

STAIR Vision Library (v2.4)

Stephen Gould
sgould@stanford.edu

Olga Russakovsky
olga@cs.stanford.edu

Ian Goodfellow
ia3n@cs.stanford.edu

Paul Baumstarck
pbaumstarck@stanford.edu

May 22, 2010

Contents

1	Introduction	7
1.1	Library Design	7
1.1.1	Directory Structure	7
1.2	Installation	8
1.2.1	Linux	9
2	SVL Libraries	11
2.1	svlBase	11
2.1.1	Command Line Processing	11
2.1.2	Code Profiling	12
2.1.3	Loop Timing	13
2.1.4	Messages, Warnings and Errors	14
2.1.5	Configuration Manager	15
2.1.6	Unconstrained Optimization	16
2.1.7	Smart Pointers	17
2.1.8	Thread Pool	17
2.1.9	File Utilities	18
2.1.10	String Utilities	18
2.1.11	Statistics Utilities	19
2.1.12	Bit Arrays	20
2.1.13	Options Interface Class	21
2.1.14	Factory	22
2.2	svlPGM	24
2.2.1	Factors and Factor Operations	24
2.2.2	Cluster Graphs	26
2.2.3	Inference	27
2.2.4	Pairwise Log-Linear CRF Models	27
2.2.5	General Log-Linear CRF Models	29
2.3	svlML	30
2.3.1	Classifiers	31
2.3.2	Feature Whitening	33
2.3.3	Sufficient Statistics	34
2.3.4	Multi-variate Gaussians and Conditional Gaussians	34
2.3.5	Linear Regression	35
2.3.6	Disjoint Sets	35
2.3.7	Vector-Quantization	36
2.3.8	Histograms	36
2.3.9	Quadratic Program Solver	36
2.3.10	Feature Selection	37
2.3.11	Evaluation	38

2.4	svlVision	38
2.4.1	OpenCV Utilities	38
2.4.2	Vision Utilities	39
2.4.3	Basic types	42
2.4.4	Object detection storage types	42
2.4.5	I/O file format	42
2.4.6	Loading images	43
2.4.7	Building a training dataset	45
2.4.8	Pixel Filter Banks	46
2.4.9	Interest point detection	46
2.4.10	Feature extraction	47
2.4.11	Filtering of fragment-based features	51
2.4.12	Sliding window object detection	51
2.4.13	Detection evaluation	52
2.4.14	Camera Models	52
2.4.15	3d	52
2.4.16	Segmentation	52
2.5	svlCuda	53
2.5.1	Data structures and allocation	53
2.5.2	Memory transfer	54
2.5.3	Line integrals	55
2.5.4	Integral images	55
2.5.5	Convolution functions	56
2.5.6	Decision tree set	57
2.5.7	Sliding window detector	58
2.5.8	Verification application	59
2.5.9	Benchmarking application	59
2.5.10	Custom application	60
3	SVL Applications and Scripts	61
3.1	GUI Applications	61
3.1.1	Point Cloud Viewer	61
3.1.2	Image Sequence Labeler	61
3.1.3	Region Labeler	62
3.2	Machine Learning Applications	62
3.2.1	Classifiers	62
3.2.2	Precision-recall curves	63
3.3	Matlab (mex) Applications	63
3.3.1	Probabilistic Graphical Model Inference	63
3.3.2	General CRF Learning and Inference	64
3.3.3	Classifier Training and Evaluation	65
3.4	Matlab (.m) Scripts	65
3.4.1	Library Examples	66
3.5	Test Applications	66
3.6	Computer Vision Applications	67
3.6.1	Standard Object Detection pipeline	67
3.6.2	Pixel-level features	69
3.6.3	Object Detection utilities	69
3.6.4	Multi-class Image Segmentation	70
3.6.5	Video Processing Utilities	72
3.6.6	Depth Superresolution	72

4	Projects	73
4.1	Building Your Own Projects	73
4.2	Vision Utilities	74
4.2.1	combineDictionaries	74
4.2.2	computeDetectionStatistics	74
4.2.3	countSlidingWindows	75
4.2.4	dir2imageSeq	75
4.2.5	dumpVideoFrames	75
4.2.6	extractorCat	75
4.2.7	labelMe2ObjectSequence	75
4.2.8	processDetections	75
4.2.9	processImageSequence	75
4.2.10	splitImages	75
4.2.11	summarizeExtractor	75
4.2.12	Scripts	76
A	Coding Guidelines	77
A.1	Source Control	77
A.2	Structure	78
A.3	Variable and Object Naming	78
A.4	Comments	78
A.5	Portability and Maintainability	78
A.6	Performance	79
A.7	Miscellaneous	79
A.8	Testing	79
B	Configuration	81
C	License	85

Chapter 1

Introduction

This document provides a detailed description of the design and implementation of the **STAIR Vision Library**. It includes an overview of the various components of the library, installation instructions, coding conventions, and outline of specific classes within the library. The intended audience is users and developers of the **STAIR Vision Library**. The document assumes that you are familiar with either the Linux or Windows (Microsoft Visual Studio) development environments, and are an experienced C/C++ programmer.

Since the code is under continuous development and improvement, the ultimate specification for classes within the library is the code itself. When using the library you should consult the various headers for specific details on expected arguments, etc.. Separate tutorial/HOWTO documents provide detail step-by-step instructions on the use of various applications and projects that are built using the library.

The **SVL** is a library of research code initially developed to support the STanford AI Project (STAIR). It has since been expanded to provide a range of software infrastructure for computer vision, machine learning, and probabilistic graphical models. However, the library has maintained its focus as a vehicle for research and this is reflected in its design—many of the classes in the library are only loosely coupled allowing researchers to pull out only what they need and link against their code. The library is intended to be platform independent and currently compiles on both Linux and Windows. We have released the library under the BSD license (see Appendix C).

1.1 Library Design

The **STAIR Vision Library** is actually a collection of software libraries (see Figure 1.1) some of which are third-party open source projects including `xmlParser`, `lbfgs`, `Eigen`, `OpenCV` and `wxWidgets`.¹ The main library is divided into base components (`svlBase`), probabilistic graphical models (`svlPGM`), machine learning algorithms (`svlML`), vision (`svlVision`), and GPU optimized code (`svlCuda`). The dependency structure for the libraries is shown in Figure 1.1.

Building on top of the library is a whole host of standard applications that are distributed with the library. These applications range from machine learning utilities to entire object detection applications. In addition to these applications, a number of student/research projects have been developed that make use of the library. However, these projects may require additional third-party libraries and do not conform to all the same coding guidelines or software architecture specified by the **SVL**.

1.1.1 Directory Structure

The following represents the Stanford University STAIR Vision project repository. The **STAIR Vision Library**, made available publicly, represents only part of this directory tree:

¹`Eigen`, `OpenCV`, and `wxWidgets` need to be installed separately.

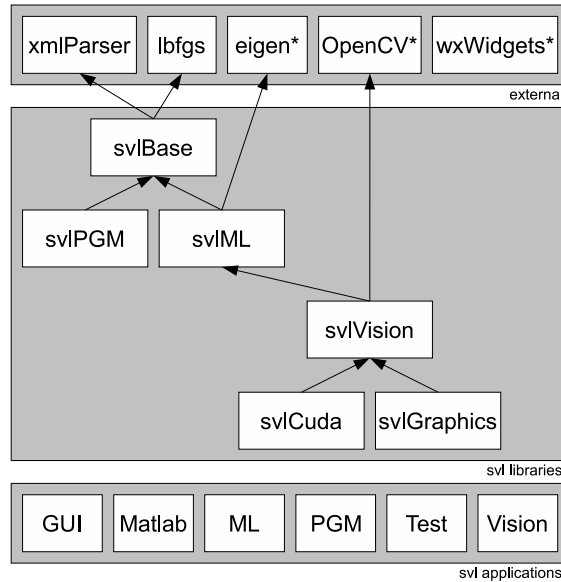


Figure 1.1: STAIR Vision Library design.

DIRECTORY	CONTENTS
bin ¹²	Compiled SVL libraries, applications, and projects.
data ¹²	(empty) Recommended for storing or linking to datasets.
doc	SVL documentation (see also http://ai.stanford.edu/~sgould/svl/).
experiments ¹	(empty) Recommended for running experiments.
external	External libraries required by the STAIR Vision Library . Small libraries are included with the SVL source code. Larger libraries (Eigen, OpenCV and wxWidgets) should be downloaded separately and placed (or linked) here. Under Windows, OpenCV and wxWidgets should be installed in their default system locations.
include	Include directory for applications that use the SVL .
models ¹	Recommended for storing model files, e.g., for object detection.
projects	STAIR Vision Library projects.
svl/apps	STAIR Vision Library applications source code.
svl/lib	STAIR Vision Library source code.
svl/scripts	STAIR Vision Library perl and Matlab scripts.
tests	SVL regression tests.

Notes: ¹Indicates that directory contents are excluded from source control. Some scripts might assume that these directories exist. ²Indicates that directory contents may be released separately, e.g., Linux and Windows binary releases.

1.2 Installation

Complete and up-to-date installation instructions for both Linux and Windows exists on the **STAIR Vision Library** website: <http://ai.stanford.edu/~sgould/svl/>. Note that you may need to install some additional build tools and software libraries.

1.2.1 Linux

Under Linux some build options can be controlled by creating a `make.local` file in the top level **SVL** directory, e.g.,

```
# force 32-bit build on 64-bit machines
BUILD32BIT = 1
```

The full list of options are described in the following table:

OPTION	DEFAULT	DESCRIPTION
FORCE32BIT	0	Set to 1 to compile 32-bit applications on 64-bit machines.
BUILD_GUI_APPS	1	Set to 0 to disable automatic building of GUI applications (needs wxWidgets).
BUILD_MEX_APPS	0	Set to 1 to automatically build Matlab applications.
SVL_SHARED_LIBS	0	Set to 1 to compile applications against shared libraries (.so) instead of static (.a).
SVL_DEBUG_SYMBOLS	1	Set to 0 to remove debug symbols from binaries (resulting in much smaller files).
BUILD_CUDA_KERNELS	0	Set to 1 to compile GPU-accelerated code (needs Cuda).

Tip: *Compiling with `SVL_SHARED_LIBS = 1` and `SVL_DEBUG_SYMBOLS = 0` will significantly reduce the disk space taken up by the executables.*

Chapter 2

SVL Libraries

2.1 svlBase

The `svlBase` library provides a number of utility classes such as profiling and logging that are used throughout the rest of the library. All applications that use the **SVL** must include `svlBase.h` and link to this library.

2.1.1 Command Line Processing

Most **SVL** applications require user-specified input from the command line. For example, an object detection application requires the user to provide the model files (patch dictionaries and classifier parameters), the image to be processed and a specification of what to do with the output. The general form of a command line for executing **SVL** applications is:

```
bin/<application> [<OPTIONS>] (<ARGUMENTS>)*
```

where `<application>` is the executable for the **SVL** application, `<OPTIONS>` are optional command line parameters and `<ARGUMENTS>` are required command line arguments. Optional parameters must start with a dash (e.g., `-verbose`) and precede any required arguments. Most **SVL** applications will print out a usage statement if invoked without any arguments. There are a number of standard options shown in Table 2.1 used by all **SVL** applications.

Command line processing is assisted by the use of a number of macros defined in `svlCommandLine.h`. The following is an example code segment that processes command line arguments.

```
#include "svlBase.h"
```

OPTION	DESCRIPTION
<code>-help</code>	Display application usage and configuration options.
<code>-config <xml></code>	Configure SVL using an XML file (see <code>svlConfiguration</code>).
<code>-set <module> <name> <val></code>	Set <code><module>::<name></code> to have <code><value></code> (see <code>svlConfiguration</code>).
<code>-profile</code>	Enable code profiling (see <code>svlCodeProfiler</code>).
<code>-quiet</code>	Disable printing of all messages except for errors (see <code>svlLogger</code>).
<code>-verbose</code>	Enable printing of verbose messages.
<code>-debug</code>	Enable printing of debug messages.
<code>-log <filename></code>	Log all messages to file <code><filename></code> .
<code>-threads <num></code>	Set the maximum number of allowed threads (see <code>svlThreadPool</code>).

Table 2.1: Standard **SVL** command line options.

```

int main(int argc, char *argv[])
{
    // initialize options with default settings
    int integerValue = 0;
    const char *stringVariable = NULL;

    // process command line
    SVL_BEGIN_CMDLINE_PROCESSING(argc, argv)
        SVL_CMDLINE_INT_OPTION("-integerOption", integerValue)
        SVL_CMDLINE_STR_OPTION("-stringOption", stringVariable)
        SVL_CMDLINE_OPTION_BEGIN("-longOption", p)
            cout << p[0] << "\n";
            cout << p[1] << "\n";
        SVL_CMDLINE_OPTION_END(2)
    SVL_END_CMDLINE_PROCESSING();

    // process required arguments
    for (int i = 0; i < SVL_CMDLINE_ARGC; i++) {
        cout << SVL_CMDLINE_ARGV[i] << endl;
    }

    return 0;
}

```

The `SVL_END_CMDLINE_PROCESSING` macro can take a code block or function call to be executed if the code comes across an unrecognized option. This is useful for printing out usage help. The macros `SVL_CMDLINE_ARGC` and `SVL_CMDLINE_ARGV[]` have the same behavior as `argc` and `argv[]` but with the optional arguments removed.

Warning: *Command line processing proceeds in left-to-right order. If two conflicting options are given, the later one will be the one passed to the application. This applies to configuration (see `-config` option) as well.*

2.1.2 Code Profiling

Code profiling is a useful debugging tool during development as well as being important for comparing the running times of different algorithms in scientific research. The `svlCodeProfiler` class provides basic support for code profiling that gets displayed when the application completes. The profiling is only an estimate and not meant to replace high-quality profiling tools such as `gprof`.

The timer accumulates the amount of processor and real (wall clock) time used between `tic()` and `toc()` calls (child processes, such as file I/O, are not counted in this time). Processor times may be inaccurate for functions that take longer than about 1 hour.

By default, code profiling is turned off. Use `svlCodeProfiler::enabled = true` to turn it on. Most **SVL** applications use the standard command line option `-profile` to enable profiling. The following code snippet provides an example of code profiling.

```

#include "svlBase.h"

void myFunc()
{
    static int h = svlCodeProfiler::getHandle("myFunc");
    svlCodeProfiler::tic(h);

    // do something

    svlCodeProfiler::toc(h);
}

int main(int argc, char *argv[])
{

```

```

svlCodeProfiler::enabled = true;
svlCodeProfiler::tic(svlCodeProfiler::getHandle("main"));

for (int i = 0; i < 1000; i++)
    myFunc();

svlCodeProfiler::toc(svlCodeProfiler::getHandle("main"));
svlCodeProfiler::print(cout);
return 0;
}

```

The macros `SVL_FCN_TIC` and `SVL_FCN_TOC` can be used at the entry and exit of your functions to instrument the entire function. Make sure you put `SVL_FCN_TOC` *before* all `return` statements within the function.

