Automatic Procedure for Accurate and Seamless Reprojection of Raster Images from the Finnish KKJ Coordinate System to ETRS-TM35FIN

Jukka Rahkonen[1,2] and Kimmo Lahtinen[2]

[1] Department of Agrotechnology
University of Helsinki, Finland
Current address:
[2] Information Centre of the Ministry of Agriculture and Forestry
P.O. Box 310, 00023 Government, Finland
jukka.rahkonen@mmmtike.fi

Abstract. According to the national recommendation from year 2003, a new reference frame EUREF-FIN which is based on the European ETRS89 system.will gradually substitute for the Finnish grid coordinates system "Kartastokoordinaattijärjestelmä KKJ" as a preferred spatial reference system in Finland. Unfortunately the KKJ system has local distortions which are leading to 0-2 metre projection errors if common 7-parameter affine transformation is used with same parameters for the whole country. For better accuracy local distortions must be taken into account. This study describes a method for eliminating local distortions in raster image reprojection by generating artificial ground control points with external reference software. In addition, a Python script automating the process by combining raster processing library GDAL and coordinate conversion program YKJETRS is published. The script has an additional feature of fitting reprojected images to a common canvas coordinate system, which enables to join even independently processed images without visible seams at image boundaries.

1 Introduction

The Finnish Ministry of Interior has given an official recommendation for the Finnish coordinate systems in June, 2003. The new preferred coordinate system is the common European reference frame ETRS, and preferred projection for the state wide use is called ETRS-TM35FIN. This projection is otherwise identical to UTM-35N projection, but adopted so that it can be widened beyond normal UTM zone width of 6 degrees to cover the whole Finnish territory.

The recommendation means a gradual change from the use of national grid coordinates system to ETRS-TM35FIN. The old system, often referred to as "Finnish KKJ" or "Kartastokoordinaattijärjestelmä", is a Gauss-Krüger projection with Hayford ellipsoid. In the KKJ system Finland is divided into 3 degrees wide projection zones with centre meridians at 18, 21, 24, 27, 30 and 33 degrees eastern longitude. The zones are named as KKJ zones 0, 1, 2, 3, 4 and 5, respectively from

west to east. False easting values of 500000 m, 1500000 m, 2500000 m, 3500000 m, 4500000 m and 5500000 m are added to zone coordinates [1]. Therefore the correct zone for a dataset in KKJ coordinate system is usually simple to recognise. Sometimes checking the coordinate values is also the only chance, because only KKJ zones 1-4 used to have registered EPSG-codes (2391, 2392, 2393 and 2394, respectively). At the moment also zones 0 and 5 have own their codes in the EPSG database (3386 and 3387), but most geospatial software miss the support for them.

Adopting the new coordinate system will lead to a common need to reproject spatial data between KKJ and ETRS-TM35FIN projections. The need will be long lasting because some part of the old data will probable remain in KKJ. Unfortunately the KKJ system has local distortions, which means that a common method of using affine transformation does not give accurate results in all parts of Finland The residual error when the optimal parameters for the 7-parameter affine transformation are used varies from 0-2 metres, an average being about 0.5 m (Fig. 1).

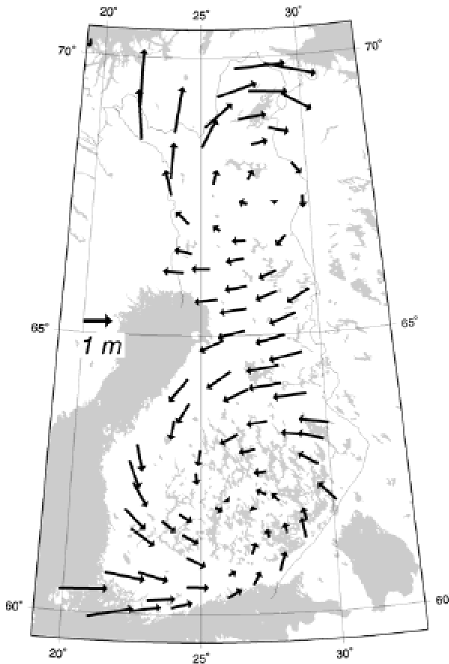

Fig. 1. Residual errors in reprojection from KKJ to ETRS-TM35FIN when the optimal parameters for the 7-parameter transformation are used nationwide [1]

The official recommendation [2] presents a method for eliminating most part of local distortions. The method is based on affine triangles and recommendation gives formulas leading to less than one metre residual errors for the whole country (Fig. 2).

