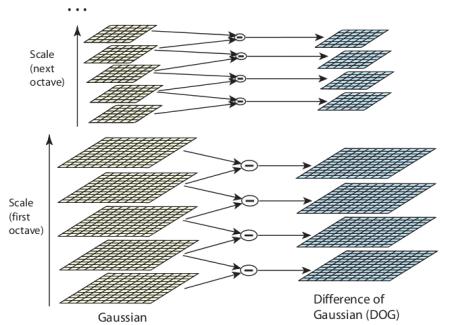
So, in 2004, D.Lowe, University of British Columbia, came up with a new algorithm, Scale Invariant Feature Transform (SIFT) in his paper, Distinctive Image Features from Scale-Invariant Keypoints, which extract keypoints and compute its descriptors. (This paper is easy to understand and considered to be best material available on SIFT. So this explanation is just a short summary of this paper).

There are mainly four steps involved in SIFT algorithm. We will see them one-by-one.

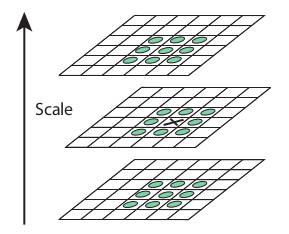
**1. Scale-space Extrema Detection**

From the image above, it is obvious that we can’t use the same window to detect keypoints with different scale. It is OK with small corner. But to detect larger corners we need larger windows. For this, scale-space filtering is used. In it, Laplacian of Gaussian is found for the image with various \sigma values. LoG acts as a blob detector which detects blobs in various sizes due to change in \sigma. In short, \sigma acts as a scaling parameter. For eg, in the above image, gaussian kernel with low \sigma gives high value for small corner while guassian kernel with high \sigma fits well for larger corner. So, we can find the local maxima across the scale and space which gives us a list of (x,y,\sigma) values which means there is a potential keypoint at (x,y) at \sigma scale.

But this LoG is a little costly, so SIFT algorithm uses Difference of Gaussians which is an approximation of LoG. Difference of Gaussian is obtained as the difference of Gaussian blurring of an image with two different \sigma, let it be \sigma and k\sigma. This process is done for different octaves of the image in Gaussian Pyramid. It is represented in below image:



Once this DoG are found, images are searched for local extrema over scale and space. For eg, one pixel in an image is compared with its 8 neighbours as well as 9 pixels in next scale and 9 pixels in previous scales. If it is a local extrema, it is a potential keypoint. It basically means that keypoint is best represented in that scale. It is shown in below image:



egarding different parameters, the paper gives some empirical data which can be summarized as, number of octaves = 4, number of scale levels = 5, initial \sigma=1.6, k=\sqrt{2} etc as optimal values.

**2. Keypoint Localization**

Once potential keypoints locations are found, they have to be refined to get more accurate results. They used Taylor series expansion of scale space to get more accurate location of extrema, and if the intensity at this extrema is less than a threshold value (0.03 as per the paper), it is rejected. This threshold is called contrastThreshold in OpenCV

DoG has higher response for edges, so edges also need to be removed. For this, a concept similar to Harris corner detector is used. They used a 2x2 Hessian matrix (H) to compute the pricipal curvature. We know from Harris corner detector that for edges, one eigen value is larger than the other. So here they used a simple function,

If this ratio is greater than a threshold, called edgeThreshold in OpenCV, that keypoint is discarded. It is given as 10 in paper.

So it eliminates any low-contrast keypoints and edge keypoints and what remains is strong interest points.

**3. Orientation Assignment**

Now an orientation is assigned to each keypoint to achieve invariance to image rotation. A neigbourhood is taken around the keypoint location depending on the scale, and the gradient magnitude and direction is calculated in that region. An orientation histogram with 36 bins covering 360 degrees is created. (It is weighted by gradient magnitude and gaussian-weighted circular window with \sigma equal to 1.5 times the scale of keypoint. The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation. It creates keypoints with same location and scale, but different directions. It contribute to stability of matching.

**4. Keypoint Descriptor**

Now keypoint descriptor is created. A 16x16 neighbourhood around the keypoint is taken. It is devided into 16 sub-blocks of 4x4 size. For each sub-block, 8 bin orientation histogram is created. So a total of 128 bin values are available. It is represented as a vector to form keypoint descriptor. In addition to this, several measures are taken to achieve robustness against illumination changes, rotation etc.