Warning: *Instrumenting code for profiling will unavoidably slow the code down, so do not use `tic()` and `toc()` within tight loops. Unlike compiling with `-pg` for `gprof`, `tic()` and `toc()` are always compiled into the code.*

2.1.3 Loop Timing

Complementary to the code profiler is the `svlLoopTimer`, which provides expected times to completion (ETC) for nested loops. In a multi-loop procedure, after just one iteration of the innermost loop, the class can provide an accurate ETC for every surrounding loop. This is useful for any procedure that takes longer than a few minutes to run, including multi-hour training.

To time a loop, one uses the `push(int n_trials)` method of the class to add a new timer (this method must know the total number of loop iterations to be able to provide ETC values). Inside of the loop, the `inc()` method should be called at the end of every loop to increment the timer. After completion of the loop, the `pop()` method discards the top-most timer. Some example code with output is given below:

```

svlLoopTimer t;
int i_trials = 10;
t.push(i_trials); // Initialize with number of iterations
for ( int i=0; i<i_trials; ++i ) {
    t.printETC(); // Call at beginning of loop

    // Wait for one second
    clock_t tic = clock();
    while ( clock()-tic < CLOCKS_PER_SEC ) {}

    t.inc(); // Increment at end of loop
}
t.pop();

```

This produces the output:

```

0/10:  0:00:00 > -:--:--
1/10:  0:00:01 > 0:00:09
2/10:  0:00:02 > 0:00:08
3/10:  0:00:03 > 0:00:07
...
9/10:  0:00:09 > 0:00:01
10/10: 0:00:10 > 0:00:00

```

This output is in the form, “(# elapsed iterations)/(total # iterations): (time elapsed) > (ETC).”

To time multiple loops, one needs only push on multiple timers:

```

svlLoopTimer t;
int i_trials = 10;

```

```

t.push(i_trials); // Initialize first loop
for ( int i=0; i<i_trials; ++i ) {
    int j_trials = 5;
    t.push(j_trials); // Initialize second loop
    for ( int j=0; j<j_trials; ++j ) {
        t.printETC();
        // Wait for one second
        clock_t tic = clock();
        while ( clock()-tic < CLOCKS_PER_SEC ) {}
        t.inc();
    }
    t.pop(); // Clear inner timer.
    t.inc(); // Increment outer timer
}
t.pop();

```

The first few lines of output here are:

```

0/5:  0:00:00 > -:--:-- -- 0/10:  0:00:00 > -:--:--
1/5:  0:00:01 > 0:00:04 -- 0/10:  0:00:01 > 0:00:49
2/5:  0:00:02 > 0:00:03 -- 0/10:  0:00:02 > 0:00:48

```

The innermost loop's elapsed time and ETC is printed first, followed by any other loops.

By default the timer assumes that each loop iteration will take the same amount of time when computing ETC. If different behavior is desired, however, additional parameters can be passed to `push()`. If, for example, each loop will take only 90% as much time as the previous loop, one can use `push(n_loops, 0.9, GEOMETRIC)`, providing the `GEOMETRIC` parameter with numeric ratio 0.9. Then, if the first loop takes 1 unit of time, the timer will correctly calculate ETC at $0.9+0.89+0.729+\dots$ for as many remaining iterations as there are.

The complete options to `push()` and their results are given below:

RATIO TYPE	ASSUMED DURATION OF LOOP i BASED ON RATIO f
GEOMETRIC	$f^i \cdot 100\%$ of the first loop ($f \cdot 100\%$ of previous loop).
ADDITIVE	$(1 + i \cdot f) \cdot 100\%$ of the first loop.

The ratio f can be greater or less than one in the geometric case, and positive or negative in the additive case. The default case uses `GEOMETRIC` with a ratio of 1.0, achieving loops of uniform duration.

2.1.4 Messages, Warnings and Errors

Messages, warnings and errors are managed via the `svlLogger` class. The `SVL_LOG()` macro will automatically write log messages to a file (if specified) and display them on the console. You can set the verbosity level to control which messages get displayed. The verbosity levels are:

VERBOSITY	DISPLAY	DESCRIPTION
SVL_FATAL	-*-	An unrecoverable error has occurred and the code will terminate.
SVL_ERROR	-E-	A recoverable error has occurred, e.g., a missing file.
SVL_WARNING	-W-	Something unexpected happened, e.g., a parameter is zero.
SVL_MESSAGE	---	Standard messages, e.g., application-level progress information.
SVL_VERBOSE	---	Verbose messages, e.g., image names and sizes during loading.
SVL_DEBUG	-D-	Debugging messages, e.g., matrix inversion results, etc..

Applications can override the message displaying functions by registering callbacks with the `svlLogger` class. This is useful for interfacing to Matlab or displaying errors in GUI dialog boxes. The following example registers a callback for capturing errors and terminates if too many errors occur.

```

#include "svlBase.h"

void errorMessageCallback(const char *msg) {
    static int counter = 0;
    std::cerr << msg << std::endl;

    if (++counter > 5) {
        std::cerr << "too_many_error_messages" << std::endl;
        exit(0);
    }
}

int main(int argc, char *argv[])
{
    // set svlLogger callbacks
    svlLogger::showErrorCallback = errorMessageCallback;

    for (int i = 0; i < 10; i++) {
        SVLLOG(SVLLOG.ERROR, "error_message_number_" << i);
    }

    return 0;
}

```

Asserts

Functions should use the `SVL_ASSERT()` or `SVL_ASSERT_MSG()` macros rather than the standard `assert()` to allow applications such as Matlab to trap errors.

2.1.5 Configuration Manager

For many research projects it is useful to have a standard configuration for running experiments with only a few parameters changing from one experiment to the next. The **STAIR Vision Library** supports this through two main mechanisms—XML configuration and command line options. The general strategy is to create an XML file with the standard configuration and then provide overrides for various settings on the command line. The system is lightweight while still catering for most configuration needs. An example XML configuration file is shown below. See Appendix B for all standard configuration options.

```

<svl>
  <svlBase.svlCodeProfiler    enabled='false' />
  <svlBase.svlLogger          logLevel='message'
                              logFile='' />
  <svlBase.svlThreadPool      threads='8' />
  <svlPGM.svlFactorOperations cacheIndexMapping='true'
                              useSharedIndexCache='false' />
  <svlML.svlConfusionMatrix   colSep='&#9;'
                              rowBegin='&#9;'
                              rowEnd='' />
  <myApplication              attributeName='attributevalue' />
    <myArbitraryData>
      1 2 3 4
    </myArbitraryData>
  </myApplication>
</svl>

```

Tip: Use `&` for `&`, `<` for `<`, `>` for `>`, and `	` for `\t` in XML configuration files.

The `svlConfigurationManager` class handles configuration of static parameters for the **SVL** libraries and can be used for configuring individual applications or projects. The command line options `-config` and

`-set` will automatically invoke the configuration manager. To invoke it manually you can simply call the `svlConfigurationManager::configure()` function.

Standard configuration parameters are defined by the triplet: module, name and value. In the XML configuration file shown above the tags “`svlBase.svlCodeProfiler`”, “`svlPGM.svlFactorOperations`”, etc. define the module and the node’s attributes define the name-value pairs. Each configurable **SVL** class is prepended with its library name, e.g., the `svlLogger` class has module name “`svlBase.svlLogger`”. An application can define its own module (XML node) with arbitrary name-value pairs. The structure of the application-specific XML node can be arbitrary and it is up to the application developer to parse non-attribute content (such as the `myArbitraryData` node in the example above). The `-set` command line option can only be used for name-value pairs.

To register a configurable class with the **SVL** Configuration Manager, an application needs to create a derived class from base `svlConfigurableModule` and override the `setConfiguration()` function. More control can be achieved by also overriding the `readConfiguration()` function. To register the class, the code simply needs to instantiate a global class member—the `svlConfigurableModule` constructor will handle registration.

```
// define configuration
class myAppConfig : public svlConfigurableModule {
public:
    static int v;

public:
    myAppConfig() : svlConfigurableModule("myApplication") { }
    ~myAppConfig() { }

    void usage(std::ostream& os) const {
        os << "_____Demonstrates the configuration manager.\n";
    }

    void setConfiguration(const char *name, const char *value) {
        if (!strcmp(name, "attributeName"))
            v = atoi(value);
        else SVLLOG(SVLLOG_FATAL, "unknown_configuration_parameter_" << name);
    }
};

// initialize static variables
int myAppConfig::v = 0;

// register configuration
myAppConfig gMyAppConfig;
```

Tip: Use `-config` on the command line by itself to get online help for most configurable modules.

2.1.6 Unconstrained Optimization

The `svlOptimizer` provides a virtual base class for minimizing large-scale unconstrained optimization problems using the L-BFGS algorithm [16, 19]. Derived classes must override functions `objective()` and `gradient()`. They should also implement the `objectiveAndGradient()` function for efficiency—the default is for this function to call the other two. Derived classes may also override the `monitor()` function. The monitor function can assume that `_x` contains the current estimate for the solution. Other functions should use the input argument `const double *x`.

The following simple example optimizes the one-dimensional function $f(x) = (x - 2)(x - 4)$.

```
class myObjective : public svlOptimizer
{
    myObjective() : svlOptimizer(1) { }
    ~myObjective() { }
```

```

double objective(const double *x) {
    return (x[0] - 2.0) * (x[0] - 4.0);
}

double gradient(const double *x, double *df) {
    return 2.0 * (x[0] - 3.0);
}
};

int main()
{
    myObjective objFunction;

    double f_star = objFunction.solve(1000);
    double x_star = objFunction[0];

    cout << "f(" << x_star << ") = " << f_star << endl;
    return 0;
}

```

2.1.7 Smart Pointers

Smart pointers are dynamically allocated objects that manage their own destruction. This allows an object to be shared between different owners. When the last owner is finished using the object, i.e., the object goes out of scope, the smart pointer will automatically delete the object. Smart pointers also avoid the need for deep copies since each owner references the same object. The following code gives an example of smart pointer usage:

```

class Object {
public:
    svlSmartPointer<char> str;
    Object() {
        str = new char[11]; strcpy(str, "helloworld"); str[10] = '\0';
    }
};

int main()
{
    Object *o = new Object();
    cout << (char *)o->str << endl;

    svlSmartPointer<char> s(o->str);
    delete o;

    cout << (char *)s << endl;
}

```

Be aware that when modifying an object through a smart pointer, all other objects that share the smart pointer will also be affected. Sometimes you will want to make a clone of the object instead:

```

svlSmartPointer<Object> p1(new Object());
svlSmartPointer<Object> p2(new Object(*p1));

```

2.1.8 Thread Pool

The `svlThreadPool` class is an abstraction for running threaded processes designed around the `pthread`s library. When the thread pool is constructed, the user requests a certain number of threads. However, the constructor will allocate only up to `svlThreadPool::MAX_THREADS` concurrent threads (see Section 2.1.5 for how to set this parameter from the command line).

The following code snippet provides an example of running threads:

```

void *myThreadFcn(void *args, unsigned threadId)
{
    cout << "In_thread_" << threadId << endl;
    return NULL;
}

int main()
{
    svlThreadPool threadPool(4);
    threadPool.start();
    for (int jobNumber = 0; jobNumber < 100; jobNumber++) {
        threadPool.addJob(myThreadFcn, NULL);
    }
    threadPool.finish();

    return 0;
}

```

Warning: *Threads are not currently implemented for Windows. All threaded applications will run in the main thread.*

2.1.9 File Utilities

The **SVL** provides some useful platform independent file utilities for creating directories `svlCreateDirectory()`, listing directory contents `svlDirectoryListing()`, checking for file existence `svlFileExists()`, reading lines of a file into a vector of strings `svlReadFile()`, or counting the number of fields in a comma-, space-, or tab-delimited file `svlCountFields()`. See the `svlFileUtils.h` header file for more information.

2.1.10 String Utilities

The `svlStrUtils` unit contains a number of useful string (character array) manipulation utilities. The general templated `toString<T>()` function converts an arbitrary datatype into an STL string representation. Specialized functions for many datatypes, e.g., OpenCV structures, have been implemented in the appropriate libraries.

The function `parseString()` will convert from a white-space delimited string to a vector of tokens of a given type, e.g.,

```

std::string s = std::string("1.0_2.0_3.0");
std::vector<double> v;

parseString<double>(s, v);
cout << toString(v) << endl;

```

The function `parseNameValueString()` will convert strings of the form “<name1>=<val1> <name2>=<val2>” into an STL map.

The function `padString()` will left-pad a string up to a given size, e.g.,

```

int index = 10;
std::string filename = padString(toString(index), 8, '0');

```

The `svlStrUtils` unit also contains a number of functions for processing paths and file names.

FUNCTION	DESCRIPTION
strBaseName	Removes the directory and extension from a file path.
strFilename	Removes the directory, leaving the filename, from a file path.
strDirectory	Removes the filename from a file path.
strExtension	Returns the file extension from a file path.
strReplaceExt	Replaces the file extension with another one.
strWithoutExt	Strips the file extension from a file path.
strFileIndex	Extracts the index from a filename of the form: <code>base<NNNNN>.ext</code> .

2.1.11 Statistics Utilities

The `svlStatUtils.h` file provides a number of templated statistics utilities that operate on STL vectors.

FUNCTION	DESCRIPTION
minElem	Returns the smallest element in the vector.
maxElem	Returns the largest element in the vector.
mean	Returns the mean of values in the vector (or zero for empty vectors).
median	Returns the median value in the vector (or zero for empty vectors).
mode	Returns the most commonly occurring value in the vector.
variance	Returns the variance of values in the vector.
stddev	Returns the standard deviation of values in the vector.
argmin	Returns the index of the (first) minimum element in the vector.
argmins	Returns a vector of indices of the minimum elements in a vector of vectors.
argmax	Returns the index of the (first) maximum element in the vector.
argmaxs	Returns a vector of indices of the maximum elements in a vector of vectors.
excessKurtosis	Returns the excess kurtosis of the values in a vector.
percentiles	Returns a vector of the percentile rank of each sample, divided by 100. i.e., if the third element is in the 99th percentile, the third element of the return value will be 0.99.
range	Returns an STL <code>pair</code> contains the minimum and maximum element in the vector.
sum	Returns the sum of a vector of <code>double</code> .