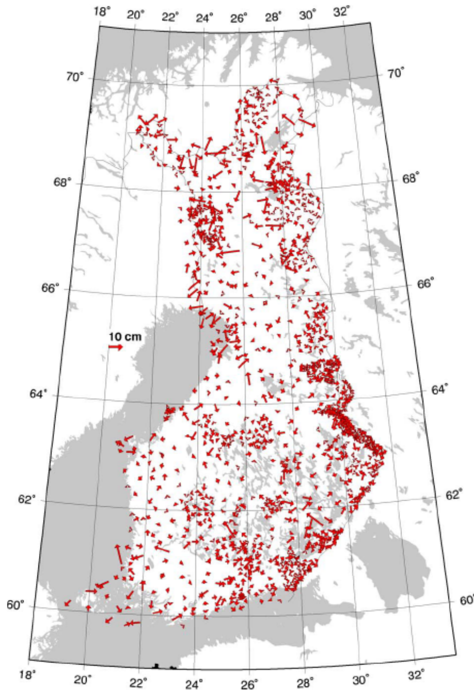

Fig. 2. . Residual errors in reprojection from KKJ to ETRS-TM35FIN when locally adjusted formulas are used [2]

National Land Survey of Finland (NLS) has made conversion programs [3] which are based on these formulas and accurate method has also been integrated to some common GIS software packages. Thus accurate reprojection of vector data between KKJ and EUREF systems is already rather well supported. However, this is not the case with raster data, but tools for automatic handling of local distortions of the KKJ system are mostly missing.

This paper describes a method which is utilising the NLS coordinate transformation program for calculating reference coordinates, which are then used as ground control points (GCP). After GCP creation the raster image can be reprojected to ETRS-TM35FIN projection with any image processing software that do transformation by using ground control points. As a result the location error caused by reprojection will be limited close to the situation presented in Fig. 2.

2 Principle of Accurate and Seamless Raster Reprojection between Distorted Coordinate Systems

The basic idea of performing accurate reprojection between KKJ and ETRS-TM35FIN projections is simple. The same method can be applied as well to any other two projection systems having local distortions. First demand is that the source raster image is georeferenced, so that the coordinate values in source spatial reference system (SRS) can be defined for any pixel of the image. The other requirement is that there exists some accurate method for converting coordinates from the source SRS to target SRS. In the case described in this paper accurate coordinate transformation from KKJ to ETRS-TM35FIN is done with the YKJETRS utility [3].

First step in the conversion is to determine the coordinates of an individual image pixel in source SRS units. Next the source coordinates are transformed to target SRS with best possible accuracy, and resulting coordinates are fed back to the source image as ground control points (GCP). Finally the source image, now containing accurate GCPs, is warped to target projection with some image processing software.

Basically any image pixel can be selected for GCP. However, image corner pixels are to be preferred because using them prevents extrapolation beyond GPC locations in the warping process. In addition, if the original images are cut to match side by side, the use of image corner pixels effectively means that the same GCPs are used for neighbouring images. This will guarantee precise match also after warping, even if the image area may not be exactly rectangular any more.

When adjacent images are reprojected individually and joined back together after warping, a visible seam like dashed line often appears at the image boundaries (Fig. 3). This happens even if he original images in the source SRS matched without any seams. However, no seams will appear if the images are mosaiced together while still at source SRS and the resulting image mosaic is warped into target SRS. Therefore it is obvious that visible seams at image boundaries are not coming from the warping itself but they are caused by some other reason.

Fig. 3. Mosaic of two individually warped images showing the typical dashed line artifact at the image boundary

The difference between adjacent images in source SRS, images which are warped to target SRS in company, and images warped to target SRS individually is that while in the two first cases images automatically share the common canvas coordinate system, in the last case they not necessarily do. This is illustrated in the Fig 4.

