Spatial Vision, Vol. 11, No. 1, pp. 107–115 (1997)

The IPRS Image Processing and Pattern Recognition System

TERRY CAELLI,[1,*] CRAIG DILLON,[1,**] EROL OSMAN[2,†] and GERHARD KRIEGER[2,‡]

[1] *Department of Computer Science, Curtin University of Technology, Perth, Western Australia 6001, Australia*
[2] *Institut fur Medizinische Psychologie, Ludwig-Maximilians-Universität, Goethestr. 31, 80336 München, Germany*

Received 7 February 1997; revised 18 March 1997; accepted 18 March 1997

Abstract—IPRS is a freely available software system which consists of about 250 library functions in C, and a set of application programs. It is designed to run under UNIX and comes with full source code, system manual pages, and a comprehensive user's and programmer's guide. It is intended for use by researchers in human vision, pattern recognition, image processing, machine vision and machine learning.

IPRS (Image and Pattern Recognition System) is a software system intended for use by researchers and students in human vision, pattern recognition, image processing, machine vision and machine learning. It is designed to cater for most needs in building complete image interpretation systems, as it spans from low-level image processing functions to high-level complex technologies for clustering image content and abstract data, pattern recognition and modelling of visual function.

The development of the IPRS library was initiated by one of us (T. C.) in 1991 to facilitate the transfer of software between research groups involved in vision research, motivated by previous work by this author in psychophysics and modeling of human visual function. Such transfer can significantly ease the task of comparing techniques and results from different vision-research groups.

IPRS is designed to run under UNIX and was initially developed on Silicon Graphics workstations. It was then ported to other platforms, so that it can now also be installed on DEC stations, Suns, and PCs running Linux. It is available as C source code, and

*tmc@cs.curtin.edu.au

**cdillon@cs.curtin.edu.au

†erol@imp.med.uni-muenchen.de (to whom correspondence should be addressed).

‡krieger@imp.med.uni-muenchen.de

ftp://ftp.cs.curtin.edu.au/pub/iprs/iprs

the main requirements for installing the IPRS software are the GNU C-Compiler (gcc) and GNU make (gmake).

IPRS comes in two basic forms: as a library of C functions and as a set of Unix application programs. Of the application programs, many are designed to mimic the C library functions as closely as possible. These applications allow fast and versatile combination of image processing functions by using the UNIX shell feature of pipelining the output of one program to the input of the next. For instance, the combined effect of several image processing functions with varying parameters can be visualized 'on the fly'; this can be helpful for selecting an appropriate stimulus in a psychophysical experiment. See the Appendix for an example.

IPRS also contains a number of larger application programs which serve to illustrate the development of complete systems for object recognition and texture classification under the IPRS framework. An example is the evidence-based object recognition system (Caelli and Dreier, 1994), which combines IPRS functions for segmentation, feature extraction, clustering and neural networking to form a robust visual object-recognition system. The evidence-based system has also been successfully used for modelling results from human pattern learning and recognition experiments (Jüttner *et al.*, 1996). Other examples for the application of IPRS can be seen in Dance and Caelli (1993), and Dillon and Caelli (1995).

A number of machine-dependent application programs allow for image displaying, editing, printing, and other functions. On systems equipped with X11 in combination with the Motif runtime library, the construction of graphical user interfaces is supported. Such interfaces may facilitate the user's communication with the functions provided by IPRS, as can be seen from a simple demonstration program which is provided with the IPRS package.

Nevertheless, the main part of IPRS is the extensive set of about 250 C library functions, which support a wide range of data types and allow arbitrary-dimensional data structures. There is no limitation to 2- or 3-dimensional structures since higher dimensionality is useful, for instance, for representing feature spaces. The data types available for images cover the usual Cartesian types, such as boolean, byte, integer, float, complex and rgb. Additionally, abstract data structures are supported such as edge or region descriptions, feature spaces, rule and neural network descriptions. These are paricularly suited for implementing higher level functions such as segmentation, feature extraction, and rule generation designed for image understanding, machine learning and pattern recognition.

The library functions cover the following classes of operations:

- low-level image processing, filtering, Fast Fourier transforms, scaling, thresholding, histogram equalisation, edge detection and others;
- image compression, including predictive coding, causal and non-causal filtering, curvature and pyramidal structures;
- image segmentation;
- feature extraction from arbitrary-dimensional data;
- clustering and automatic rule generation in arbitrary-dimensional feature spaces, including clustering and sub-clustering, and hierarchical rule structures;

- graph matching;
- neural networks and other learning paradigms;
- machine-dependent functions for displaying, printing, etc.