In addition, the file implements the following useful functions.

containsInvalidEntries

Returns true if the vector contains NaN or `inf` values.

extractSubVector

Extracts a subvector from a vector given a list of indices. For example,

```
double data[] = {1.0, 2.0, 3.0, 4.0, 5.0};
int index[] = {0, 2, 4};
vector<double> x(data, data + 5);
vector<int> i(index, index + 3);
vector<double> y = extractSubVector(x, i);
```

will return the vector $y = [1.0, 3.0, 5.0]$.

removeOutliers

Trims a vector of values to remove the middle `keepSize` entries. For example to keep the middle 90-percentile use:

```
vector<double> dataOut = removeOutliers(dataIn, (int)(0.9 * dataIn.size()));
```

powerSet

Computes the power set (excluding the empty set) of a given set of objects. For example, if $x = \{0, 1, 2\}$ then `powerSet()` will return $\mathcal{P}(x) = \{\{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$.

logistic

Computes the logistic function $y = \frac{1}{1 + \exp\{-\theta^T x\}}$.

entropy

Computes the entropy, $H = -\sum_i p_i \log p_i$, of a discrete distribution. The distribution does not need to be normalized (but must be non-negative).

expAndNormalize

Converts a vector of log-likelihoods to a normalized probability distribution by performing $y_i = \frac{1}{Z} \exp\{x_i - \alpha\}$ where $Z = \sum_i \exp\{x_i - \alpha\}$ and $\alpha = \max_j x_j$.

randomPermutation

Computes a random permutation of the integers in the range $[0, n - 1]$.

successor and predecessor

Computes successor and predecessor discrete vectors such as would be produced by an odometer, but from left-to-right ordering.

```
vector<int> x(3, 0);
for (int i = 0; i < 1000; i++) {
    cout << toString(x) << endl;
    successor(x, 10);
}
```

huberFunction and huberDerivative

Computes the Huber function (or its derivative) where the Huber function is defined as $h_\lambda(x) = x^2$ for $-\lambda \leq x \leq \lambda$ and $\lambda(2|x| - \lambda)$ otherwise.

bhattacharyyaDistance

Computes the Bhattacharyya distance between two discrete distributions, $D_B(p, q) = -\log \sum_i \sqrt{p_i q_i}$. The distributions do not need to be normalized.

euclideanDistanceSq

Computes the squared Euclidean distance between two vectors.

2.1.12 Bit Arrays

The `svlBitArray` class implements a packed array bits as demonstrated by the following example:

```
svlBitArray bits(10);

for (int i = 0; i < bits.size(); i++) {
    if (i % 2) bits.set(i);
    else bits.clear(i);
}
```

```

}
for (int i = 0; i < bits.size(); i++) {
    if (bits[i]) {
        cout << "bit_" << i << "_is_set\n";
    }
}
}

```

2.1.13 Options Interface Class

The `svlOptions` base class provides a convenient and consistent mechanism for dynamically getting and setting optional parameters for instantiated objects. Note that this is different from the configuration manager (Section 2.1.5) which is used for setting default parameters. The class provides `setOption()` and `getOption()` methods for different data-types (internally all data is stored as strings). In the constructor for a class, all options should be declared using the `declareOption()` method. Derived classes can remove options using the `undeclareOption()` method.

```

class myClass : public svlOptions {
public:
    myClass() : svlOptions() {
        declareOption("featureExponent", 1.0);
    }
    ...
};

```

The following provides an example of setting a real-valued option. Trying to set an option that has not been declared (or that has been undeclared) will result in an error.

```

int main()
{
    myClass myObject;
    myObject.setOption("featureExponent", 2.0);
    ...
}

```

The option value can be retrieved using the `getOption()` method of the appropriate type.

```

double myClass::featureValue(double x)
{
    return pow(x, this->getOptionAsDouble("featureExponent"));
}

```

Note that getting and setting options costs processing time so do not use them in tight loops; rather copy the value into a temporary variable before entering the loop:

```

vector<double> myClass::featureValues(const vector<double>& x)
{
    vector<double> y(x.size());

    double exponent = this->getOptionAsDouble("featureExponent");
    for (unsigned i = 0; i < x.size(); i++) {
        y[i] = pow(x[i], exponent);
    }
    return y;
}

```

2.1.14 Factory

The `svlFactory` class allows for the construction of objects whose type is not known until runtime. It is a templated class—`svlFactory<T>` is capable of constructing all classes derived from type `T`, provided that they have been appropriately registered with the factory.

The **STAIR Vision Library** instantiates `svlFactory` to make factories for the `svlClassifier` family of classes and the `svlFeatureExtractor` class. Future releases of the library will most likely construct other types of objects with factories as well.

Constructing objects from an existing factory

`svlFactory<T>` provides three methods for constructing an object whose type is not known until runtime. The `make()` method takes a string argument that identifies which subclass of `T` should be constructed. This could be used to implement, for example, an application that allows the user to specify the classification algorithm to use on the command line. The following code demonstrates how to create an `svlFeatureExtractor` from a string:

```
svlFeatureExtractor * extractor = svlFeatureExtractorFactory().make("HaarFeatureExtractor");
```

(Note: `svlFeatureExtractorFactory()` is a function that returns a reference to a global variable of type `svlFactory<svlFeatureExtractor>`.)

There are also two `load()` methods, one that takes a file path as its argument and another that accepts an XML node. Both of these methods load the object stored in the file or XML node. This is different from calling the `load()` method of a pre-existing object, because the factory can load the contents of the file regardless of its type (so long as it is a subclass of `T`) while the object's `load` method would only work if the object were of exactly the same type as the object stored in the file.

The following code demonstrates how to load an `svlClassifier` of unknown type from a file, then check its type:

```
svlClassifier * classifier = svlClassifierFactory().load("myClassifier.dat");

if (classifier)
{
    cout << "Successfully loaded a classifier." << endl;

    if (dynamic_cast<svlBoostedClassifier *>(classifier))
        cout << "It is a boosted classifier." << endl;

    if (dynamic_cast<svlMultiClassLogistic *>(classifier))
        cout << "It is a multiclass logistic classifier." << endl;

    if (dynamic_cast<svlLogistic *>(classifier))
        cout << "It is a logistic classifier." << endl;

    if (dynamic_cast<svlBoostedClassifierSet *>(classifier))
        cout << "It is a boosted classifier set." << endl;
}
```

(Note: `svlClassifierFactory()` is a function that returns a reference to a global variable of type `svlFactory<svlClassifier>`.)

Registering a class to a factory

Before a factory can create a class, it must be registered to the factory. There are a variety of ways of registering a class.

In the simplest case, the class has a no-argument constructor and is stored in an XML file that opens with

```
<familyType id="subclass_id" ... >
```

where “familyType” should be substituted with the name of the superclass whose factory should be used (e.g., `svlClassifier`) and “subclass_id” should be substituted with the string ID you want to use to identify the subclass you are registering (e.g. “JOHN_SMITH’S_CUSTOM_SVM”).

In this case, your class can be configured to be automatically registered when your application launches. To do this, put

```
SVLAUTOREGISTER_H ( subclass_id , familyType )
```

in the header file below the definition of your class. You also need to add a call to `SVL_AUTOREGISTER_CPP()` in a `.cpp` file whose corresponding `.o` file is guaranteed to be linked to your application. Since `.o` files are not linked if none of their symbols are referenced by the main application, this location must be chosen carefully. If your new subclass is part of the **STAIR Vision Library**, then the only way to guarantee that all applications that are built on **SVL** will be able to use your class is to put the macro call in the same file as the definition of the parent class of the family (e.g., `svlClassifier.cpp`). If your new subclass is part of your own application, then you are free to put the macro call in any `.cpp` file that you are sure will be linked to your application. In either case, the call is

```
SVLAUTOREGISTER_CPP(subclass_id , familyType , yourSubclass)
```

where “subclass_id” and “familyType” are substituted as before, and “yourSubclass” should be substituted with the name of your subclass.

If it is not possible to store your object in an XML file, then you will also need to write and register a file checking function. When the factory attempts to load a file, it will first check if the file agrees with the standard **STAIR Vision Library** XML file format. If it does not, the factory will then iteratively call each registered file checking function on the file. A file checking function is simply a function that parses the file and attempts to identify it as being a certain type of object. If the file can be positively identified, the file checking function returns the string ID of the class contained in the file. Otherwise, the file checking function returns the empty string. File checking functions should not report errors if they do not recognize the format of the file—another file checking function might be able to identify the contents of the file, and if no file checking function can do so, the factory will report an error. For an example of a file checking function, see `svlCvBoostFileChecker()` in `svlBoostedClassifier.cpp`.

If you write your own file checking function, you will also need to register it. This can be accomplished with the `SVL_AUTOREGISTER_FILECHECKER_H()` and the `SVL_AUTOREGISTER_FILECHECKER_CPP()`, which are similar to the registration macros for subclasses.

If you cannot write your class to have a no-argument constructor, then the `SVL_AUTOREGISTER_CPP()` macro will fail. In this case, instead of putting the call to the macro in your `.cpp` file, you should look at the macro’s definition in `svlFactory.h`, then expand its contents in the appropriate `.cpp` file, replacing the call to the no-argument constructor with some set of appropriate default arguments. Since the same thing could be accomplished by adding default arguments to your class’s constructor’s declaration, this is not a use-case that we have attempted to support with special case macros.

Lastly, it is also possible to manually register classes and file checking functions to a factory using the factory’s `registerType()` and `registerFileChecker()` methods. The former will require you to define a function that can create your subclass. Manual registration is mainly useful if you suspect that there is a bug with the automatic registration, for instance, that the `.cpp` file with your `SVL_AUTOREGISTER_CPP()` calls is being ignored by the linker.

Creating new factories

End users of the library are unlikely to need to create new instances of `svlFactory`, but if you are developing the library itself you may find it necessary to do so. This must be done with care to ensure that auto-registration works properly.

The auto-registration macros declare objects of type `svlFactory<T>::autoreg`. When these objects are constructed, their constructors automatically call the registration methods of a factory object that is passed to them by the macro. If the auto-registration objects are constructed before the factory, this will cause a

segmentation fault, since the factory will attempt to insert information about the classes being registered into a map that hasn't been initialized yet. In order to prevent this, it is necessary to route all accesses to the factory object through a function that contains the factory as a static local variable. This ensures that an attempt to access the factory will cause it to be constructed, thus preventing its premature use.

In order for the auto-registration macros to be able to locate the factory, the function returning the reference to the factory must be named `[familyType]Factory()`. Likewise, for XML file parsing to follow the conventions described in this documentation, the argument to the factory's constructor must be "[familyType]".

2.2 svlPGM

Probabilistic graphical models are factored representations of probability distributions that include Bayesian Networks (BNs), Markov random fields (MRFs) and conditional Markov random fields (CRFs). The models use a graph-based representation—nodes representing random variables and edges representing probabilistic relationships—from which they derive their name. In the **SVL** we only deal with discrete random variables (i.e., finite domains). Given a set of random variables $\mathbf{Y} = \{Y_1, \dots, Y_n\}$ where each variable Y_i has domain $\mathbf{dom}(Y_i)$, we can write the general form for a graphical model over \mathbf{Y} given some features \mathbf{X} as

$$P(\mathbf{Y} | \mathbf{X}) = \frac{1}{Z(\mathbf{X})} \prod_c \Psi_c(\mathbf{Y}_c; \mathbf{X}) \quad (2.1)$$

where $\mathbf{Y}_c \subseteq \mathbf{Y}$ is a subset of the variables (clique) and $Z(\mathbf{X})$ (called the *partition function*) ensures that the probability distribution sums to 1. Here the *potentials* or *factors*, $\Psi_c(\mathbf{Y}_c; \mathbf{X})$, are functions that take an assignment to the variables \mathbf{Y}_c and return a non-negative real number \mathbb{R}_+ indicating the model's preference for that assignment. Often it is more convenient to represent the model in log-space,

$$P(\mathbf{Y} | \mathbf{X}) = \frac{1}{Z(\mathbf{X})} \exp\{-E(\mathbf{Y}; \mathbf{X})\}. \quad (2.2)$$

Here $E(\mathbf{Y}; \mathbf{X}) = -\sum_c \psi_c(\mathbf{Y}_c; \mathbf{X})$ is called the *energy* function (with $\psi_c = \log \Psi_c$). Note that maximizing probability (i.e., finding the MAP assignment) is equivalent to minimizing energy.

In addition to the library code described here, the **STAIR Vision Library** provides a number of C++ and Matlab applications for graphical model inference and learning (see Section 3).

2.2.1 Factors and Factor Operations

The `svlFactor` class implements a discrete (table) factor (or *clique potential*) for use in Bayesian Networks, Markov random fields (MRFs) and conditional Markov random fields (CRFs). The factor assumes that variables are indexed by integer. Variable values are indexed from zero to `varCardinality(v) - 1`. The `indexOf()` and `valueOf()` functions can be used to access table entries. For faster access, the `[]`-operator can also be used. Entries are stored in a linear array and ordered by first-indexed variable so for the factor $\psi(X, Y)$ we would have `psi[0] = $\psi(0, 0)$` , `psi[1] = $\psi(1, 0)$` , ..., `psi[psi.size() - 1] = $\psi(|\mathbf{dom}(X)| - 1, |\mathbf{dom}(Y)| - 1)$` .

Some optimizations have been incorporated for fast factor multiply and marginalization, but for really fast operations use the derived `svlFactorOperation` classes which allow indices to be precomputed. This is especially important for iterative algorithms such as message passing inference which perform repeated operations on the same factors.

The following example constructs two factors over variables $\{X_0, X_1\}$ and $\{X_1, X_2\}$ and multiplies them together to produce a factor over $\{X_0, X_1, X_2\}$.

```
svlFactor psi1 , psi2 ;

// define factors
psi1.addVariable(0 , 3); // X-0 has cardinality 3
```

```

psi1.addVariable(1, 2); // X-1 has cardinality 2
psi2.addVariable(1, 2); // X-1 has cardinality 2
psi2.addVariable(2, 2); // X-2 has cardinality 2

// populate directly, but we could also use:
//   psi1[psi1.indexOf(1, <x-1>, psi1.indexOf(0, <x-0>))] = <v>;
psi1[0] = 0.5; psi1[1] = 0.8; psi1[2] = 0.1;
psi1[3] = 0.0; psi1[4] = 0.3; psi1[5] = 0.9;
psi1.write(cout);

psi2[0] = 0.5; psi2[1] = 0.7;
psi2[2] = 0.1; psi2[3] = 0.2;
psi2.write(cout);

// multiply factors together
svlFactor psi3;
svlFactorProductOp op1(&psi3, &psi1, &psi2);
op1.execute();
psi3.write(cout);

```

The available factor operations (derived from `svlFactorOperation`) are:

OPERATION	DESCRIPTION
<code>svlFactorAtomicOp</code>	Executes a sequence of factor operations as a single operation.
<code>svlFactorCopyOp</code>	Copies one factor into another. Assumes that the order of variables is the same in both factors.
<code>svlFactorProductOp</code>	Multiplies two or more factors together.
<code>svlFactorDivideOp</code>	Divides the entries in one factor by corresponding entries from another.
<code>svlFactorAdditionOp</code>	Adds two or more factors together.
<code>svlFactorSubtractOp</code>	Subtracts the entries in the second factor from corresponding entries in the first.
<code>svlFactorWeightedSumOp</code>	Adds a weighted combination of two factors.
<code>svlFactorMarginalizeOp</code>	Marginalizes one or more variables out of a factor.
<code>svlFactorMaximizeOp</code>	Maximizes over one variable within a factor.
<code>svlFactorNormalizeOp</code>	Normalizes the entries in a factor to sum to one. (Inline)
<code>svlFactorLogNormalizeOp</code>	Subtracts the maximum entry in a factor from all entries. (Inline)

I/O File Format

Factors are serialized as XML objects with the following format:

```

<Factor>
  <Vars> 1 2 </Vars>
  <Cards>2 2 </Cards>
  <Data> 0.5 0.7 0.1 0.2 </Data>
</Factor>

```

Factor Table Storage

Some large models, such as those used in low-level computer vision, contain a large number of variables but repeated structure (such as pairwise smoothness terms between adjacent pixels). Since pairwise terms in these models are not conditioned on features of the pixels, the values can be shared. The `svlFactorStorage` class handles storage of the factor entries and can be shared between different factors via the `svlFactor(svlFactorStorage* sharedStorage)` constructor. Currently only full tables are supported. Future versions of **STAIR Vision Library** will include mechanisms for dealing with potentials sparse forms (such as Potts models).