**5. Keypoint Matching**

Keypoints between two images are matched by identifying their nearest neighbours. But in some cases, the second closest-match may be very near to the first. It may happen due to noise or some other reasons. In that case, ratio of closest-distance to second-closest distance is taken. If it is greater than 0.8, they are rejected. It eliminaters around 90% of false matches while discards only 5% correct matches, as per the paper.

So this is a summary of SIFT algorithm. For more details and understanding, reading the original paper is highly recommended. Remember one thing, this algorithm is patented. So this algorithm is included in Non-free module in OpenCV.

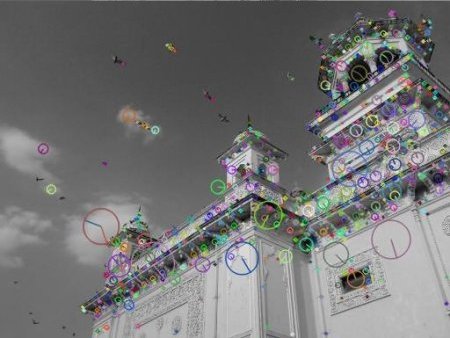
**SIFT in OpenCV**

So now let’s see SIFT functionalities available in OpenCV. Let’s start with keypoint detection and draw them. First we have to construct a SIFT object. We can pass different parameters to it which are optional and they are well explained in docs.

sift.detect() function finds the keypoint in the images. You can pass a mask if you want to search only a part of image. Each keypoint is a special structure which has many attributes like its (x,y) coordinates, size of the meaningful neighbourhood, angle which specifies its orientation, response that specifies strength of keypoints etc.

OpenCV also provides cv2.drawKeyPoints() function which draws the small circles on the locations of keypoints. If you pass a flag, cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS to it, it will draw a circle with size of keypoint and it will even show its orientation .

See the two results below:



Now to calculate the descriptor, OpenCV provides two methods.

1-Since you already found keypoints, you can call sift.compute() which computes the descriptors from the keypoints we have found. **Eg: kp,des = sift.compute(gray,kp)**

2-If you didn’t find keypoints, directly find keypoints and descriptors in a single step with the function, **sift.detectAndCompute().**

Chapter 2

**Definition of a Thread:**

Thread or a Thread of Execution is defined in computer science as the smallest unit that can be scheduled in an operating system. Threads are normally created by a fork of a computer script or program in two or more parallel (which is implemented on a single processor by multitasking) tasks. Threads are usually contained in processes. More than one thread can exist within the same process. These threads share the memory and the state of the process. In other words: They share the code or instructions and the values of its variables.

There are two different kind of threads:

Kernel threads

User-space Threads or user threads

Kernel Threads are part of the operating system, while User-space threads are not implemented in the kernel.

In a certain way, user-space threads can be seen as an extension of the function concept of a programming language. So a thread user-space thread is similar to a function or procedure call. But there are differences to regular functions, especially the return behaviour.

Threads and global Variables Every process has at least one thread, i.e. the process itself. A process can start multiple threads. The operating system executes these threads like parallel "processes". On a single processor machine, this parallelism is achieved by thread scheduling or timeslicing.

Advantages of Threading:

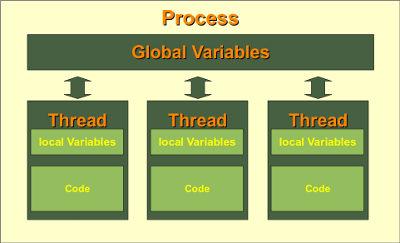
Multithreaded programs can run faster on computer systems with multiple CPUs, because theses threads can be executed truly concurrent.

A program can remain responsive to input. This is true both on single and on multiple CPU

Threads of a process can share the memory of global variables. If a global variable is changed in one thread, this change is valid for all threads. A thread can have local variables.