For almost every C library function and application program, a system manual entry is available. Further, there exists a printable manual of about 300 pages which covers most of IPRS and includes issues such as programming guidelines for developers and a general description of data structures. The manual is available in Postscript format at the ftp site mentioned earlier.

The IPRS library can easily be extended by providing functions that meet individual requirements. For example, a package for nonlinear image processing based on the Volterra–Wiener series has been developed by using the IPRS feature of arbitrary-dimensional image structures. The calculations in Krieger and Zetzsche (1996) were performed with this extended IPRS software.

The complete IPRS system, including all sources and manuals, is freely available via anonymous ftp at the previously mentioned site. The authors who have contributed to IPRS make no claims on the reliability or appropriateness of algorithms, code, or application programs contained within IPRS. Currently, no support is available.

Acknowledgements

IPRS has been developed in a spirit of free exchange of code and ideas. Many people have contributed to the IPRS system: E. Barth, J. Cooper, A. Dreier, M. Ollila, D. Squire, T. Wild, C. Zetzsche, the authors of this article, and many more. We thank H. Strasburger for valuable comments on the manuscript. This work has been supported by DFG grant Re337/7 and the Stiftung für Bildung und Behindertenförderung, Stuttgart.

REFERENCES

Caelli, T. and Dreier, A. (1994). Variations on the evidence-based object recognition theme. *Patt. Recogn.* **27**, 185–204.

Dance, S. and Caelli, T. (1993). A symbolic object-oriented picture interpretation network: SOO–PIN. In: *Advances in Structural and Syntactic Pattern Recognition, Proceedings of the International WorkShop*. H. Bunke (Ed.). World Scientific Publishing Co., Bern, Switzerland, pp. 530–541.

Dillon, C. and Caelli, T. (1995). Cite: A scene understanding and object recognition system. In: *Recent Developments in Computer Vision: Second Asian Conference on Computer Vision*, ACCV '95, vol. I. S. Z. Li (Ed.). Springer, Berlin, pp. 214–218.

Jüttner, M., Caelli, T. and Rentschler, I. (1996). Recognition-by-parts: A computational approach to human learning and generalization of shapes. *Biol. Cybern.* **74**, 521–535.

Krieger, G. and Zetzsche, C. (1996). Nonlinear image operators for the evaluation of local intrinsic dimensionality. *IEEE Trans. Image Proc.* **5**, 1026–1042.

APPENDIX I: OVERVIEW OVER THE IPRS DATA TYPES

The general image format

```
typedef struct {
  char *parameter;
  int mode;
  int numimages;
  int *type;
  void **image;
  char **name;
} IPRS_IMAGE;
```

The IPRS_IMAGE image format is the general image format. It can contain any number of sub-images, each of which can be any image of any size and format. The sub-images can also be of type IPRS_IMAGE, so trees and linked lists of images can be formed.

The parameter string allows the user to store a description together with an image. This can be the date of creation, operations performed on the image, etc. Also, the performance of functions can be controlled through certain parameters.

The Cartesian image format

```
typedef struct {
  char *parameter;
  int numdim;
  int *dim;
  char *image;
} IPRS_BYTE_IMAGE;
```

This structure is an example for a Cartesian image format. Other formats available are of type INT, FLOAT, COMPLEX, RGB, etc. The Cartesian image formats are all stored as linear arrays of elements. The images are of arbitrary dimensionality, with the number of dimensions given by `numdim` and the length of each dimension given by the `dim` array.

Non-Cartesian image formats

There are many non-Cartesian image formats, adapted to specific purposes and applications. Examples are EDGE_IMAGE to describe edges in images, FEATURE_SPACE_IMAGE to store extracted image features, and NNET_IMAGE to store neural network connection weights.

APPENDIX II: IPRS PROGRAM EXAMPLE

The following line provides a simple example for using IPRS programs from the UNIX command line shell:

```
unix% highpass -I srcfile -c 50 | threshold -t 100 |
  scale -s 255 | xshow
```

Here, `highpass` will perform a filter operation on the input image `srcfile`. The cutoff frequency is given by the parameter `-c`. The result of this operation is piped to the thresholding program which will set all pixels to 0 whose value is lower than the parameter given by `-t`, whereas pixels whose value exceeds this threshold will be set to 1. The `scale` function multiplies its input by a constant (here 255) and finally the resulting image is displayed on the monitor, using the `xshow` function.