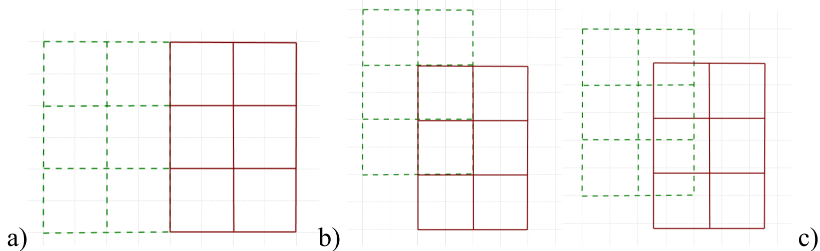

Fig. 4. Two images are sharing the same canvas coordinate system in cases a) and b). The images in c) do not share the same canvas, which is likely to give artifacts at image boundaries if images are show together on screen or mosaiced together.

Normally a batch of geospatial images is processed so that the pixels of adjacent images are aligned. The images are sharing the same canvas coordinate system, which means that pixel rows and columns continue from one image to neighbouring one linearly like in case *a* in Fig 4. This situation often changes in reprojection, because reprojecting software usually does not care about the canvas coordinate system when individual images are warped. The software just calculates the exact extents of the minimum bounding box of the resulting image, and sets the image origin accordingly. For this reason it is almost sure that individually reprojected adjacent images do not anymore have their origins and pixels in the same canvas coordinate system (case *c* in Fig 4). However, a new common canvas coordinate system must be created if warped images are to be shown together on computer screen or if they will be mosaiced together. This means that at least one of the adjacent images must be resampled, which occasionally leads to situation where images from the both sides of the seam are giving no-data value for the pixel at the seam. The result can be seen as a dashed line on the combined image.

There is no need to do preliminary mosaics of source images, perform feathering at seams of the resulting images, or warp original images with overlaps in order to avoid visible seams between reprojected images. All that is needed is to force also the individually warped images to share a common canvas coordinate system (case *b* in Fig. 4). The effect of the procedure can be seen in the Fig. 5. In practice forcing images to use the same canvas means that only multiples of the pixel size of the warped image are accepted as the x- and y-values of the image origin and pixel sizes of the images to be joined must suit each other. The areas in the resulting images which are not part of the original image must be handled as transparent in join, which can be easier if no-data value is used in image creation. In addition, to avoid loosing original image data the extents of the warped image must be set wider in each direction than the minimum bounding box.

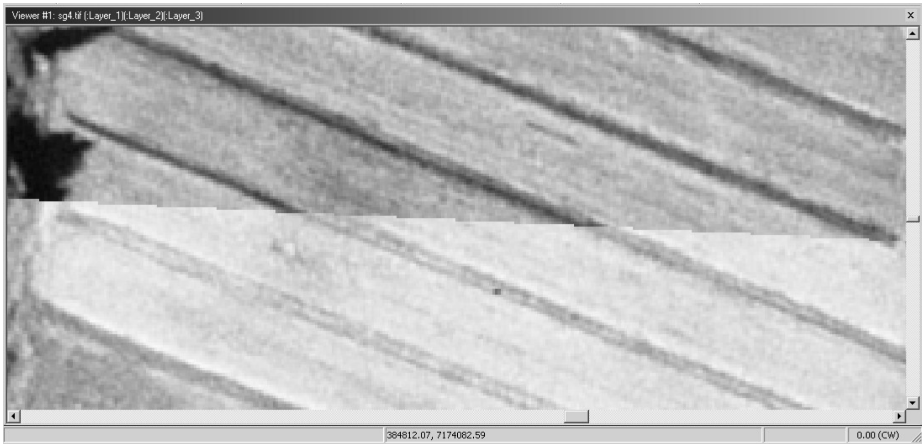

Fig. 5. Mosaic made from two images which are warped individually but forced to share the common canvas coordinate system

3 Execution of the Accurate KKJ - ETRS-TM35FIN Raster Reprojection Step by Step

In the following chapters the procedure of accurate reprojection is described in details by using GDAL executable programs as an example. GDAL is used as a reference; the method itself does not involve the use of GDAL.

