```
/** @brief Data structure for salient point detectors.

The class instance stores a keypoint, i.e. a point feature found by one of many available keypoint
detectors, such as Harris corner detector, cv::FAST, cv::StarDetector, cv::SURF, cv::SIFT,
cv::LDetector etc.

The keypoint is characterized by the 2D position, scale (proportional to the diameter of the
neighborhood that needs to be taken into account), orientation and some other parameters. The
keypoint neighborhood is then analyzed by another algorithm that builds a descriptor (usually
represented as a feature vector). The keypoints representing the same object in different images
can then be matched using cv::KDTree or another method.
*/
class CV_EXPORTS_W_SIMPLE KeyPoint
{
public:
    //! the default constructor
    CV_WRAP KeyPoint();
    /**
    @param _pt x & y coordinates of the keypoint
    @param _size keypoint diameter
    @param _angle keypoint orientation
    @param _response keypoint detector response on the keypoint (that is, strength of the
keypoint)
    @param _octave pyramid octave in which the keypoint has been detected
    @param _class_id object id
     */
    KeyPoint(Point2f _pt, float _size, float _angle=-1, float _response=0, int _octave=0, int
_class_id=-1);
    /**
    @param x x-coordinate of the keypoint
    @param y y-coordinate of the keypoint
    @param _size keypoint diameter
    @param _angle keypoint orientation
    @param _response keypoint detector response on the keypoint (that is, strength of the
keypoint)
    @param _octave pyramid octave in which the keypoint has been detected
    @param _class_id object id
     */
    CV_WRAP KeyPoint(float x, float y, float _size, float _angle=-1, float _response=0, int _octave=0,
int _class_id=-1);
```

```
    size_t hash() const;

    /**
    This method converts vector of keypoints to vector of points or the reverse, where each
keypoint is
    assigned the same size and the same orientation.

    @param keypoints Keypoints obtained from any feature detection algorithm like SIFT/SURF/ORB
    @param points2f Array of (x,y) coordinates of each keypoint
    @param keypointIndexes Array of indexes of keypoints to be converted to points. (Acts like a
mask to
    convert only specified keypoints)
    */
    CV_WRAP static void convert(const std::vector<KeyPoint>& keypoints,
                                CV_OUT std::vector<Point2f>& points2f,
                                const std::vector<int>& keypointIndexes=std::vector<int>());
    /** @overload
    @param points2f Array of (x,y) coordinates of each keypoint
    @param keypoints Keypoints obtained from any feature detection algorithm like SIFT/SURF/ORB
    @param size keypoint diameter
    @param response keypoint detector response on the keypoint (that is, strength of the keypoint)
    @param octave pyramid octave in which the keypoint has been detected
    @param class_id object id
    */
    CV_WRAP static void convert(const std::vector<Point2f>& points2f,
                                CV_OUT std::vector<KeyPoint>& keypoints,
                                float size=1, float response=1, int octave=0, int class_id=-1);

    /**
    This method computes overlap for pair of keypoints. Overlap is the ratio between area of
keypoint
    regions' intersection and area of keypoint regions' union (considering keypoint region as circle).
    If they don't overlap, we get zero. If they coincide at same location with same size, we get 1.
    @param kp1 First keypoint
    @param kp2 Second keypoint
    */
    CV_WRAP static float overlap(const KeyPoint& kp1, const KeyPoint& kp2);

    CV_PROP_RW Point2f pt; //!< coordinates of the keypoints
    CV_PROP_RW float size; //!< diameter of the meaningful keypoint neighborhood
    CV_PROP_RW float angle; //!< computed orientation of the keypoint (-1 if not applicable);
```

```
                                        //!< it's in [0,360) degrees and measured relative to
                                        //!< image coordinate system, ie in clockwise.
    CV_PROP_RW float response; //!< the response by which the most strong keypoints have been
selected. Can be used for the further sorting or subsampling
    CV_PROP_RW int octave; //!< octave (pyramid layer) from which the keypoint has been
extracted
    CV_PROP_RW int class_id; //!< object class (if the keypoints need to be clustered by an object
they belong to)
};

template<> class DataType<KeyPoint>
{
public:
    typedef KeyPoint        value_type;
    typedef float           work_type;
    typedef float           channel_type;

    enum { generic_type = 0,
           depth        = DataType<channel_type>::depth,
           channels     = (int)(sizeof(value_type)/sizeof(channel_type)), // 7
           fmt          = DataType<channel_type>::fmt + ((channels - 1) << 8),
           type         = CV_MAKETYPE(depth, channels)
         };

    typedef Vec<channel_type, channels> vec_type;
};

/** @brief Structure containing image keypoints and descriptors. */
        struct CV_EXPORTS ImageFeatures {
            int img_idx;
            Size img_size;
            std::vector<KeyPoint> keypoints;
            VecR d;
            UMat descriptors;
        };

/** @brief Feature finders base class */
        class CV_EXPORTS FeaturesFinder {
        public:
            virtual ~FeaturesFinder() {}
            /** @overload */
            void operator()(InputArray image, ImageFeatures &features);

            /** @brief Finds features in the given image.