In order to give the reader both an idea of the program structure and show an example of how to add programs that match individual requirements here is a listing of the source code of the `threshold` routine mentioned above:

```
#include<iprs.h>

char help[] =
"Threshold an image\n"
" -I -> input filename\n"
" -O -> output filename\n"
" -t -> threshold\n";

void main(int argc,char **argv)
{
  char *infile=NULL,*outfile=NULL;
  IPRS_IMAGE *in,*out;
  float th=1.0;
/***** Scan the command line arguments for options.
       The control string is similar to the scanf
       function.  With no input/output filename stdin/
       stdout are assumed.  Recognizes reserved options
       to enable debugging, tracing, etc.  *****/
  iprs_getargs(argc,argv,help,"%ft %II %IO",&th, &infile,
               &outfile);
/***** Read input file *****/
  in=iprs_load_image(infile);
/**** Create output image *****/
  out=iprs_imagetobyte(in);
```

```
/**** Call thresholding function *****/
  iprs_threshold_image(in,out,th);
/**** Save output image *****/
  iprs_save_image(out,outfile);
/**** Display debugging summary, depending on debugging
      flags *****/
  iprs_debug_exit();
}

void iprs_threshold_byteimage(
   IPRS_BYTE_IMAGE *src,
   IPRS_BYTE_IMAGE *dest,
   char threshold)
{
   int num,i,j;
   char *sptr;
   char *dptr;
/***** Inform about begin of function for tracing,
       debugging, etc *****/
  iprs_debug_enter("iprs_threshold_byteimage");
/***** Macros to check for valid image pointers *****/
  IPRS_STDERR_IMAGE(src);
  IPRS_STDERR_IMAGE(dest);
/***** Determine total image size *****/
  for(num=1,i=0;i<dest->numdim;i++) num*=dest->dim[i];
/***** Set the pointers to image data *****/
  sptr=src->image;
  dptr=dest->image;
/***** Loop through all image points and do
       thresholding *****/
  for(i=0;i<num;i++) {
     if((*sptr)>threshold) *dptr=1;
     else *dptr=0;
     sptr++;
     dptr++;
  }
/***** Set the parameter string for operation
```

```
        history *****/
   iprs_setiparam(dest,"THRESHOLD",(int)threshold);
/***** Inform about end of function *****/
   iprs_debug_exit();
}

/***** Determine which actual threshold function
        to call *****/
/***** for the general image format *****/
void iprs_threshold_type(
    void *src,
    int srctype,
    IPRS_BYTE_IMAGE *dest,
    float threshold)
{
   iprs_debug_enter("iprs_threshold_type");
   IPRS_STDERR_IMAGE(dest);
   switch(srctype) {
      case IPRS_IMAGETYPE_BYTE:
         iprs_threshold_byteimage(src,dest,(char)threshold);
         break;
      case IPRS_IMAGETYPE_INT:
         iprs_threshold_intimage(src,dest,(int)threshold);
         break;
      case IPRS_IMAGETYPE_FLOAT:
         iprs_threshold_floatimage(src,dest,threshold);
         break;
      case IPRS_IMAGETYPE_RGB:
         iprs_threshold_rgbimage(src,dest,(int)threshold);
         break;
      default:
/***** Call an error function that gives
        formatted output *****/
/***** If no error handling function is set
        it will exit *****/
       iprs_ierror("%e:  %t",IPRS_IMAGETYPE_NS,srctype);
       break;
```

```
  }
  iprs_debug_exit();
}

/***** Handle thresholding for general image format *****/

void iprs_threshold_image(
   IPRS_IMAGE *src,
   IPRS_IMAGE *dest,
   float threshold)
{
  int i,iprs_freef;
  IPRS_BYTE_IMAGE *image;

  iprs_debug_enter("iprs_threshold_image");
  IPRS_STDERR_IIMAGE(src);
  IPRS_STDERR_IIMAGE(dest);
  IPRS_STDERR_NUMSUB(src,dest);

/***** Loop through all sub--images *****/
  for(i=0;i<src->numimages;i++) {
/***** If a sub--image is again a general image,
       do recursion *****/
    if(src->type[i]==IPRS_IMAGETYPE_IMAGE) {
      iprs_threshold_image(src->image[i],dest->image[i],
                           threshold);
    } else {
/***** else create temporary byte image
       if necessary *****/
/***** to hold thresholding result
       and call iprs_threshold_type *****/
      image=iprs_typetobytef(dest->image[i],dest->type[i],
                             &iprs_freef);
      iprs_threshold_type(src->image[i],src->type[i],image,
                          threshold);
      if(iprs_freef) {
/***** now convert back to destination actual type *****/
/***** and remove temporary image *****/
```

```
            iprs_bytetype(image,dest->image[i],dest->type[i]);
            iprs_free_byteimage(image);
          }
        }
      }
      iprs_debug_exit();
}
```