The test image is *t202112.jp2.*

3.1 Acquiring Image Information

The necessary information about coordinate system, image size, pixel size and image orientation can be obtained by using 'gdalinfo' utility [3]

Excerpt of the gdalinfo listing:

```
gdalinfo t202112.jp2
```

Driver: JP2KAK/JPEG-2000 (based on Kakadu)
Size is 10000, 10000
Coordinate System is `'
Origin = (2450000.000000,6710000.000000)

Pixel Size = (1.00000000,-1.00000000)
Corner Coordinates:
Upper Left (2450000.000, 6710000.000)
Lower Left (2450000.000, 6700000.000)
Upper Right (2460000.000, 6710000.000)
Lower Right (2460000.000, 6700000.000)
Center (2455000.000, 6705000.000)

Coordinate system is not set for this image, but we know that the image is in KKJ system and by the coordinate values we can conclude that it must be in KKJ zone 2.

3.2 Connecting Pixel Coordinates with Geographical Coordinates

In later steps we need to know both the pixel coordinates of the image and corresponding geographical coordinates. In this example we need to know also that GDAL starts counting pixel coordinates from (0,0). Moreover, GDAL is using the upper left corner as exact reference point for pixels. Therefore the corner coordinates reported on the list above must be corrected with pixel height and width values in order to get the list of "Corner pixel upper left coordinates".

Table 1. Upper left coordinates of the corner pixels of the original image in KKJ2

GPC	pixel-x	pixel-y	geo-x	geo-y
UpperLeft	0	0	2450000,000	6710000.000
LowerLeft	9999	0	2450000,000	6700001.000
UpperRight	0	9999	2459999.000	6710000.000
LowerRight	9999	9999	2459999.000	6700001.000

3.3 Intermediate Transformation to KKJ Zone 3

Geographical coordinates are now in KKJ zone 2, but the conversion program to be used later can only do transformations from KKJ zone.3 Therefore we must first calculate the transformation from KKJ2 to KKJ3 with 'cr2cr' utility [4]

The command to be used has a format:

```
cs2cs -v +init=epsg:2392 +to +init=epsg:2393 [files]
```

Table 2. Upper left coordinates of the corner pixels of the original image in KKJ3

GPC	pixel-x	pixel-y	geo-x	geo-y
UpperLeft	0	0	3285196.010	6716036.750
LowerLeft	9999	0	3284740.250	6706042.780
UpperRight	0	9999	3295189.760	6715580.800
LowerRight	9999	9999	3294734.040	6705587.230

3.4 Accurate Conversion with YKJETRS utility

The next step is to convert KKJ3 zone coordinates to accurate ETRS-TM35FIN coordinates with YKJETRS utility by the National Land Survey of Finland [3].

The command to be used is:

```
ykjetrs ykj TM35 <North> <East>
```

Table 3. Upper left coordinates of the corner pixels of the original image in ETRS-TM35FIN

GPC	pixel-x	pixel-y	geo-x	geo-y
UpperLeft	0	0	285113,632	6713217,584
LowerLeft	9999	0	284658,101	6703227,690
UpperRight	0	9999	295103,322	6712761,873
LowerRight	9999	9999	294647,828	6702772,378

3.5 Inserting Ground Control Points to Image

The original JPEG2000 image file is converted to GeoTIFF format and corner pixel coordinates in ETRS-TM35FIN system are added to the file as ground control points with gdal_translate utility [3].

The command to be used is:

```
gdal_translate t202112.jp2 t202112_gcp.tif -gcp 0 0
285113.632 6713217.584 -gcp 9999 0 284658.101
6703227.690 -gcp 0 9999  295103.322 6712761.873 -gcp
9999 9999 294647.828 6702772.378
```

3.6 Calculating Widened Extents to Enable Seamless Joins

The new GeoTIFF file *t202112_gcp.tif* is now ready to be warped. However, we do not want to get a result image with occasional extents but we want to force it to use canvas coordinates rounded to full meters. This ensures that the resulting image can be joined together with other images processed in the same way without visible seams. Therefore the extents of the image are widened in each direction to next full metres and the transformation parameter is set to make an image with 1 metre pixel size. For achieving seamless join the canvas coordinate grid size must suit the image pixel size. Canvas grid size does not need to be the same than pixel size but multiplies of pixel size can be used as well.