2.2.2 Cluster Graphs

Encapsulates a *cluster graph* (or *clique graph/tree*) for Bayesian networks and Markov random fields. This is used to define the model for various inference algorithms (see Section 2.2.3). Briefly, a cluster graph is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where \mathcal{V} is a set of clusters over variables and \mathcal{E} is the set of edges between clusters. The nodes $v \in \mathcal{V}$ are annotated with a factor over the variables in v , i.e., $\psi_v(\mathbf{Y}_v)$, and the edges are annotated with separators, i.e., for $(u, v) \in \mathcal{E}$ the separator set is $\mathbf{Y}_{uv} \subseteq \mathbf{Y}_u \cap \mathbf{Y}_v$. A valid cluster graph \mathcal{G} for a graphical model must satisfy the following two properties:

- **Family Preserving Property.** For all factors in the model $\psi_c(\mathbf{Y}_c)$ there must exist a least one node $v \in \mathcal{V}$ such that $\mathbf{Y}_c \subseteq \mathbf{Y}_v$.
- **Running Intersection Property.** For any variable Y_i and two nodes $u, v \in \mathcal{V}$ such that $Y_i \in \mathbf{Y}_u$ and $Y_i \in \mathbf{Y}_v$, then there must exist one and only one path between u and v in \mathcal{G} such that for all w on the path, $Y_i \in \mathbf{Y}_w$.

Note that some inference algorithms (e.g., α -expansion) use the cluster graph representation, but do not require the running intersection property to hold.

The following pseudo-code constructs a Bethe-approximation cluster graph for a graphical model.

```
int numVariables = ...;
vector<int> varCardinalities(numVariables);
...

svlClusterGraph model(numVariables, varCardinalities);

// add singleton (unary) potentials
for (int i = 0; i < numVariables; i++) {
    svlClique c = getSingletonVariables(i);
    svlFactor psi = getSingletonFactor(i);
    model.addClique(c, psi);
}

// add higher-order potentials
vector<pair<int, int>> edges;
for (int i = 0; i < numHighOrderTerms; i++) {
    svlClique c = getHighOrderVariables(i);
    svlFactor psi = getHighOrderFactor(i);
    model.addClique(c, psi);

    for (svlClique::const_iterator ci = c.begin(); ci != c.end(); ci++) {
        edges.push_back(make_pair(*ci, model.numCliques() - 1));
    }
}

// add edges for Bethe-approximation
model.connectGraph(edges);
```

I/O File Format

Cluster graphs are serialized as XML objects with the following format:

```
<ClusterGraph vars="..." nodes="..." edges="..." version="1">
  <VarCards>
    <!-- variable cardinalities -->
  </VarCards>
  <Clique index="..." size="...">
    <!-- list of clique variables -->
  </Clique>
  ...
</Edges>
```

```

<!-- optional list of edges between cliques -->
</Edges>

<Potentials>
  <Factor>
    <!-- factor corresponding to cliques -->
  </Factor>
  ...
</Potentials>
</ClusterGraph>

```

2.2.3 Inference

Inference in graphical models involves computing marginals over variable or finding the most likely assignment to variables (MAP inference). The **SVL** implements a number of variants of state-of-the-art message passing and graph-cut-based inference algorithms via the `svlMessagePassingInference`, `svlGraphCutsInference`, `svlAlphabetSOUPInference`, and `svlDualDecompositionInference` classes including:

- Synchronous or asynchronous sum-product/max-product message passing [21].
- Synchronous or asynchronous max-product message passing on log-space messages.
- Synchronous or asynchronous sum-product-divide/max-product-divide message passing.
- Synchronous or asynchronous max-product-divide message passing on log-space messages.
- Asynchronous max-product message passing on log-space messages (lazy variant).
- Residual sum-product/max-product belief propagation [5].
- General convergent message passing algorithm of Globerson and Jaakkola [8].
- Cluster pursuit algorithm of Sontag et al. [22].
- α -expansion and $\alpha\beta$ -swap algorithms [2].
- Max-product message-passing variant of the Alphabet SOUP algorithm [11].
- Dual decomposition [14].

All MAP inference algorithms (other than the max-product variants) assume that the potentials are defined in log-space (i.e., negative energy).

An example for running sum-product inference on an existing cluster graph file is show below.

```

const char *filename = "...";

svlClusterGraphs graph;
graph.read(filename);

svlMessagePassingInference inferenceObject(graph);
inferenceObject.inference(SVLMP_SUMPROD, 100);

for (int i = 0; i < graph.numCliques(); i++) {
  cout << "Clique_" << i << " : " << toString(graph.getClique(i)) << "\n";
  cout << inferenceObjects[i] << "\n";
}

```

2.2.4 Pairwise Log-Linear CRF Models

The `svlPairwiseCRFModel` and associated classes implement learning and inference algorithms for a log-linear condition Markov random field (CRF) model with singleton (or *unary*) and pairwise potentials. All variables must have the same cardinality (for mixed cardinality CRF models see Section 2.2.5). Formally, we define the CRF over N variables \mathbf{Y} as

$$P(\mathbf{Y} | \mathbf{X}) = \frac{1}{Z(\mathbf{X})} \exp \left\{ \sum_n \psi_1(Y_n, X_n) + \sum_{n,m} \psi_2(Y_n, Y_m, X_{nm}) \right\} \quad (2.3)$$

where X_n are the singleton features, $X_{n,m}$ are the pairwise features, and $Z(\mathbf{X})$ is the (feature dependent) partition (normalization) function. The singleton and pairwise potentials are defined as $\psi_1(y_n, x_n) = \theta_{y_n}^T x_n$ and $\psi_2(y_n, y_m, x_{nm}) = \theta_{y_n, y_m}^T x_{nm}$ respectively. Here the subscript on the θ indicates that a separate set of parameters (weights) exists for each assignment to the Y_n or pair (Y_n, Y_m) .

A pairwise CRF instance is defined by the assignment to its variables (possible unobserved), singleton and pairwise features, and graph structure. The class `svlPairwiseInstance` encapsulates a pairwise CRF instance.

The following example shows how to define a Potts model with singleton potentials of the form $\psi(Y_n = k, X_n = \mathbf{x}) = e_k^T \mathbf{x}$, i.e., \mathbf{x} is the prior unnormalized log-likelihood of Y_n taking label k .

```
// initialize model
svlPairwiseCRFWeights weights(K, K, 1);
vector<double> Xn(K);
for (int Yn = 0; Yn < K; Yn++) {
    fill(Xn.begin(), Xn.end(), 0.0);
    Xn[Yn] = 1.0;
    weights.singletonAddWeighted(Yn, Xn);
}

vector<double> Xnm(1, 1.0);
for (int Yn = 0; Yn < K; Yn++) {
    for (int Ym = 0; Ym < K; Ym++) {
        weights.pairwiseAddWeighted(Yn, Ym, Xnm, 1.0);
    }
}

svlPairwiseCRFModel model(weights);

// evaluate on instance
svlPairwiseCRFInstance instance;
int varId = 0;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        instance.Yn.push_back(-1);
        instance.Xn.push_back(vector<double>(K, 0.0));
        for (int k = 0; k < K; k++) {
            instance.Xn.back()[k] = drand48();
        }
        if (i > 0) {
            instance.edges.push_back(make_pair(varId, varId - M));
        }
        if (j > 0) {
            instance.edges.push_back(make_pair(varId, varId - 1));
        }
        varId += 1;
    }
}

instance.Xnm.resize(instance.edges.size(), Xnm);

vector<vector<double>> marginals;
model.inference(instance, marginals);
```

I/O File Format

Pairwise CRF models have XML format:

```
<crfWeights version="2" classes="..." singletonFeatures="..." pairwiseFeatures="..." >
  <!-- required weights for singleton potentials -->
  <singletonWeights>
    ...
  </singletonWeights>
```

```

<!-- optional weights for pairwise potentials -->
<pairwiseWeights>
  ...
</pairwiseWeights>

<!-- optional mask bits -->
<singletonMasks>
  ...
</singletonMasks>

</crfWeights>

```

Pairwise CRF instances have XML format:

```

<crfInstance nodes="..." edges="..." weight="..." >
  <!-- edge list between adjacent nodes of the form N_a N_b -->
  <edges>
    ...
  </edges>

  <!-- singleton feature vectors (one per node) -->
  <singletonFeatures>
    ...
  </singletonFeatures>

  <!-- pairwise feature vectors (one per edge) -->
  <pairwiseFeatures>
    ...
  </pairwiseFeatures>

  <!-- groundtruth assignments (one per node, if known) -->
  <targetAssignment>
    ...
  </targetAssignment>

</crfInstance>

```

2.2.5 General Log-Linear CRF Models

The `svlGeneralCRF` class encapsulates log-linear conditional Markov random field models over a set of discrete random variables $\mathbf{Y} = \{Y_1, \dots, Y_N\}$ given continuous random variables (or features) \mathbf{X} and parameters Θ . Each Y_n can have a different cardinality, $K_n \geq 2$. The Y_n 's take assignments $0, \dots, K_n - 1$. Formally, we have

$$P(\mathbf{Y} | \mathbf{X}; \Theta) = \frac{1}{Z(\mathbf{X})} \exp \left\{ \sum_{m=1}^M \psi_m(\mathbf{Y}_m, \mathbf{X}_m; \theta) \right\} \quad (2.4)$$

where the $\mathbf{Y}_m \subseteq \mathbf{Y}$ are cliques over variables and \mathbf{X}_m is the corresponding subsets of features for the m -th clique. The factors m specify log-linear functions, i.e.,

$$\psi_m(\mathbf{y}_m, \mathbf{x}_m; \theta) = \theta(\mathbf{y}_m)^T \mathbf{x}_m(\mathbf{y}_m) \quad (2.5)$$

where $\theta(\mathbf{y}_m)$ and $\mathbf{x}(\mathbf{y}_m)$ indicate the subset of parameters and features contributing to the factor when $\mathbf{Y}_m = \mathbf{y}_m$.

Note that in many cases the structure of the ψ_m will repeat, e.g., pairwise CRFs. In addition, each instance of a problem may have a different structure (number of variables, etc.), but share the same types of factors and model parameters. Therefore we will make use of factor templates for describing how features and parameters map to the various factor entries. Thus we have $\psi_m = \psi_{t(m)}$ where $t(m) \in \{0, \dots, T-1\}$ indexes a pre-specified template.

We can now specify the CRF in terms of two data structures: a model $\mathcal{M} = \{\Theta, \psi_t : t = 0, \dots, T - 1\}$ and an instance $\mathcal{I} = \{y_n, k_n, \mathbf{y}_m, \mathbf{x}_m, t_m : n = 0, \dots, N - 1; m = 0, \dots, M - 1\}$. The model holds the weights (parameters) and factor templates, while the instance describes the variables, cliques, features and assignment of factors to cliques. Factor templates, ψ_t , define $\theta(\mathbf{y}_m)$ and $\mathbf{x}_m(\mathbf{y}_m)$ for each assignment to \mathbf{y}_m by listing the associated weight and features indices.

Special cases.

- A variable value of $y_n < 0$ in \mathcal{I} means unobserved or unknown.
- A feature index $[\mathbf{x}_m(\mathbf{y}_m)]_i < 0$ in \mathcal{M} means constant feature 1.
- A weight index $[\theta(\mathbf{y}_m)]_i < 0$ in \mathcal{M} means constant weight 1 (i.e., don't train).

Parameter Learning

Since learning in CRF models is computationally challenging, parameters are learned using the pseudo-likelihood objective,

$$J(\theta; \mathcal{D}) = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{D}} \sum_n \log P(y_n | \mathbf{x}, \mathbf{y} \setminus Y_n; \theta) \tag{2.6}$$

I/O File Format

General CRF models have XML format:

```
<crfModel version="1" numWeights="..." numTemplates="...">
  <Weights>
    <!-- crf weights -->
  </Weights>

  <FactorTemplate>
    <!-- crf factor template -->
  </FactorTemplate>
  ...
</crfModel>
```

General CRF instances have XML format:

```
<crfInstance version="1" numVars="..." numCliques="...">
  <Cards>
    <!-- variable cardinalities -->
  </Cards>

  <Clique templateIndex="...">
    <!-- clique variables and features -->
  </Clique>
  ...
</crfInstance>
```

2.3 sv1ML

The sv1ML library provides basic machine learning capability such as classifiers and probability distributions. Currently, some of the classes utilize the OpenCV machine learning library code, but this dependency will be removed in future releases.

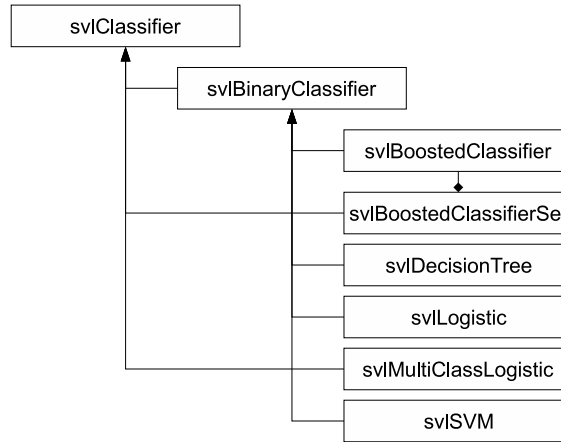


Figure 2.1: SVL classifier class hierarchy.

2.3.1 Classifiers

The **STAIR Vision Library** implements a number of standard machine learning classifiers. The classifiers are trained discriminatively given a set of training M samples $\{\mathbf{x}^{(m)}, y^{(m)}\}_{m=1}^M$ where $\mathbf{x}^{(m)} \in \mathbb{R}^n$ are the features and $y^{(m)} \in \{0, \dots, K-1\}$ is the class label for the m -th training sample, respectively. Some classifiers allow for the training examples to be weighted by some non-negative value, $w^{(m)} \geq 0$. In general, the **SVL** considers a negative training label to be unknown and is not considered during training or evaluation.