The handling of threads is simpler than the handling of processes for an operating system. That's why they are sometimes called light-weight process (LWP)

Threads in Python



**Threads in Python:**

There are two modules which support the usage of threads in Python:

**.**thread

and

**.**threading

Please note: The thread module has been considered as "deprecated" for quite a long time. Users have been encouraged to use the threading module instead. So,in Python 3 the module "thread" is not available anymore. But that's not really true: It has been renamed to "\_thread" for backwards incompatibilities in Python3.

The module "thread" treats a thread as a function, while the module "threading" is implemented in an object oriented way, i.e. every thread corresponds to an object.

**Thread-Local Data**

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of local (or a subclass) and store attributes on it:

mydata = threading.local()

mydata.x = 1

The instance’s values will be different for separate threads.

class threading.local

A class that represents thread-local data.

For more details and extensive examples, see the documentation string of the \_threading\_local module.

**Thread Objects**

The Thread class represents an activity that is run in a separate thread of control. There are two ways to specify the activity: by passing a callable object to the constructor, or by overriding the **run()** method in a subclass. No other methods (except for the constructor) should be overridden in a subclass. In other words, only override the **\_\_init\_\_()** and **run()** methods of this class.

Once a thread object is created, its activity must be started by calling the thread’s **start()** method. This invokes the run() method in a separate thread of control.

Once the thread’s activity is started, the thread is considered ‘alive’. It stops being alive when its **run()** method terminates – either normally, or by raising an unhandled exception. The is\_alive() method tests whether the thread is alive.

Other threads can call a thread’s join() method. This blocks the calling thread until the thread whose join() method is called is terminated.

A thread has a name. The name can be passed to the constructor, and read or changed through the name attribute.

A thread can be flagged as a “daemon thread”. The significance of this flag is that the entire Python program exits when only daemon threads are left. The initial value is inherited from the creating thread. The flag can be set through the daemon property or the daemon constructor argument.

**Note:** Daemon threads are abruptly stopped at shutdown. Their resources (such as open files, database transactions, etc.) may not be released properly. If you want your threads to stop gracefully, make them non-daemonic and use a suitable signalling mechanism such as an Event.

There is a “main thread” object; this corresponds to the initial thread of control in the Python program. It is not a daemon thread.

There is the possibility that “dummy thread objects” are created. These are thread objects corresponding to “alien threads”, which are threads of control started outside the threading module, such as directly from C code. Dummy thread objects have limited functionality; they are always considered alive and daemonic, and cannot be join()ed. They are never deleted, since it is impossible to detect the termination of alien threads.

class threading.**Thread**(group=None, target=None, name=None, args=(), kwargs={}, \*, daemon=None)

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If not None, daemon explicitly sets whether the thread is daemonic. If None (the default), the daemonic property is inherited from the current thread.

If the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.\_\_init\_\_()) before doing anything else to the thread.

Changed in version 3.3: Added the daemon argument.

**start()**

Start the thread’s activity.

It must be called at most once per thread object. It arranges for the object’s **run()** method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

run()

Method representing the thread’s activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object’s constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

join(timeout=None)

Wait until the thread terminates. This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is\_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raise the same exception.

name

A string used for identification purposes only. It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

**getName()**

**setName()**

Old getter/setter API for name; use it directly as a property instead.

ident

The ‘thread identifier’ of this thread or None if the thread has not been started. This is a nonzero integer. See the get\_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

is\_alive()

Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

daemon

A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

**isDaemon()**

**setDaemon()**

Old getter/setter API for daemon; use it directly as a property instead.

CPython implementation detail: In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use multiprocessing or concurrent.futures.ProcessPoolExecutor. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

**Lock Objects**

A primitive lock is a synchronization primitive that is not owned by a particular thread when locked. In Python, it is currently the lowest level synchronization primitive available, implemented directly by the \_thread extension module.

A primitive lock is in one of two states, “locked” or “unlocked”. It is created in the unlocked state. It has two basic methods, acquire() and release(). When the state is unlocked, acquire() changes the state to locked and returns immediately. When the state is locked, acquire() blocks until a call to release() in another thread changes it to unlocked, then the acquire() call resets it to locked and returns. The release() method should only be called in the locked state; it changes the state to unlocked and returns immediately. If an attempt is made to release an unlocked lock, a RuntimeError will be raised.

Locks also support the context management protocol.