            @param image Source image
            @param features Found features
            @param rois Regions of interest

            @sa detail::ImageFeatures, Rect_
            */
            void operator()(InputArray image, ImageFeatures &features, const
std::vector<cv::Rect> &rois);
            /** @brief Finds features in the given images in parallel.

            @param images Source images
            @param features Found features for each image
            @param rois Regions of interest for each image

            @sa detail::ImageFeatures, Rect_
            */
            void operator()(InputArrayOfArrays images, std::vector<ImageFeatures> &features,
                            const std::vector<std::vector<cv::Rect> > &rois);
            /** @overload */
            void operator()(InputArrayOfArrays images, std::vector<ImageFeatures> &features);
            /** @brief Frees unused memory allocated before if there is any. */
            virtual void collectGarbage() {}
            /* TODO OpenCV ABI 4.x
            reimplement this as public method similar to FeaturesMatcher and remove private
function hack
            @return True, if it's possible to use the same finder instance in parallel, false
otherwise
            bool isThreadSafe() const { return is_thread_safe_; }
            */
        protected:
            /** @brief This method must implement features finding logic in order to make the
wrappers
            detail::FeaturesFinder::operator()_ work.
            @param image Source image
            @param features Found features
            @sa detail::ImageFeatures */
```

```
                    virtual void find(InputArray image, ImageFeatures &features) = 0;
                    /** @brief uses dynamic_cast to determine thread-safety
                    @return True, if it's possible to use the same finder instance in parallel, false
otherwise
                    */
                    bool isThreadSafe() const;
            };
class CV_EXPORTS SurfFeaturesFinder : public FeaturesFinder {
            public:
                    SurfFeaturesFinder(double hess_thresh = 300., int num_octaves = 3, int num_layers =
4,
                                    int num_octaves_descr = /*4*/3, int num_layers_descr =
/*2*/4);
            private:
                    void find(InputArray image, ImageFeatures &features);

                    Ptr<FeatureDetector> detector_;
                    Ptr<DescriptorExtractor> extractor_;
                    Ptr<Feature2D> surf;
            };
SurfFeaturesFinder::SurfFeaturesFinder(double hess_thresh, int num_octaves, int num_layers,
                                                    int num_octaves_descr, int
num_layers_descr) {
#ifdef HAVE_OPENCV_XFEATURES2D
                    if (num_octaves_descr == num_octaves && num_layers_descr == num_layers)
                    {
                          Ptr<SURF> surf_ = SURF::create();
                          if( !surf_ )
                                CV_Error( Error::StsNotImplemented, "OpenCV was built without SURF
support" );
                          surf_->setHessianThreshold(hess_thresh);
                          surf_->setNOctaves(num_octaves);
                          surf_->setNOctaveLayers(num_layers);
                          surf = surf_;
                    }
                    else
                    {
                          Ptr<SURF> sdetector_ = SURF::create();
                          Ptr<SURF> sextractor_ = SURF::create();

                          if( !sdetector_ || !sextractor_ )
                    CV_Error( Error::StsNotImplemented, "OpenCV was built without SURF
support" );

                sdetector_->setHessianThreshold(hess_thresh);
                sdetector_->setNOctaves(num_octaves);
                sdetector_->setNOctaveLayers(num_layers);

                sextractor_->setNOctaves(num_octaves_descr);
                sextractor_->setNOctaveLayers(num_layers_descr);

                detector_ = sdetector_;
                extractor_ = sextractor_;
            }
#else
            CV_UNUSED(hess_thresh);
            CV_UNUSED(num_octaves);
            CV_UNUSED(num_layers);
            CV_UNUSED(num_octaves_descr);
            CV_UNUSED(num_layers_descr);
            CV_Error(Error::StsNotImplemented, "OpenCV was built without SURF support");
#endif
        }

        void SurfFeaturesFinder::find(InputArray image, ImageFeatures &features) {
            UMat gray_image;
            CV_Assert((image.type() == CV_8UC3) || (image.type() == CV_8UC1));
            if (image.type() == CV_8UC3) {
                cvtColor(image, gray_image, COLOR_BGR2GRAY);
            } else {
                gray_image = image.getUMat();
            }
            if (!surf) {
                detector_->detect(gray_image, features.keypoints);
                extractor_->compute(gray_image, features.keypoints, features.descriptors);
            } else {
                UMat descriptors;
                surf->detectAndCompute(gray_image, Mat(), features.keypoints, descriptors);
                features.descriptors = descriptors.reshape(1, (int) features.keypoints.size());
            }
        }
```