All classifiers in the **SVL** are derived from the `svClassifier` base class. Individual classifiers may have additional or specialized methods (for example, binary classifiers allow training from separate positive and negative training sets). Classifier specific parameters, for example, number of boosting rounds, are controlled through the `svOption` interface. A standard classifier provides the following interface:

```

class svClassifier : public svOptions {
    svClassifier();
    svClassifier(unsigned n, unsigned k);
    svClassifier(const svClassifier &c);
    virtual ~svClassifier();

    // access functions
    int numFeatures() const;
    int numClasses() const;
    bool trained() const;

    // i/o functions
    void initialize(...);
    bool save(...);
    bool load(...);

    // training functions
    double train(...);

    // evaluation functions
    void getClassScores(...) const;
    void getMarginals(...) const;
    int getClassification(...) const;
    void getClassifications(...) const;
};
  
```

The `load()` method is used when it is known in advance that the file to be loaded is of the same derived type as the object it is called on. When the type of the classifier stored in the file is not known at compile time,

`svlClassifierFactor().load()` should be used to allocate a new classifier object loaded from file instead. Typically, objects derived from `svlClassifier` are stored in XML files with a root named “svlClassifier”, but it is possible to define other file formats. For details on both these topics, see 2.1.14.

Most of the other methods are fairly self-explanatory.

Boosted Decision Tree Classifiers

The `svlBoostedClassifier` and `svlBoostedClassifierSet` encapsulate binary boosted decision tree classifiers based on real-valued feature vectors. Client code can control the boosting method (“GENTLE”, “DISCRETE”, or “LOGIT”), number of boosting rounds, number of splits per decision tree, trim rate per boosting round, and class pseudo counts.

Warning: *The code currently wraps the OpenCV boosting implementation which limits the size of the training sets and file I/O. The underlying implementation may change in future revisions.*

Binary and Multi-class Logistic Classifiers

The `svlLogistic` class implements a binary logistic regression classifier,

$$P(Y = 0 \mid X = \mathbf{x}) = \frac{1}{1 + \exp(-\theta^T \mathbf{x})} \quad (2.7)$$

The classifier is trained via Newton’s method using a weighted L_2 objective, i.e.,

$$\theta^* = \operatorname{argmin}_{\theta} \sum_m w^{(m)} \left\| y^{(m)} - P\left(y^{(m)} \mid \mathbf{x}^{(m)}; \theta\right) \right\|^2. \quad (2.8)$$

The `svlMultiClassLogistic` class supersedes the `svlLogistic` class and implements a multiclass logistic regression classifier,

$$P(Y = y \mid X = \mathbf{x}) = \begin{cases} \frac{\exp(\theta_y^T \mathbf{x})}{1 + \sum_{k=1}^K \exp(\theta_k^T \mathbf{x})} & \text{for } 0 \leq y < K - 1 \\ \frac{1}{1 + \sum_{k=1}^K \exp(\theta_k^T \mathbf{x})} & \text{for } y = K - 1 \end{cases} \quad (2.9)$$

The multi-class logistic is trained via LBFGS to maximize the (weighted) likelihood objective

$$\theta^* = \operatorname{argmax}_{\theta} \sum_m w^{(m)} \log P\left(y^{(m)} \mid \mathbf{x}^{(m)}; \theta\right). \quad (2.10)$$

Decision Trees

The `svlDecisionTree` class implements a binary decision tree classifier. The classifier can be trained to arbitrary depths via its `train` method:

```
double train(const MatrixXd &X, const VectorXi &y, int depth);
```

An argument of 1 to `depth` trains a single-node decision stump, while an argument of n trains n levels of decision/branching nodes. Training is done via calculation of the Gini coefficient. After training, tree nodes can be trimmed that contain too few positive or negative samples.

The `svlDecisionTreeSet` class implements a binary classifier based upon an ensemble of decision trees, d_i , each with an associated weight a_i . The final decision is given as $\sum_{i=1}^n a_i d_i(\mathbf{x})$. The `svlDecisionTreeSet` has no `train` method, but it can be used to wrap the results returned from boosting on `svlDecisionTree`, as that procedure returns an array of trees along with normalized weights.

Support Vector Machines

The `svlSVM` class is a wrapper around the `libSVM` library. It can be trained using a linear, polynomial or RBF kernels, and allows for cross-validation of the regularization parameter `C` and the width of the RBF kernel γ . See Appendix B for all the command line options. Cross-validation can only be specified from the configuration file instead of the command line and has the following form (described in detail below):

```
<svlML .svlSVM>
  <cvparam
    name = "C"
    min = "'-15'"
    max = "'3'"
    type = "'pow'"
    delta = "'1'"/>
  <cvparam
    name = "gamma"
    min = "'-5'"
    max = "'15'"/>
</svlML .svlSVM>
```

The `name`, `min` and `max` values are required if specifying a `cvparam`, although it's possible to only cross-validate one of the two parameters (or none). The `type` parameter is `'POW'` by default, which means that the search will proceed in powers of 2, starting from 2^{min} , and proceeding in increments of 2^{delta} , where `delta` is 1 by default. Thus for the parameter `gamma` above the explored values are

$$\{2^{-5}, 2^{-4}, 2^{-3}, \dots, 2^{13}, 2^{14}, 2^{15}\}$$

If the setting for `delta` was 2, the values would be

$$\{2^{-5}, 2^{-3}, 2^{-1}, \dots, 2^{11}, 2^{13}, 2^{15}\}$$

If `type` was `'LINEAR'` instead of `'POW'`, and the values would be

$$\{-5, -4, -3, \dots, 13, 14, 15\}$$

Note that unless the kernel is RBF, the `gamma` parameter is never used and will be ignored even if specified in the configuration file.

2.3.2 Feature Whitening

The `svlFeatureWhitener` class learns the mean and variance of each feature in a set of training features vectors. It then *whitens* the feature vectors by subtracting the mean and dividing by the variance of each feature independently. Features with zero-variance (e.g., so-called bias terms) are set to one. The means and variances can be saved to disk so that they can be applied at test time.

A typical use for feature whitening is in conjunction with a logistic classifier:

```
int main(int argc, char *argv[])
{
  vector<vector<double>> x;
  vector<int> y;

  // load training data into x and y
  ...

  // train classifier with whitened features
  svlFeatureWhitener whitener(x[0].size());
  whitener.train(x);
  whitener.evaluate(x);

  svlMultiClassLogistic classifier(x[0].size(), max(y) + 1);
  classifier.train(x, y);
}
```

```

// load evaluation data into x and y
...

// evaluate classifier and print confusion matrix
vector<int> y_hat;
whitener.evaluate(x);
classifier.getClassifications(x, y_hat);

svlConfusionMatrix confusion(classifier.numClasses());
confusion.accumulate(y, y_hat);
confusion.print(cout);

return 0;
}

```

2.3.3 Sufficient Statistics

The `svlSufficientStatistics` class accumulates sufficient statistics (moments) and conditional sufficient statistics:

$$\text{count} = \sum_i 1[y_i = k] \quad \text{sum} = \sum_i 1[y_i = k]x_i \quad \text{sumSq} = \sum_i 1[y_i = k]x_i x_i^T$$

Second-order sufficient statistics can be restricted to diagonal elements only (thus reducing storage and computational requirements from n^2 to n).

2.3.4 Multi-variate Gaussians and Conditional Gaussians

The `svlGaussian` encapsulates a Gaussian distribution over continuous random variables $\mathbf{x} \in \mathbb{R}^n$:

$$P(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}} \det \Sigma^{\frac{1}{2}}} \exp \left\{ -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu}) \right\} \quad (2.11)$$

The log-likelihood of a new data sample can be evaluated using the `evaluate()` or `evaluateSingle()` functions. The distribution can be learned from sample data or sufficient statistics using the `train()` methods. Random samples can be drawn from the distribution using the `sample()` methods.

The `svlConditionalGaussian` encapsulates a conditional Gaussian distribution. The class is optimized to pre-compute the *gain* matrices to allow for fast conditioning on new input. The following code demonstrates how to initialize a Gaussian distribution and then condition on a particular subset of the variables.

```

VectorXd mu(3);
mu << 1.0, 2.0, 3.0;
MatrixXd sigma(3, 3);
sigma << 1.0, 0.1, 0.2, 0.1, 1.0, -0.5, 0.2, -0.5, 1.0;

svlGaussian gaussian(mu, sigma);

cout << "Mean:\n[\" << gaussian.mean() << "\t]\n";
cout << "Variance:\n[\" << gaussian.covariance() << "\t]\n";

// compute P(X_0 | X_1 = 1.9, X_2 = 3.2)
vector<double> x(2);
vector<int> indx(2);

x[0] = 1.9; indx[0] = 1;
x[1] = 3.2; indx[1] = 2;

svlConditionalGaussian conditionalGaussian = gaussian.conditionOn(indx);
svlGaussian gaussian2 = conditionalGaussian.reduce(x);

```

```

cout << "Mean:\n[\t" << gaussian2.mean() << "\t]\n";
cout << "Variance:\n[\t" << gaussian2.covariance() << "\t]\n";

// alternative approach produces conditional distribution with a single call
svlGaussian gaussian3 = gaussian.reduce(x, indx);

cout << "Mean:\n[\t" << gaussian3.mean() << "\t]\n";
cout << "Variance:\n[\t" << gaussian3.covariance() << "\t]\n";

```

Sampling from a Gaussian

The `svlBoxMuller` class implements the polar Box Muller algorithm for sampling from a $\mathcal{N}(0,1)$ one-dimensional Gaussian distribution.

2.3.5 Linear Regression

The class `svlLinearRegressor` implements linear regression using an L_2 or Huber penalty for fitting the data points. Each data point can also be independently weighted as shown in the following code snippet:

```

vector<vector<double>> features = getFeatures();
vector<double> target = getTarget();
vector<double> weight = getWeights();

svlLinearRegressor J(true, 1.0); // use Huber penalty
J.initialize(features[0].size(), features, target, weight);
J.solve(1000, true); // solve (and show progress)

// print coefficients
for (int i = 0; i < J.size(); i++) {
    cout << "x[" << i << "] = " << J[i] << "\n";
}

```

2.3.6 Disjoint Sets

Implements a forest of disjoint sets abstract data type. The elements are numbered from 0 to (`size() - 1`). Each element belongs to exactly one set. The sets have ID in the range $[0, \text{size}())$. To get the number of elements of each set, call `size(id)` with a valid set ID.

```

int main(int argc, char *argv[])
{
    // create disjoint sets
    const int n = 100;
    svlDisjointSets sets(n);

    // randomly merge some sets
    for (int i = 0; i < n; i++) {
        int s1 = sets.find(rand() % n);
        int s2 = sets.find(rand() % n);
        if (s1 != s2) {
            sets.join(s1, s2);
        }
    }

    // show sets
    vector<int> s = sets.getSetIds();
    for (vector<int>::const_iterator it = s.begin(); it != s.end(); it++) {
        cout << toString(sets.getMembers(*s)) << endl;
    }

    return 0;
}

```

```
}
```

2.3.7 Vector-Quantization

The `svlCodebook` class implements a generic vector-quantization codebook. The codebook can be constructed externally and loaded using `initialize()` or learned from data using the `learn()` method. Vectors can then be discretized using `encode()`; codewords can be decoded to a representative vector using `decode()`.

K-means

Separately from the `svlCodebook` class there is also a templated `svlKMeansT` class which runs K-means clustering given a set of data points.

2.3.8 Histograms

The `svlHistogram` and `svl2dHistogram` are defined in `svlHistogram.h`. The simplest usage is to create a one-dimensional histogram with n bins of size 1 each, encompassing the range $[0, n)$, and insert values into it:

```
int numBins = 10;
svlHistogram hist(numBins);
bool bSuccess = hist.insert(5); SVL_ASSERT(bSuccess);
bSuccess = hist.insert(3); SVL_ASSERT(bSuccess);
bSuccess = hist.insert(10); SVL_ASSERT(bSuccess); // will fail
```

Other options include:

- explicitly specifying the range for the histogram values
- providing a weight for each example when calling `insert()`
- using `insertLinearSmooth()` to do linear smoothing for insertion so the sample's weight is linearly distributed between the two closest bins
- making the histogram circular via a call to `makeCircular()`, so on insertion values that are outside the valid range are wrapped around (useful e.g. when working with a histogram of directions)
- providing a pointer to a `vector<double>` for the class to use for its memory storage, thereby eliminating the need to later copy the memory out of the histogram once it's populated.
- optionally providing an integer `offset` into the above vector, which will result in the histogram populating the values from `offset` to `offset + numBins` (resizing the vector as needed).
- creating a 2-dimensional histogram with the above options still available

2.3.9 Quadratic Program Solver

The class `svlQP Solver` uses an infeasible start Newton method [1] to solve small-scale quadratic programs of the form

$$\begin{aligned} \text{minimize (over } x) \quad & \frac{1}{2}x^T Px + q^T x + r \\ \text{subject to} \quad & Ax = b \\ & l \leq x \leq u \end{aligned}$$

In the future, we will also implement algorithms such as LQOQ [24] and specialized routines using variants of the SMO algorithm [7] handle the case of $A \in \mathbb{R}^{1 \times n}$ (single equality constraint) or $A \in \{0, 1\}^{m \times n}$ (simplex constraints).

2.3.10 Feature Selection

The `svlFeatureSelector` is a generic abstract class for selecting informative features. It provides only one public function for the user to call, and is very simple to use:

```
int main(int argc, char *argv[])
{
    vector<vector<double>> posFeatureVectors;
    vector<vector<double>> negFeatureVectors;

    // load feature vectors of positive and negative examples
    // (posFeatureVectors[i][j] is the value of feature j on
    // example i)
    ...

    // since the svlFeatureSelector is an abstract class,
    // choose which of its subclasses to use for filtering
    svlFeatureSelector *selector = new svlFeatureSelectorSubclass();

    // perform filtering
    vector<bool> selectedFeatures;
    selector->filter(posFeatureVectors, negFeatureVectors, selectedFeatures);

    // trim the feature vectors of positive examples
    for (unsigned i = 0; i < posFeatureVectors.size(); i++) {
        for (int j = posFeatureVectors[i].size() - 1; j >= 0; j--) {
            if (!selectedFeatures[j]) {
                posFeatureVectors[i].erase(j);
            }
        }
    }

    // do the same for negative examples
    ...

    return 0;
}
```

The user can specify the minimum and maximum number of features to be selected; see Appendix B for configuration details.