When more than one thread is blocked in acquire() waiting for the state to turn to unlocked, only one thread proceeds when a release() call resets the state to unlocked; which one of the waiting threads proceeds is not defined, and may vary across implementations.

All methods are executed atomically.

class threading.Lock

The class implementing primitive lock objects. Once a thread has acquired a lock, subsequent attempts to acquire it block, until it is released; any thread may release it.

Note that Lock is actually a factory function which returns an instance of the most efficient version of the concrete Lock class that is supported by the platform.

acquire(blocking=True, timeout=-1)

Acquire a lock, blocking or non-blocking.

When invoked with the blocking argument set to True (the default), block until the lock is unlocked, then set it to locked and return True.

When invoked with the blocking argument set to False, do not block. If a call with blocking set to True would block, return False immediately; otherwise, set the lock to locked and return True.

When invoked with the floating-point timeout argument set to a positive value, block for at most the number of seconds specified by timeout and as long as the lock cannot be acquired. A timeout argument of -1 specifies an unbounded wait. It is forbidden to specify a timeout when blocking is false.

The return value is True if the lock is acquired successfully, False if not (for example if the timeout expired).

Changed in version 3.2: The timeout parameter is new.

Changed in version 3.2: Lock acquisition can now be interrupted by signals on POSIX if the underlying threading implementation supports it.

**release()**

Release a lock. This can be called from any thread, not only the thread which has acquired the lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

When invoked on an unlocked lock, a RuntimeError is raised.

There is no return value.

RLock Objects

A reentrant lock is a synchronization primitive that may be acquired multiple times by the same thread. Internally, it uses the concepts of “owning thread” and “recursion level” in addition to the locked/unlocked state used by primitive locks. In the locked state, some thread owns the lock; in the unlocked state, no thread owns it.

To lock the lock, a thread calls its acquire() method; this returns once the thread owns the lock. To unlock the lock, a thread calls its release() method. acquire()/release() call pairs may be nested; only the final release() (the release() of the outermost pair) resets the lock to unlocked and allows another thread blocked in acquire() to proceed.

Reentrant locks also support the context management protocol.

class **threading.RLock**

This class implements reentrant lock objects. A reentrant lock must be released by the thread that acquired it. Once a thread has acquired a reentrant lock, the same thread may acquire it again without blocking; the thread must release it once for each time it has acquired it.

Note that RLock is actually a factory function which returns an instance of the most efficient version of the concrete RLock class that is supported by the platform.

acquire(blocking=True, timeout=-1)

Acquire a lock, blocking or non-blocking.

When invoked without arguments: if this thread already owns the lock, increment the recursion level by one, and return immediately. Otherwise, if another thread owns the lock, block until the lock is unlocked. Once the lock is unlocked (not owned by any thread), then grab ownership, set the recursion level to one, and return. If more than one thread is blocked waiting until the lock is unlocked, only one at a time will be able to grab ownership of the lock. There is no return value in this case.

When invoked with the blocking argument set to true, do the same thing as when called without arguments, and return true.

When invoked with the blocking argument set to false, do not block. If a call without an argument would block, return false immediately; otherwise, do the same thing as when called without arguments, and return true.

When invoked with the floating-point timeout argument set to a positive value, block for at most the number of seconds specified by timeout and as long as the lock cannot be acquired. Return true if the lock has been acquired, false if the timeout has elapsed.

Changed in version 3.2: The timeout parameter is new.

**release()**

Release a lock, decrementing the recursion level. If after the decrement it is zero, reset the lock to unlocked (not owned by any thread), and if any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed. If after the decrement the recursion level is still nonzero, the lock remains locked and owned by the calling thread.

Only call this method when the calling thread owns the lock. A RuntimeError is raised if this method is called when the lock is unlocked.

There is no return value.

**Condition Objects**

A condition variable is always associated with some kind of lock; this can be passed in or one will be created by default. Passing one in is useful when several condition variables must share the same lock. The lock is part of the condition object: you don’t have to track it separately.

A condition variable obeys the context management protocol: using the with statement acquires the associated lock for the duration of the enclosed block. The acquire() and release() methods also call the corresponding methods of the associated lock.