Table 4. Widened extents

	minX	**minY**	**maxX**	**maxX**
Calculated values	284658.101	6702772.378	295103.322	6713217.584
Widened values	284658.000	6702772.000	295104.000	6713218.000

3.7 Warping the Image

Final step is to perform the transformation with gdalwarp utility with command:

```
gdalwarp  -rc -dstnodata "0 0 0" -s_srs EPSG:3067 -
t_srs EPSG:3067 -tr 1.0 1.0 -te 284658 6702772 295104
6713218 t202112_gcp.tif t202112_TM35FIN.tif
```

Explanation for the parameters used:

- **-rc** **U**ses cubic convolution method for resampling, which was giving the best image quality with the sample image.
- **-dstnonata "0 0 0"** Adds information that pixel value 0,0,0 (black) stands for no-data, area that is outside of the original image area. This setting helps some software in making seamless joins.
- **-s_srs, t_srs EPSG:3067** Informst the utility that both the source image and the target image are in ETRS-TM35FIN coordinate system. By giving the ground control points in ETRS-TM35FIN coordinates we have discarded the original KKJ2 projection.
- **-tr 1.0 1.**0 Sets target resolution as 1.0 metre in both x- and y- directions, which suits the resolution of the original image as well as the canvas grid we have in mind.
- **-te [values]** Sets the extents of the resulting image to the values which were calculated in phase 3.6.

Gdalinfo can be used to list the capabilities of the result image.

```
gdalinfo t202112_tm35fin.tif
```

Driver: GTiff/GeoTIFF
Size is 10446, 10446
Coordinate System is:
PROJCS["ETRS89 / ETRS-TM35FIN",
...
...
Corner Coordinates:
Upper Left (284658.000, 6713218.000) (23d 4'43.40"E, 60d29'50.42"N)
Lower Left (284658.000, 6702772.000) (23d 5'24.01"E, 60d24'13.56"N)
Upper Right (295104.000, 6713218.000) (23d16'6.44"E, 60d30'10.03"N)
Lower Right (295104.000, 6702772.000) (23d16'45.09"E, 60d24'33.10"N)
Center (289881.000, 6707995.000) (23d10'44.73"E, 60d27'11.90"N)
Band 1 Block=10446x1 Type=Byte, ColorInterp=Red
NoData Value=0
Band 2 Block=10446x1 Type=Byte, ColorInterp=Green
NoData Value=0
Band 3 Block=10446x1 Type=Byte, ColorInterp=Blue
NoData Value=0

The seven steps described in this chapter have yielded a reprojected image *t202112_TM35FIN.tif* which has its corner pixels transformed from KKJ system to ETRS-TM35FIN system as accurately as YKJETRS program makes the transformation. Transformation of the pixels between the GCPs is following a linear formula. There is a potential source for error in the transformation on the intermediate image area if the distortion of the KKJ system is not changing in a linear way. The image that was converted here is only 5000 metres wide and there is a good reason to consider KKJ distortions to be close to linear at this distance (see Fig. 1). However, additional GCPs can be simply calculated with the same procedure.

4 Python Script for Automatic, Accurate and Seamless Reprojection from KKJ to ETRS-TM35FIN

For automating the procedure described in chapter 3 we wrote a Python script to tie all the steps together. The script is utilising GDAL libraries, YKJETRS executable program and *gdalwarp* executable. A simple way to make a working installation on a

computer is to install FWTools package [5] and put both the WARPTOFIN.py script and YKJETRS.exe program to FWTools main directory.