Mutual information

Warning: The `svlClassifierResponseList` has been deleted from the library; use this instead

The `svlMIFeatureSelector` is a subclass of `svlFeatureSelector`. It implements feature selection using joint mutual information as described in [25]. The `svlFeatureResponse` class within `svlMIFeatureSelector.h` is a helper class that encompasses the mutual information computations between a given feature and the class label. Mutual information between a discrete feature X and a class label C is defined as

$$I(X; C) = \sum_{c \in C, x \in X} P(c, x) \log \frac{P(c, x)}{P(c)P(x)}$$

The *maximum* mutual information between a feature and the class label, as well as the threshold at which it is achieved, can be computed with the `findHighestMI()` function. `binarizeFeature()` can be used to threshold the feature responses and create a binary feature.

The *joint* mutual information of two features given the class label is defined as

$$I(X, Y; C) = \sum_{c \in C, x \in X, y \in Y} P(c, x, y) \log \frac{P(c, x, y)}{P(c)P(x, y)}$$

and is implemented in `computeJointMI()`. The joint mutual information computation assumes that both features have been converted to binary.

The user can specify the minimum increase in joint mutual information required for a new feature to be selected; see Appendix B for configuration details.

2.3.11 Evaluation

Once the classifiers are trained, there are some standard classes that provide useful evaluation methods.

Precision-Recall Curves

The `svlPRCurve` class (derived from `svlClassifierSummaryStatistics`) encapsulates the accumulation of classification results and computation of precision/recall-derived statistics such as F_1 -score and average precision (or area under the PR-curve). The class accumulates three types of events: positives, negatives and misses. A positive and negative events count the scored output of the classifier on positive and negative examples, respectively. Miss events count positive examples that are not even considered by the classifier (for example, due to pruning by non-maxima suppression). The `INCLUDE_MISSES` flag indicates whether misses should be counted as zero-probability positive events or simply ignored. Positive and negative examples can be weighted differently. In particular, the `normalize()` function will set the weight of positive events so as to equalize the weighted sum of positive and negative examples.

The member function `writeCurve()` will generate a file that can be plotted using the `plotPR.m` script in the `svl/scripts` directory.

Confusion Matrices

`svlConfusionMatrix` is a utility class for computing and printing confusion matrices. A negative actual/predicted class is considered unknown and not counted. The rows in the confusion matrix represent the *actual* class and the columns represent the *predicted* class. The confusion matrix can be printed with either rows or columns (or both) normalized. You can also print the class-specific precision and recall metrics.

Tip: You can change the row and column separators in the confusion matrix, e.g., for easy cut-and-paste into $L^A_T E^X$ tables, using the `svlML.svlConfusionMatrix` configuration parameters.

2.4 svlVision

2.4.1 OpenCV Utilities

The `svlOpenCVUtils.h` file provide a number of useful conversion functions that wrap around the `OpenCV` library.

FUNCTION	DESCRIPTION
toString	Converts a <code>CvPoint</code> , <code>CvRect</code> , <code>CvSize</code> , <code>CvMat</code> , <code>IplImage</code> or <code>vector<IplImage*></code> to an <code>stl::string</code>
toMatlabString	Converts a <code>CvMat</code> to an <code>stl::string</code> that's in Matlab-readable format
dump	Writes the data from an <code>IplImage</code> or <code>CvMat</code> to <code>stdout</code>
readMatrix	Reads the data from a text file into a pre-allocated <code>CvMat</code>
writeMatrix	Writes the data from a <code>CvMat</code> into a text file
readMatrixAsIplImage	Reads the matrix from a text file into an <code>IplImage</code>
writeMatrixAsIplImage	Writes out an <code>IplImage</code> as a matrix to a text file
makeIplImage	Converts a byte stream to an <code>IplImage</code>
makeByteStream	Converts an <code>IplImage</code> to a byte stream
svlCreateMatrix	Converts a 2-dimensional <code>vector</code> to a <code>CvMat</code>
svlCreateImages	Creates a <code>vector</code> of <code>IplImages</code>
svlReleaseImages	Frees memory for a <code>vector</code> of <code>IplImages</code>
svlCreateMatrices	Creates a <code>vector</code> of <code>CvMat</code> matrices
svlReleaseMatrices	Frees memory for a <code>vector</code> of <code>CvMat</code> matrices
combineImages	Creates one big image from a <code>vector</code> of <code>IplImages</code>
eigenToCV	Converts <code>Eigen</code> data types to OpenCV <code>CvMats</code>
cvToEigen	Converts a <code>CvMat</code> to <code>Eigen::MatrixXd</code>

It also implements:

captureOpenCVerrors

Redirects `openCV` errors to the `svlLogger` error handler, which then provides helpful debugging information and terminates the program by calling the `abort()` function which can be trapped with a debugger. This function should be called once at the beginning of each application.

svlCreateHeatMap

Function to convert a matrix (with entries in the range $[0, 1]$) to a heatmap image (8-bit, 3-channel). The following figure shows an example of converting an image egde map to a heatmap:

```
// load the image
IplImage *img = cvLoadImage(inputFilename , CV_LOAD_IMAGE_GRAYSCALE);

// create the soft edge map
CvMat *softEdgeMap = cvCreateMat(img->height , img->width , CV_32FC1);
svlSoftEdgeMap edgeMapCalculator;
cvConvertScale(edgeMapCalculator.processImage(img , true) , softEdgeMap);
scaleToRange(softEdgeMap , 0.0 , 1.0);

// create the heat map
IplImage *heatImg = svlCreateHeatMap(softEdgeMap , SVL_COLORMAP_RAINBOW);
cvSaveImage(outputFilename , heatImg);

// free memory
cvReleaseImage(&heatImg);
cvReleaseMat(&softEdgeMap);
cvReleaseImage(&img);
```

2.4.2 Vision Utilities

The `svlVisionUtils.h` file provides a number of helpful utility functions, summarized below. It also defines the equality operator on `CvPoint` and `CvRect` and the relational `<` operator on `CvPoint`.

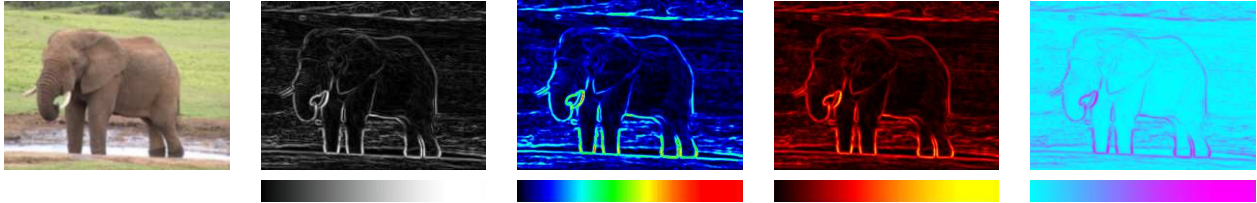


Figure 2.2: Heatmap example: (a) original image, (b) soft edge map, (c)-(e) rainbow, hot and cool heatmaps.

Simple arithmetic operations

FUNCTION	DESCRIPTION
svlAddSquared	Computes a per-element sum of squares of two CvMat
svlSubtractMedian	Normalizes a 32F IplImage based on its median
svlMin	Finds the min element of a CvMat
svlMax	Finds the max element of a CvMat
svlFindLocalMaxima	Finds a list of elements each of which is maximal within a $w \times h$ window of a cvMat
svlMeanSquaredError	Finds the average squared difference between pixels in two CvMat
svlNGC	Computes the normalized greyscale correlation of two CvMat
svlFloatCompare	Returns true if two floats differ by $\geq 1\%$ of the larger magnitude
svlIntersection	Finds the intersection of two CvRect
svlFitBoundingBox	Finds the bounding box around a set of CvRects
svlFitBoundingBox	Finds the bounding box around all the black pixels in a binary image
svlCenterOfMass	Finds the center of mass of the black pixels in a binary image
svlContainsNanOrInf	Checks the CvMat
svlImageUniform	Returns true if all values in the 8U or 32F IplImage are the same
svlImageAlmostUniform	Returns true if min and max values in the 8U IplImage differ by ≥ 32 , or in the 32F differ by ≥ 0.125

Scaling and resizing operations

FUNCTION	DESCRIPTION
scaleToRange	Scales a CvArr given the min and max values
resizeInPlace	Resizes an IplImage
cropInPlace	Crops an IplImage
svlClipRect	Clips parts of a CvRect that are outside a bounding box or IplImage
svlFitRect	Moves a CvRect to fit inside a bounding box or IplImage
svlIncreaseToAspectRatio	Increase the CvRect to match the aspect ratio
svlDecreaseToAspectRatio	Decrease the CvRect to match the aspect ratio

Image type conversions

FUNCTION	DESCRIPTION
convertInPlace	Performs in-place cvConvertScale() on an IplImage
create32Fimage	Creates a new 32F IplImage from an 8U or 32F one

svlColorType

This enum defines the type of color spaces that the `svlImageLoader` class discussed in 2.4.6 and the `svlChangeColorModel()` function discussed below can support.

- `SVL_COLOR_GRAY` is a grayscale image
- `SVL_COLOR_BGR` is the standard format a color image is loaded in
- `SVL_COLOR_BG` corresponds to a two-channel image obtained from the one above by averaging together the B and G channels and leaving R intact (similarly for `SVL_COLOR_BR`, `SVL_COLOR_GR`)
- `SVL_COLOR_R`, `SVL_COLOR_B`, `SVL_COLOR_G` correspond to only leaving the specified channel
- `SVL_COLOR_HSV` is the standard HSV format
- `SVL_COLOR_HS`, `SVL_COLOR_HV`, `SVL_COLOR_H` correspond to only leaving the specified channels
- `SVL_COLOR_YCrCb` is the standard YCrCb format
- `SVL_COLOR_CrCb` corresponds to only leaving the specified channels of the image above
- `SVL_COLOR_UNDEFINED` is useful for non-standard images (e.g. depth images)
- `SVL_COLOR_ERROR` is useful as a potential return value from functions parsing the user input

The `svlColorTypeToText()` function is useful for debugging.

Color utilities

FUNCTION	DESCRIPTION
<code>greyImage</code>	Converts an <code>IplImage</code> to grayscale
<code>colorImage</code>	Converts a grayscale <code>IplImage</code> to 8U BGR
<code>superSaturateImage</code>	Super-saturates a color <code>IplImage</code>
<code>svlContrastNormalize</code>	Increases the contrast in a grayscale <code>IplImage</code>
<code>svlChangeColorModel</code>	Changes a BGR <code>IplImage</code> to another color (cf. <code>svlColorType</code> enum)
<code>svlDeleteChannel</code>	Deletes the specified channel of a 3-channel <code>IplImage</code>
<code>svlLeaveSingleChannel</code>	Leaves only a single channel of an <code>IplImage</code>
<code>svlAverageColorChannels</code>	Given a 3-channel 8U <code>IplImage</code> and a channel <code>i</code> , averages together the other two channels while leaving channel <code>i</code> intact
<code>svlAverageChannels</code>	Averages together all channels of an 8U <code>IplImage</code>

3-d utilities

FUNCTION	DESCRIPTION
<code>estimatePointNormals</code>	Estimates the point normals from 3d point cloud projection into image
<code>estimatePointNormalsFast</code>	Same as above, but for fixed sized 3×3 window
<code>estimatePlane</code>	Estimates the 3d planar fit to a set of points
<code>svlRotatePointCloud</code>	Rotates a dense point cloud represented as three matrices.
<code>svlTranslatePointCloud</code>	Translates a dense point cloud represented as three matrices.

Other utilities

FUNCTION	DESCRIPTION
<code>svlConnectedComponents</code>	Renumbers the connected components (in place)
<code>svlIsConnectedComponent</code>	Returns true if the given component is 4- or 8-connected
<code>svlNearestNeighbourFill</code>	Fill zero points with value from nearest-neighbor
<code>svlInsidePolygon</code>	Returns true if a given <code>CvPoint</code> is inside the given polygon
<code>svlPolynomialFit</code>	Fits a polynomial to a set of <code>CvPoint</code>
<code>svlLineFit</code>	Fits a line to a set of <code>CvPoint</code>

2.4.3 Basic types

svlImageBufferManager

The `svlImageBufferManager` class reserves a certain number of `IplImage` buffers and reuses them without allocating and de-allocating memory every time. The buffers are automatically resized as needed. This is useful if within your application you need multiple 640×480 image buffers to perform intermediate computations for every image you're processing, but you want to (1) avoid explicitly calling `cvCreateImage()` and `cvReleaseImage()` many times and (2) implicitly deal with the case where some images might actually be a different size from the default.

svlFeatureVectors

The `svlFeatureVectors` class is used to store feature vectors at different locations and scales. It is currently only used within the sliding windows framework.

2.4.4 Object detection storage types

When dealing with object detection and classification, the following classes defined in `svlObjectList.h` might be useful.

svlObject2d

This class contains the x, y, w, h coordinates of the bounding box around the object, as well as its *name*, the detection *probability*, and an *index* that can be used for external reference.

svlObject2dFrame

This is subclass of `std::vector<svlObject2d>` usually corresponding to all objects detected within an image. There are various useful methods and functions for processing an `svlObject2dFrame` including reading and writing utilities for XML files, scaling of all objects, filtering based on name or overlap within itself or with another `svlObject2dFrame`, non-maximal suppression of detections, and sorting objects by detection probability.

svlObject2dSequence

This is subclass of `std::map<string, svlObject2dFrame>`, and can be used to hold one list of objects per image frame for the entire video or image sequence. The map is indexed by filename (or ID), or in the case of videos by a (0-based) frame index. Again various functions that defined for reading, writing and scaling an `svlObject2dSequence`, as well as for counting the objects, removing specified frames, filtering based on overlap with objects from another `svlObject2dSequence`, doing non-maximal suppression, and computing the average aspect ratio of all objects.

2.4.5 I/O file format

Most vision applications take as an input an XML file corresponding to a directory with a list of images to analyze, e.g.

```
<ImageSequence
  dir = "directoryName/"
  version = "1.0">
  <Image name="image001.jpg" />
  <Image name="image002.jpg" />
</ImageSequence>
```

The `svlImageSequence` class reads and writes these files and stores all the necessary information. It provides useful access functions such as the `operator[]` which returns the name of the i^{th} image stored, or the function `image()` which loads and returns the i^{th} `IplImage`.

Two other helpful functions in `svlImageSequence.h` are `hasHomogeneousExtensions()` which returns true if all images in the image sequence have the same extension and `getExtension()`.

The `svlImageRegionsSequence` is a subclass of `svlImageSequence` which can deal with sub-regions of images instead of full image files. This is particularly useful when cropping out small training examples from full scenes (see section 2.4.7). Most of the functionality of `svlImageSequence` is implemented for the subregions as well.

The ground truth files are also represented in XML format:

```
<Object2dSequence version="1.0">
  <Object2dFrame id="image001">
    <Object name="mug" x="368.604" y="385.592" w="76.3774" h="78.4834" pr="1" />
    <Object name="cup" x="275.623" y="227.488" w="45.3836" h="83.0332" pr="1" />
    <Object name="cup" x="197.031" y="230.9" w="44.2767" h="54.5972" pr="1" />
  </Object2dFrame>
  <Object2dFrame id="image002">
    <Object name="cup" x="325.434" y="201.327" w="36.5283" h="48.91" pr="1" />
  </Object2dFrame>
</Object2dSequence>
```

They can be read using the `svlObject2dSequence` methods described above. Also, the `svlLabeledSequence` can read both the xml file for the images and the xml file for the labels. This can be a convenient way of accessing labeled frames through a single sequence object.