Other methods must be called with the associated lock held. The wait() method releases the lock, and then blocks until another thread awakens it by calling notify() or **notify\_all().** Once awakened, **wait()** re-acquires the lock and returns. It is also possible to specify a timeout.

The **notify()** method wakes up one of the threads waiting for the condition variable, if any are waiting. The **notify\_all()** method wakes up all threads waiting for the condition variable.

Note: the notify() and notify\_all() methods don’t release the lock; this means that the thread or threads awakened will not return from their wait() call immediately, but only when the thread that called notify() or notify\_all() finally relinquishes ownership of the lock.

The typical programming style using condition variables uses the lock to synchronize access to some shared state; threads that are interested in a particular change of state call wait() repeatedly until they see the desired state, while threads that modify the state call notify() or notify\_all() when they change the state in such a way that it could possibly be a desired state for one of the waiters. For example, the following code is a generic producer-consumer situation with unlimited buffer capacity:

# Consume one item

with cv:

while not an\_item\_is\_available():

cv.wait()

get\_an\_available\_item()

# Produce one item

with cv:

make\_an\_item\_available()

cv.notify()

The while loop checking for the application’s condition is necessary because wait() can return after an arbitrary long time, and the condition which prompted the notify() call may no longer hold true. This is inherent to multi-threaded programming. The wait\_for() method can be used to automate the condition checking, and eases the computation of timeouts:

# Consume an item

with cv:

cv.wait\_for(an\_item\_is\_available)

get\_an\_available\_item()

To choose between notify() and notify\_all(), consider whether one state change can be interesting for only one or several waiting threads. E.g. in a typical producer-consumer situation, adding one item to the buffer only needs to wake up one consumer thread.

class threading.Condition(lock=None)

This class implements condition variable objects. A condition variable allows one or more threads to wait until they are notified by another thread.

If the lock argument is given and not None, it must be a Lock or RLock object, and it is used as the underlying lock. Otherwise, a new RLock object is created and used as the underlying lock.

Changed in version 3.3: changed from a factory function to a class.

acquire(\*args)

Acquire the underlying lock. This method calls the corresponding method on the underlying lock; the return value is whatever that method returns.

release()

Release the underlying lock. This method calls the corresponding method on the underlying lock; there is no return value.

wait(timeout=None)

Wait until notified or until a timeout occurs. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

This method releases the underlying lock, and then blocks until it is awakened by a notify() or notify\_all() call for the same condition variable in another thread, or until the optional timeout occurs. Once awakened or timed out, it re-acquires the lock and returns.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof).

When the underlying lock is an RLock, it is not released using its release() method, since this may not actually unlock the lock when it was acquired multiple times recursively. Instead, an internal interface of the RLock class is used, which really unlocks it even when it has been recursively acquired several times. Another internal interface is then used to restore the recursion level when the lock is reacquired.

The return value is True unless a given timeout expired, in which case it is False.

Changed in version 3.2: Previously, the method always returned None.

wait\_for(predicate, timeout=None)

Wait until a condition evaluates to true. predicate should be a callable which result will be interpreted as a boolean value. A timeout may be provided giving the maximum time to wait.

This utility method may call wait() repeatedly until the predicate is satisfied, or until a timeout occurs. The return value is the last return value of the predicate and will evaluate to False if the method timed out.

Ignoring the timeout feature, calling this method is roughly equivalent to writing:

while not predicate():

cv.wait()

Therefore, the same rules apply as with wait(): The lock must be held when called and is re-acquired on return. The predicate is evaluated with the lock held.

New in version 3.2.

notify(n=1)

By default, wake up one thread waiting on this condition, if any. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.

This method wakes up at most n of the threads waiting for the condition variable; it is a no-op if no threads are waiting.

The current implementation wakes up exactly n threads, if at least n threads are waiting. However, it’s not safe to rely on this behavior. A future, optimized implementation may occasionally wake up more than n threads.

Note: an awakened thread does not actually return from its wait() call until it can reacquire the lock. Since notify() does not release the lock, its caller should.

notify\_all()

Wake up all threads waiting on this condition. This method acts like notify(), but wakes up all waiting threads instead of one. If the calling thread has not acquired the lock when this method is called, a RuntimeError is raised.