The utility is extremely simple to use. The user starts the utility by giving a following command:

```
python warptofin.py <input file> <virtual file>
<output file>
```

The utility is thereafter performing automatically the following procedure:

1. The coordinate system of the input image is interpreted. If the image has EPSG code 2391-2394 for some of the KKJ zones 1-4 in its GeoTIFF tags, that code is selected. Otherwise the zone is interpreted by the image extents. In this phase GDAL libraries are utilised.
2. Corner coordinates of the original image are converted to coordinates of the upper left corner of the corner pixel, then to KKJ zone 3 coordinates and finally to ETRS-TM35FIN coordinates by using GDAL libraries and YKJETRS executable program.
3. ETRS-TM35FIN coordinates are transferred as ground control points to a GDAL virtual raster dataset (VRT). This virtual dataset is a simple text file that is referring to the original image file. By using VRT file as an intermediate format that contains ground control points the writing of a physical GeoTIFF file of size 300 MB can be skipped. This phase is using DGAL libraries directly.
4. The extents of the resulting image are widened to next full metre in each direction by a program code written inside Python script.
5. Parameters are passed to the GDALWARP utility that is performing the warping to ETRS-TM35FIN projection. The script is generating some parameters automatically, and some parameters are set manually as constants inside the Python code. No user interface was built around the utility but it can only be used from command line. In normal case there is no need to change the program code. Because Python code is text based, the editing hard coded parameters inside the Python script is not very difficult.
6. Information about the transformation, including names of the source and target files and times of execution are written into log file

5. Evaluation and conclusions

For evaluating the efficiency of the utility we were measuring execution time for the following test:

Original three-channel image of size 10000 by 10000 pixels (300 MB) in GeoTIFF format and in KKJ2 coordinate system was warped to ETRS-TM35FIN projection with *warptofin* utility. Pixel size was 0.5 metres in both images. The computer used

for the test was running on Windows XP Professional SP2 operating system and it had one Intel Xeon 3.0 GHz processor. Amount of central memory was 1 GB. Hard disks on the computer were of ATA-133 type. In this test the total time needed to warp one image was 3 minutes and 10 seconds. After that *warptofin* utility has been used for transforming a total amount of 4 terabytes of aerial orthophotos from KKJ to EUREF-TM35FIN and it has proved to be very reliable.

It has to be noted that the residual error in affine transformation from KKJ to ETRS-TM35FIN is at maximum around 2 meters and usually less. The improved accuracy obtained by using YKJETRS utility is not significant unless the original imagery is both geometrically accurate and with small enough pixel size. As a rule of thumb, affine transformation can well be used if the pixel size of the image is more than 2-5 meters, or if the location error of the original imagery, expressed as Root mean square error (RMSE) is more that 2-5 meters. However, even if the better accuracy is not needed the Python script with automatic source reference system recognition and seamless join feature might be useful for less accurate imagery as well.

Acknowledgements

The authors would like to thank Mr. Lassi Lehto and Mr. Jaakko Kähkönen from the Finnish Geodetic Institute for the basic idea of using ground control points generated by YKJETRS program for accurate raster image warping to ETRS-TM35FIN.

References

1. JHS 153 ETRS89 coordinates in Finland
   http://www.jhs-suositukset.fi/intermin/hankkeet/jhs/home.nsf/files/JHS153/$file/JHS153.pdf
2. JHS 154 Map projections, plane coordinates and map sheet index for ETRS89 in Finland
   http://www.jhs-suositukset.fi/intermin/hankkeet/jhs/home.nsf/files/JHS154/$file/JHS154.pdf
3. EurefMuunnos and YKJETRS programs. Available on request from the National Land Survey of Finland
4. GDAL and OGR libraries, available from http://www.gdal.org
5. FWTools binaries, available from http://fwtools.maptools.org