2.4.6 Loading images

The **STAIR Vision Library** object detector pipeline is set up to accept input data from a variety of different sources, e.g. the grayscale image of a scene and a laser scan providing depth data for every pixel of the image. The `svlImageLoader` class encapsulates most of the work associated with loading images in such a variety of formats. It supports a variety of *channels* of input, the most common ones being

- **INTENSITY** corresponding to a grayscale image
- **EDGE** corresponding to first loading a grayscale image and then computing its gradient edge map.¹
- **DEPTH** corresponding to loading a text file with depth values for every pixel

The user specifies how to load the channels from an image directory by specifying the appropriate extensions, e.g. the grayscale images might be contained in `.jpg` files while depth map readings might come from `.depth.txt` files. The base name of the file is used to match the images, so `image001.jpg` is assumed to correspond to `image001.txt`. Further, the `svlExtensionTree` class appropriately handles the case where extensions are substrings of each other, e.g. `.jpg` and `.processed.jpg`.

The list of channels is set through the configuration manager discussed in Section 2.1.5 using either of the following three options (the first two can also be used from the command line, using `-set`):

```
<svl>
  <svlVision.svlImageLoader channels="INTENSITY_*.jpg _DEPTH_*.depth.txt" />
</svl>
```

```
<svl>
  <svlVision.svlImageLoader channels="INTENSITY:*.jpg:DEPTH:*.depth.txt" />
</svl>
```

¹A variety of other color spaces can also be used for computing the gradient edge map. The channels in this case would be specified as `EDGE_<color type>`, e.g. `EDGE_HSV`. Further, the intensity image can be created by considering just one channel of an RGB image, e.g. `INTENSITY_R` would treat the red channel as the full intensity image. Please refer to the `textToColorType()` and `textToColorType()` in `svlImageLoader.cpp` for a complete list of channels supported for loading.

```

<svl>
  <svlVision.svlImageLoader>
    <channel type="INTENSITY" ext=".jpg" />
    <channel type="DEPTH" ext=".depth.txt" />
  </svlVision.svlImageLoader>
</svl>

```

One other useful feature is when the `useMask` attribute of `svlImageLoader` is set (optionally with the `maskExtension` specified), a binary mask corresponding to each image is loaded, using the same extension swapping scheme as for other channels, and certain applications use it to treat a subset of pixels within the image as unobserved.

Finally, the `svlImageLoader` class provides a variety of access functions, including

readDir

Uses the channel definitions to load all of the appropriate file paths for a directory.

readImageSeq

Loads in the `imageSequence`, strips off the extension from every filename and uses the appropriate channel extensions instead.

readCommandLine

Reads in the list of file names, each one corresponding to either an image sequence (.xml extension) or an image file; then reads the image sequences and stores all image names together for later access.

getAllFrames

Gets all of the multi-channel patches referred to by the file paths already loaded internally from a call to one of the read functions.

getFrame

Gets a single specific frame by index; meant to be called only after calling `read()`. Alternatively, can take a name of a file and load all channels by swapping the extension for each channel.

getSimpleImageFromFilename

Takes the name of a file, swaps the extension for the specified channel's extension, and loads the single image corresponding to this channel, resizing appropriately but without any color conversion or edge computation post-processing.

Example

These functions often return a `vector<IplImage*>` per base file name, where each `IplImage` corresponds to a different channel, loaded from the file name with the appropriate extension. The current object detection pipeline expects this as an input, and extract features from all the available channels (cf. Section 2.4.10). The following is an example of how the `svlImageLoader` can be used to load images.

```

int main(int argc, char *argv[])
{
  //get the image sequence filename
  const char *imageSeqFilename = ...

  svlImageLoader loader;
  loader.readImageSeq(imageSeqFilename);
}

```

```

for (int i = 0; i < loader.numFrames(); i++) {
    vector<IplImage *> image;
    bool success = loader.getFrame(i, image);

    // check success, process multi-channel image
    ...
}

return 0;
}

```

svlSoftEdgeMap

It is often useful to work with not only the intensity image but also to specifically focus on the edges in that image. The `svlSoftEdgeMap` can compute the gradient magnitude image from an intensity image using the Sobel operator for the directional derivatives. `svlSoftEdgeMap` can work with both grayscale and color images, and it is used by the `svlImageLoader` to create an edge map when the `EDGE` channel is specified.

2.4.7 Building a training dataset

The `svlTrainingDatasetBuilder`, `svlImageWindowExtractor` and `svlClipWriter` (implemented within `svlTrainingDatasetBuilder.h` classes are responsible for building a training dataset of image patches that can be used for object classification. The user provides an `svlImageSequence` of scenes and an `svlObject2dSequence` of corresponding groundtruth labellings to the `writeDataset()` function of `svlTrainingDatasetBuilder`. A variety of options can be set through the command line, cf. Appendix B. Note that the object names can be set either as an attribute `objects` as described in the Appendix, or individually, each an element `object` of `svlVision.svlTrainingDatasetBuilder` with attribute `name`.

The “overlap” between a window W and a ground truth bounding box G is defined by default as $\frac{\text{area}(W \cap G)}{\text{area}(W \cup G)}$ or, in the case of textures, as $\frac{\text{area}(W \cap G)}{\text{area}(W)}$. The positive examples are then extracted from groundtruth labellings in one of three ways. For each groundtruth bounding box, one can

- Take the patch contained in that bounding box
- Among all windows that would be considered by the sliding window detector (cf. Section 2.4.12), take the window with the greatest overlap with this bounding box.
- Take *all* sliding windows with significant (default: $\geq 50\%$) overlap with this bounding box.

For negative examples, only patches that don’t significantly overlap a positive groundtruth bounding box are considered, and the following extraction schemes are possible:

- False positive detections extracted from a provided `svlObject2dSequence` file
- All negative windows that would’ve been considered by the sliding window detector
- Objects from the groundtruth file that are not specified as positive, e.g. to build a dataset for mug classification, one can choose to explicitly include all *other* labeled objects in the negative set
- Random image patches from the provided scenes (note: these will not overlap any labeled objects by any amount)

Note that the current implementation ignores the `EDGE` channel, since `EDGE` maps can be computed on the fly later from intensity images. Also, patches that are close to uniform in the `INTENSITY` channel are ignored; no such checks are performed in any other channel.

The extracted patches are either written as individual image files into corresponding folders, one per each positive object and one for all negative examples, or simply listed in an XML file.

2.4.8 Pixel Filter Banks

svlConvolution

The `svlConvolution` class defines an interface for convolving an image with a large non-separable kernel function (for separable kernels, using OpenCV's `cvFilter()` function). Concrete instantiations for Gabor filters (`svlGaborConvolution`) and Laplacian-of-Gaussian filters (`svlLoGConvolution`) are also provided.

Gabor filters are defined by

$$G_r(x, y) = \exp\left(-\frac{X^2 + \gamma^2 Y^2}{2\sigma^2}\right) \cos\left(\frac{2\pi}{\lambda} X\right) \quad (2.12)$$

$$G_i(x, y) = \exp\left(-\frac{X^2 + \gamma^2 Y^2}{2\sigma^2}\right) \cos\left(\frac{2\pi}{\lambda} Y\right) \quad (2.13)$$

where $X = x \cos(\theta) + y \sin(\theta)$ and $Y = -x \sin(\theta) + y \cos(\theta)$ with orientation θ (in radians), aspect ratio γ , effective width σ and spatial frequency λ .

Laplacian-of-Gaussian filters are defined by

$$\text{LoG}(x, y) = \frac{1}{\pi\sigma^4} (r^2 - 1) \exp(-r^2) \quad (2.14)$$

where $r^2 = \frac{x^2 + y^2}{2\sigma^2}$.

Filtering is performed by calling the `filter()` function. The caller must provide CV32F destination image the same size as the source image. If the source image is not single channel, 32-bit it will be converted (wasting some time). The image is being temporarily downsized to ensure `height % 8 == 0` in order to be able to use the OpenCV `cvFilter2D()` function; if argument `padWithZeros` is true, the image is instead temporarily increased in size and padded with zeros. This changes the expected edge effects on the right/bottom parts of the image, but is necessary for very small images (i.e., height less than 8 pixels).

svlTextonFilterBank

The `svlTextonFilterBank` class defines a set of filters that are useful for multi-class image segmentation problems. The class produces a 17-dimensional feature vector for each pixel based on the filter bank defined in [26].



Figure 2.3: Illustration of texton filters. Note that the filters operate in CIELab color space (RGB shown for illustrative purposes only).

svlFilterBankResponse

The `svlFilterBankResponse` class collects multiple filter responses into a single data structure. The class facilitates computing statistics of the filter responses over rectangular regions. For example, `mean()` member function can be used to compute a vector of average filter responses over a small rectangular region around a pixel.

2.4.9 Interest point detection

The **STAIR Vision Library** provides a standardized interface for detecting interest points in images. This should make it easy to vary means of detecting interest points without modifying the surrounding code.

svlInterestPointDetector

The `svlInterestPointDetector` class is an abstract superclass that defines the interest point detection interface. If you want your interest point detection code to be polymorphic, you should implement it using pointers of this type.

The `svlScaledInterestPointDetector` is a subclass of `svlInterestPointDetector` and is meant to be used for scale-invariant detectors. The only added requirement is the `findInterestPointsScaled()` function which returns the corresponding scale for each detected interest point.

svlHarrisCornerDetector

This is just an implementation of Harris corner detection that wraps around OpenCV's corner detection.

svlHarrisLaplaceDetector

This is a subclass of `svlScaledInterestPointDetector` which implements Harris Laplace corner detection with automatic scale selection [17].

2.4.10 Feature extraction

The first step in most computer vision algorithms is extracting informative features from the image. The **STAIR Vision Library** provides a general framework for feature extraction, and implements a few useful feature extraction methods.

General framework

The `svlFeatureExtractor` class defines the general framework for feature extraction. By defining different subclasses of this abstract superclass, it is possible to define different types of features to extract.

Extracting features. Each feature extractor class produces a vector of feature values at a given window location. Different extraction methods allow windows to be processed individually or in batch mode (set the “sparse” argument to the `extract()` method to `true` in order to process windows individually), allowing the extractor to take advantage of optimization strategies such as sharing one integral image of the whole frame between all windows.

Input. Every feature extractor must be capable of receiving several channels of data as input, though typically each feature extractor will only work on a small number of channels. Currently, images are the only kind of data supported by the **STAIR Vision Library**, but it is possible to write your own feature extractor subclass that supports arbitrary types of data channels by subclassing the `svlDataFrame` class. The feature extractors work on a vector of pointers to `svlDataFrame` objects. All current implementations convert these to `svlImageFrame`, which is a wrapper around `Ip1Image`.

For an example of a complicated setup, one might use a channel of visible light images, a channel of edge images computed from the visible light images, a channel of “motion images” made by differencing consecutive frames of visible light video, and a channel of depth maps created by projecting the output of a range scanner into the image plane of the visible light camera. A patch dictionary might operate on the visible light, edge, and depth channels, while a Haar feature extractor might operate on the motion channel. See 2.4.6 for more information on loading multiple channels of data.

Factory. The `svlFeatureExtractorFactory()` function allows construction of feature extractors in cases where the type is not known until runtime. See 2.1.14 for a code example. The `svlCompositeFeatureExtractor`, derived off of `svlFeatureExtractor` allows multiple feature extractors to be combined into a single composite detector feature extractor.

Haar features

One of the feature extractor types currently implemented are Haar features [20] in `svlHaarFeatureExtractor`.

Haar features are simple features based on the difference of the sum of intensity values of pixels that fall into different rectangles. Figure 2.4 shows the feature templates specifying the locations of the rectangles for the Haar features implemented in the **STAIR Vision Library**. To extract a Haar feature from an image patch, imagine placing the template over that image patch and subtracting the sum of the values of all pixels that lie in a black rectangle from the sum of the values of all pixels that lie in a white pixel. The feature value is also normalized by dividing by the total intensity of the image patch.

Figure 2.5 demonstrates how a Haar feature can be computed efficiently given an integral image. Pre-computing the integral image once and using it to compute several Haar features can avoid unnecessarily re-computing several summations.

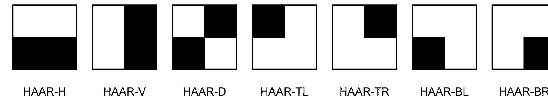


Figure 2.4: Some Haar feature templates.

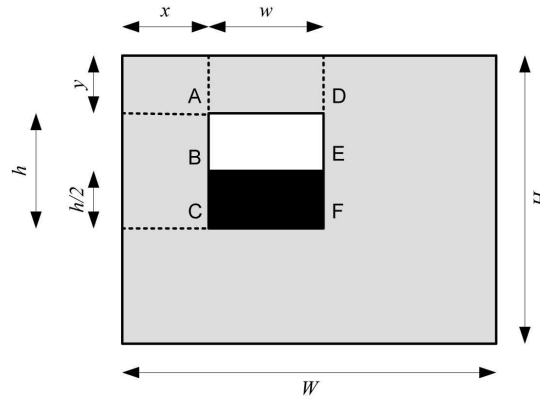


Figure 2.5: Given an integral image I , this Haar feature may be computed as $(I(E) - I(B) - I(D) + I(A) - (I(F) - I(E) - I(C) + I(B)))/(I(W, H) - I(W, 0) - I(0, H) - I(0, 0))$, where $(0, 0)$ is the upper-left corner of this figure.

Fragment-based features

Another type of feature extraction, used for object detection by Torralba et al. [23], is implemented in `svlPatchDictionary.h` and `svlPatchDefinition.h`. Briefly, a dictionary is built by extracting a set of random patches from the training examples, and recording the location within the image that each patch originated from. For each patch and for each training examples, the corresponding feature value is then computed by finding the maximum normalized cross-correlation between the patch and the training image within a small window around the original location of the patch.

Training examples The `svlPatchDictionary` class (which is a subclass of `svlFeatureExtractor`) relies on `svlPatchDefinition` to extract the features from sample images. Training examples for object classification correspond to cropped images of the object of interest. Each training example is represented by a `vector<IplImage*>`, as described in section 2.4.6, with one `IplImage` per input channel type (e.g. intensity image, edge map, depth map, etc.) The images are all assumed to be of size `svlPatchDictionary::_windowSize`. For simplicity of notation for the rest of this discussion, let's assume the training images are all 32×40 pixels in size.