Annex 1 Program code

Script made by Kimmo Lahtinen and Jukka Rahkonen. Published under the same conditions than GDAL libraries and YKJETRS executable.

```
#import pyproj

    import gdal
    from gdalconst import *
    import osr
    import sys
    import os
    import re
    import math
    import time

    GDALWARP = "bin\\gdalwarp.exe"
    YKJETRS = "ykjetrs.exe"

    # Gets the EPSG code from the projection WKT (well known text),
    # !Not a very robust method, just gets the last epsg from the text!
    def get_epsg(dataset):
            pattern = re.compile(".*EPSG.*\"(.*)\"")
            match = pattern.search(dataset.GetProjection())
            if match != None:
                    return match.group(1)
            else:
                    return None

    def get_corner_pixel_positions(dataset):
            gtrn = dataset.GetGeoTransform()
            x_extent = gtrn[1] * (dataset.RasterXSize - 1);
            y_extent = gtrn[5] * (dataset.RasterYSize - 1);

            return [
                    (gtrn[0], gtrn[3]),                             # upper left
                    (gtrn[0], gtrn[3] + y_extent),                  # lower left
                    (gtrn[0] + x_extent, gtrn[3]),                  # upper right
                    (gtrn[0] + x_extent, gtrn[3] + y_extent),       # lower right
            ]

    BASE_PROJ = "+proj=tmerc +lat_0=0 +k=1.000000 +y_0=0 +ellps=intl
    +towgs84=-90.7,-106.1,-119.2,4.09,0.218,-1.05,1.37 +units=m +no_defs"

    COORD_DICT = {
            "kkj0" : BASE_PROJ + " +lon_0=18 +x_0=500000",
            "kkj1" : BASE_PROJ + " +lon_0=21 +x_0=1500000",
            "kkj2" : BASE_PROJ + " +lon_0=24 +x_0=2500000",
            "kkj3" : BASE_PROJ + " +lon_0=27 +x_0=3500000",
            "kkj4" : BASE_PROJ + " +lon_0=30 +x_0=4500000",
            "kkj5" : BASE_PROJ + " +lon_0=33 +x_0=5500000",
    }
```

```
EPSG_TO_PROJ = {
        2391 : COORD_DICT["kkj1"],
        2392 : COORD_DICT["kkj2"],
        2393 : COORD_DICT["kkj3"],
        2394 : COORD_DICT["kkj4"],
}

def guess_coord_system(easting):

        res = ""
        for i in range(0,6):
                if (i*1000000) <= easting and easting < ((i+1)*1000000):
                        res = COORD_DICT["kkj" + str(i)]
                        break
        return res

# transforms different kkj slice coordinates to ykj
def to_ykj(dataset, coord_list):

        source = osr.SpatialReference()
        target = osr.SpatialReference()
        target.ImportFromProj4(EPSG_TO_PROJ[2393])

        epsg_str = get_epsg(dataset)

        if epsg_str != None:
                epsg = int(epsg_str)
                if epsg == 2393:
                        return coord_list

                if not EPSG_TO_PROJ.has_key(epsg):
                        print "Error: " + epsg + " not supported as source coordinate
system."
                        return None

                source.ImportFromProj4(EPSG_TO_PROJ[epsg])
        else:
                proj4_def = guess_coord_system(coord_list[0][0])

                if proj4_def == "":
                        print "Error: unrecognized coordinate system in source picture."
                        return None

                source.ImportFromProj4(proj4_def)

        transformer = osr.CoordinateTransformation(source, target)

        res = []
        for x_y in coord_list:
                ykj_coord = transformer.TransformPoint(x_y[0], x_y[1])
                res.append((ykj_coord[0], ykj_coord[1]))

        return res

# transfroms coordinates from ykj to tm35fin
def ykj_to_tm35(coord_list):

        # open pipes to ykjetrs util
```

```
		stdin,out,err = os.popen3((YKJETRS + " ykj tm35"), "t")

		# feed coordinate list to the program
		for coord in coord_list:
				# reverse coordinate order (ykjetrs takes northing first)
				stdin.write("a %f %f\n" % (coord[1], coord[0]))
		stdin.close()

		# parse the result

		result = []

		for line in out:
				split = line.split()
				result.append((float(split[2]), float(split[1])))

		status = out.close()

		if status != None:
				print ("Error: ykjetrs exit status " + str(status))
				return []

		return result

	# creates a vrt file from a given database and inserts the given
	# coordinates to it as ground control points
	def create_temp_vrt_file(temp_filename, dataset, transformed_coords):

		driver = gdal.GetDriverByName("VRT")
		newds = driver.CreateCopy(temp_filename, dataset)

		srs = osr.SpatialReference()
		srs.ImportFromEPSG(3067)
		newds.SetProjection(str(srs))

		ref_pixels = [
				(0,0),
				(0, dataset.RasterYSize-1),
				(dataset.RasterXSize-1, 0),
				(dataset.RasterXSize-1, dataset.RasterYSize-1)
		]

		ref_ids = [ "UpperLeft", "LowerLeft", "UpperRight", "LowerRight" ]

		gcps = []

		for i in range(0,4):
				gcp = gdal.GCP()
				gcp.GCPX = transformed_coords[i][0]
				gcp.GCPY = transformed_coords[i][1]
				gcp.GCPPixel = ref_pixels[i][0]
				gcp.GCPLine  = ref_pixels[i][1]
				gcp.Id = ref_ids[i]

				gcps.append(gcp)

		newds.SetGeoTransform(gdal.GCPsToGeoTransform(gcps))
	#	newds.SetGCPs(gcps)
```

```
        newds.FlushCache()

# get the minimun and maximum coordinates from the given
# coordinate array
def get_minmax(coords):
        def helper(i, func):
                def ith(s): return s[i]
                return func(map(ith, coords))
        return [
                (helper(0, min), helper(1, min)),
                (helper(0, max), helper(1, max))
        ]

# runs the gdalwarp command to transform the picture based on
# the ground control points int the source file
def warp(fromfile, tofile, output_extent):
        oe = output_extent

        cmd = GDALWARP
#       cmd += " -multi"
        cmd += " -rc"
        cmd += " -dstnodata \"0 0 0\""
        cmd += " -s_srs EPSG:3067"
        cmd += " -t_srs EPSG:3067"
        cmd += " -co TFW=YES"
        cmd += " -tr 0.5 0.5"
#       cmd += " -tr 50.0 50.0"
        cmd += (" -te %d %d %d %d" % (oe[0][0], oe[0][1], oe[1][0], oe[1][1]))
        cmd += " " + fromfile + " " + tofile

#       print cmd
        status = os.system(cmd)

        if status > 0:
                print ("Error: gdalwarp exit status " + str(status))

def log_clear(outfilename):
        log = open(outfilename + ".log", "w")
        log.close()
def log(text, outfilename):
        log = open(outfilename + ".log", "a")
        log.write(text + "\n")
        log.flush()
        log.close()

def doit(filename, vrtfilename, outfilename):

        print "Converting " + filename
        log_clear(outfilename)
        log("Starting at: " + time.ctime(), outfilename)
        log("In file:" + filename, outfilename)
        log("VRT file: " + vrtfilename, outfilename)
        log("Result file:" + outfilename, outfilename)

        dataset = gdal.Open(filename, GA_ReadOnly)

        # the upper left pos of each 4 corner pixels in the original image
        corner_coords = get_corner_pixel_positions(dataset)
```

```
	log("Source coordinates (UL,LL,UR,LR): " + str(corner_coords), outfilename)

	if corner_coords[0][0] <= 0:
		print "Error: Picture " + filename + " did not contain proper coordinates."
		return

	# transform the corners to ykj
	corner_coords = to_ykj(dataset, corner_coords)

	log("YKJ coords (UL,LL,UR,LR): " + str(corner_coords), outfilename)

	if corner_coords == None:
		return

	# transform the corners to tm35fin
	tm32_coords = ykj_to_tm35(corner_coords)

	log("tm35fin coords (UL,LL,UR,LR): " + str(tm32_coords), outfilename)

	# creates a .vrt file with transformed coordinates from the given image
	create_temp_vrt_file(vrtfilename, dataset, tm32_coords)

	minmax = get_minmax(tm32_coords)
	minmax_wider = [
		(int(math.floor(minmax[0][0])), int(math.floor(minmax[0][1]))),
		(int(math.ceil (minmax[1][0])), int(math.ceil (minmax[1][1]))),
	]

	log("Extents (min,max): " + str(minmax), outfilename)
	log("Widened extents (min,max): " + str(minmax_wider), outfilename)

#	print corner_coords
#	print minmax
#	print minmax_wider

	# transform the .vrt to an actual file with gdalwarp
	warp(vrtfilename, outfilename, minmax_wider)

	log("Finished at: " + time.ctime(), outfilename)

### startup ###

#print to_ykj(None, [(2450000.000, 6700000)])

if len(sys.argv) < 4:
	print "Need 3 command line parameters (in.???, temp.vrt, out.tif)"
else:
	doit(sys.argv[1], sys.argv[2], sys.argv[3])
	#doit("..\\euref_test50.tif", "..\\test.vrt", "..\\out.tif")
```