Building the dictionary. The `buildDictionary()` function in `svlPatchDictionary` takes a vector of training examples. To build the patch dictionary, first for each training example, each input channel is normalized by either its mean or median value. Currently, the median value is used only for depth images which often contain outliers, e.g. reading at maximum range of the sensor. The mean is used for everything else. Then patches of random size, varying from 4 pixels to half the window size in each dimension, are extracted from random locations in each of the image channels. These patches from all of the training examples are used to create a dictionary of features.

Feature storage. The `svlPatchDefinition` class is used to store the features using an `IplImage` fragment template, a `CvRect` valid response region within the 32×40 image, and a valid channel specifying which type of input image (e.g. intensity image, edge map, depth map, etc.) this fragment was extracted from. The valid response region is currently defined to be a 7×7 pixel region around the source location (within the bounds of the 32×40 training image). The `svlIntensityPatchDefinition` and `svlDepthPatchDefinition` are the currently implemented feature subclasses, although they are mostly there just for historical reasons and currently implement only slightly different I/O.

Feature computation. Feature computation is implemented in the `patchValueHelper()` function of the `svlPatchDefinition` class. For a particular feature with template T , given a 32×40 image window W (normalized to have the mean or median of 0, as discussed below), we compute the feature value using normalized cross-correlation:

$$F(x, y) = \frac{\sum_{x', y'} [T(x', y') \times W(x + x', y + y')]}{\sqrt{\sum_{x', y'} T(x', y')^2 \times \sum_{x', y'} W(x + x', y + y')^2}} \quad (2.15)$$

where x', y' range over the size of the template T . There are various optimizations employed to make this computation more efficient, described in detail below. The feature value is

$$f = \max_{(x, y) \in \text{validRegion}} F(x, y)$$

Feature computation pipeline. The `extract()` function of `svlPatchDictionary` performs the feature computation steps. As described above for the general `svlFeatureExtractor` class, the input consists of an image I with one or more windows of interest within it. First, for each 32×40 window of interest, a normalization constant is computed for each channel of input, using the median for depth images and the mean for everything else. This computation is parallelized for efficiency. Then the `calcFV()` function is called in parallel on each entry in the dictionary. It computes the corresponding feature value for each window of interest. The `calcFV()` function is mainly used to manage the various optimization options and make the appropriate function calls in `svlPatchDefinition`.

Optimizations. If the windows of interest are “sparse”, as is for example the case during training, when the image I is not of an entire scene but of just one object of size 32×40 and the only location of interest is the 32×40 window at $(0, 0)$, the `patchValues()` function of `svlPatchDefinition` is used to compute the feature value at each window independently without any pre-computation. However, if the windows of interest are densely packed within the supplied image I , as is the case when sliding window object detection is performed (cf. Section 2.4.12), two optimizations described below are employed. The feature value within each window is then computed with `patchValue()` of `svlPatchDefinition`, using the precomputed response and integral images:

1. Within the `calcFV()` function, the `responseImage()` function of `svlPatchDefinition` is first called to obtain the cross-correlation between the feature template T and the entire image I , i.e.

$$R(x, y) = \sum_{x', y'} T(x', y') \times I(x + x', y + y')$$

Recall equation 2.15 and observe that the normalized window W is simply a part of the big image I after subtracting the normalization constant c_W . Thus

$$\begin{aligned} \sum_{x',y'} T(x',y') \times W(x+x',y+y') &= \sum_{x',y'} T(x',y') \times (I(\tilde{x}+x',\tilde{y}+y') - c_W) \\ &= \left(\sum_{x',y'} T(x',y') \times I(\tilde{x}+x',\tilde{y}+y') \right) - c_W \sum_{x',y'} T(x',y') \\ &= R(\tilde{x},\tilde{y}) + c_W \times \text{TEMPLATESUM} \end{aligned}$$

Since `TEMPLATESUM` is precomputed for each feature and doesn't depend on the current window being analyzed, the numerator of equation 2.15 is significantly simplified.

2. At the beginning of the `extract()` function, even before the `calcFV()` function is called on each dictionary entry, for each input image I , the integral images are computed for each channel:

$$\begin{aligned} \text{IMAGE\SUM}(x,y) &= \sum_{x'<x} \sum_{y'<y} I(x',y') \\ \text{IMAGE\SUMSQ}(x,y) &= \sum_{x'<x} \sum_{y'<y} I(x',y')^2 \end{aligned}$$

This is useful for two reasons. First, for channels where the normalization constant c_W for each window W is computed using the mean and not the median, the computation becomes trivial. Letting (x_1, y_1) and (x_2, y_2) be the coordinates of the upper left and lower right corners of W respectively, it is easy to see that

$$c_W = \frac{\text{IMAGE\SUM}(x_2, y_2) - \text{IMAGE\SUM}(x_1, y_2) - \text{IMAGE\SUM}(x_2, y_1) + \text{IMAGE\SUM}(x_1, y_1)}{(x_2 - x_1 + 1)(y_2 - y_1 + 1)}$$

Second, once c_W is computed, in the feature computation step of equation 2.15, we have

$$\begin{aligned} \sum_{x',y'} W(x+x',y+y')^2 &= \sum_{x',y'} (I(\tilde{x}+x',\tilde{y}+y') - c_W)^2 \\ &= \sum_{x',y'} I(\tilde{x}+x',\tilde{y}+y')^2 - 2c_W \sum_{x',y'} I(\tilde{x}+x',\tilde{y}+y') + c_W^2 \\ &= \text{SUMSQ} - 2c_W \times \text{SUM} + c_W^2 \end{aligned}$$

where `SUMSQ` and `SUM` can be computed as above using 4 values each from `IMAGE\SUMSQ` and `IMAGE\SUM` respectively.

Local pixel feature extractors

Another way of extracting features is locally for each pixel (e.g. SIFT, RIFT, Spin features). The pixels can then be classified individually (e.g. as belonging to a specific texture or not). Alternatively, windows within the image can be classified using the bag-of-words approach with these local pixel features.

Warning: *These classes do not currently support multiple channels of input, only a single `IplImage`*

Pixel Feature Extractors The `svlPixelFeatureExtractor` is an abstract superclass for all local pixel descriptors. The simplest case is when the descriptors are extracted at each location individually from an `IplImage`; only the basic `getDescriptor()` function needs to be implemented. These descriptors can then be sampled densely over an image or used with a scale-invariant interest point detector (i.e. sampled only over a fixed set of locations over various scales). This is done using the `getDescriptors()` function which accepts a set of scales and interest points within each scale, and does all the appropriate resizing of the image.

The `svlRotInvariantExtractor` is a subclass of `svlPixelFeatureExtractor`. It defines a superclass for all feature descriptors that consist of radial histograms. The two variants, both implemented within `svlRotInvariantExtractor.h`, are the Spin (`svlSpinDescriptor`) and RIFT (`svlRIFTdescriptor`) features [15].

The other type of `svlPixelFeatureExtractor` is the `svlTextonExtractor`, which convolves the image with the 17-texton filter bank described in section 2.4.8 and returns the corresponding 17-dimensional feature vector at each position in the image. It wraps the `svlTextonFilterBank` class and puts it into the `svlPixelFeatureExtractor` framework.

Pixel dictionary The descriptors above are computed over individual pixels; the `svlPixelDictionary` class defines a superclass for all local pixel feature extractors that are accumulated over a region and vector-quantized to form a feature histogram. In other words, if you wish to use a bag-of-words approach to classification, you can define an `svlPixelFeatureExtractor` class for extracting the features (or use the ones described above), and then pass them into the `svlPixelDictionary` which is a subclass of `svlFeatureExtractor` and can be used within the sliding windows framework (see section 2.4.12).

Briefly, the `svlPixelDictionary` first takes a set of image window samples and builds a dictionary of features using vector-quantization (k-means clustering). In other words, it extracts a set of pixel descriptors from each window and clusters them into a pre-defined number of clusters. It can either use an interest point detector (only `svlHarrisLaplaceDetector` is currently supported) or densely sample the features within each window. Then given a new image window, it extracts the set of descriptors (again either based on the interest point detector or densely sampled), associates each one with a dictionary entry (closest cluster centroid) and returns a histogram of occurrences of the dictionary features within the window.

2.4.11 Filtering of fragment-based features

The `svlPatchFeatureSelector` is a subclass of `svlFeatureSelector` described in section 2.3.10. This filtering method iterates over the patch-based features, repeatedly selecting a feature with the highest classification F-score that hasn't already been selected and discarding all features that correspond to patches that are "too correlated" with it.

The correlation code is implemented in the `isCorrelatedWith()` method of the `svlPatchDefinition` class described in Section 2.4.10. This function returns `true` only if two patches are (1) extracted from the same type of image (e.g. intensity, depth, etc.), (2) overlap significantly in the regions they are considering within the 32×40 window, and (3) their templates are strongly correlated with each other. If all 3 conditions are satisfied, these two features would give very similar responses and there is no reason to keep both of them around in the dictionary.

See Appendix B for configuration options, and the `filterPatchDictionary` application described in Section 3.6.1 for more details about using the `svlPatchFeatureSelector` class.

2.4.12 Sliding window object detection

A common way to detect objects begins by building a binary classifier that takes as an input a small (say 32×40) rectangular image patch, and classifiers that image patch as either containing an object of interest or not. Given a full-size image, most object detection algorithms then rely on the *sliding window* technique, where this classifier is used to analyze every 32×40 sub-image of this larger image.

The `svlSlidingWindowDetector` encompasses this functionality by using an `svlFeatureExtractor` (cf. Section 2.4.10) and a trained `svlBinaryClassifier` (cf. Section 2.3.1) to analyze every patch of an image. Appendix B describes a variety of options. The `svlWindowDiscarder` class is used to quickly filter out "uninteresting" image patches that should not even be considered by the classifier. The current implementation simply discards patches that are close to uniform in the first channel (assumed to be the intensity channel).

The `svlMultiAspectAngleSWD` class implements a sliding window detector for multiple aspect ratios and angles. This is done using the `svlRotatableIpl` class which consists of an `IplImage` that can be rotated through all deformations without pixel loss using the `OpenCV` rotation commands.

2.4.13 Detection evaluation

Once objects in a set of scenes have been detected and the corresponding `svlObject2dSequence` created, the `svlObjectDetectionAnalyzer` class can be used to compare these detections with the groundtruth bounding boxes. The `svlObjectDetectionAnalyzer` class works with just a single object name, while the `svlMultObjectsDetectionAnalyzer` keeps track of multiple objects separately, and is able to perform analysis on each object individually. The class interfaces are fairly straight-forward, and they wrap around the `svlPRcurve` and `svlClassifierResponseList` described in section 2.3.11. See Appendix B for configuration options.

Understanding detection results may be facilitated by using the `svlLabelVisualizer` class, which renders color-coded ground truth and detection labels using `svlObject2dRenderable` objects (implemented in `svlObject2dExtensions.h`).

2.4.14 Camera Models

The `svlCameraIntrinsics` and `svlCameraExtrinsics` encapsulate standard camera models (including lens distortion) and help transform points between camera and world coordinate systems. Briefly, a point in the world coordinate system (x, y, z) projects into the (undistorted) camera plane via:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{R} & t \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (2.16)$$

where $\mathbf{K} \in \mathbb{R}^{3 \times 3}$ is the camera calibration matrix, \mathbf{R} is the rotation of the camera coordinate system from the world coordinate system and $t \in \mathbb{R}^3$ is the offset of the camera center (in world coordinates).

The `svlImageProjector` class encapsulates back projecting 3D points into an image plane. It is used by the `mrfDepthSmooth` application.

2.4.15 3d

- Points in 2D and 3D are encapsulated by the `svlPoint2d` and `svlPoint3d` data types. These may be deprecated in the future in favor of the `Eigen::Vector2d` and `Eigen::Vector3d` data types. The class `svlPointCloudData` holds a 3D point cloud (including point locations in 3D, surface normals, and point colors). This is the data structure used by the `pointCloudViewer` application (see Section 3.1.1).
- The `svlSpinImage` class can compute so-called spin-image features [13] from point cloud data. Refer to the `svlSpinImage.h` file for details.
- It is often convenient to project data from different sensors into a single image plane. Furthermore, since different sensors have different capabilities, it is desirable to obtain data at the highest possible resolution. The `svlSuperResolution` class implements a super-resolution MRF [4] for exactly this purpose. The `mrfDepthSmooth` application uses this class to smooth depth data projected from a point cloud into an image plane.

2.4.16 Segmentation

The **SVL** has a number of classes which support computer vision algorithms that work over segmented images (such as multi-class image segmentation). The `svlSegImage` class (derived from `svlSegImageBase`) holds an image and its over segmentation (superpixel description). The segmentation should be numbered from 0 to $K - 1$. The class also contains groundtruth labels (evidence) for training and evaluation. Groundtruth labels can be negative, indicating void (i.e., ignore during training and evaluation). The `svlSegImageBase` class is also used by `svlDepthSegImage` for an over segmented image with depth (3D) information (or

any real-valued map over the image pixels). Images are assumed to be 8-bit colour (3-channel). Matrices (segmentation and labels) are assumed to be 32-bit signed (integers).

Unlike `svlSegImage`, which holds an image and its single over-segmentation, the `svlMultiSeg` class holds multiple segmentations for the same image. Each segmentation is represented by the `svlImageSegmentation` class. The `svlImageSegmentation` class contains functionality for merging segments together, labeling segments (from pixel labels), finding all segments within a bounding-box, etc.

Features over segments can be computed using the `svlRegionFeatures` class. This class computes color, texture, geometric and location features over each region in an image. The feature vectors (for each region) include mean, standard deviation, skewness and kurtosis of features responses over the region. Options are controlled via the standard `svlOptions` interface and include boolean options “noColor”, “noIntensity”, “noTexture”, “noGeometry”, “noLocation” (all false by default), “includeStdev”, “includeSkewness”, and “includeKurtosis” (all true by default).

2.5 svlCuda

The `svlCuda` library contains GPU-optimized code using the NVIDIA Cuda library. The library provides an interface to GPU (hereafter “device”) memory, provides functions for performing mathematical operations on the device, and contains applications which use Cuda behind the scenes without exposing device-side memory to the user.

The major classes of the library and their functions are listed below:

CLASS	DESCRIPTION
<code>svlCudaMemHandler</code>	Manages device-side memory allocation, deallocation, transfer, and some esoteric brands of access.
<code>svlCudaMath</code>	Provides general mathematical functions, including addition, multiplication, min/max, and block-wise integration.
<code>svlCudaMatrix</code>	Provides general matrix-style functions.
<code>svlCudaConvolution</code>	Provides template matching functions.
<code>svlCudaFilter</code>	Provides image filtering functions.
<code>svlCudaDecisionTreeSet</code>	Provides a decision tree for device-side evaluation.
<code>svlCudaSWD</code>	A Cuda implementation of <code>svlSlidingWindowDetector</code> .

2.5.1 Data structures and allocation

The library provides device-side memory in three variants, defined in `svlCudaCommon.h`:

TYPE	DESCRIPTION
<code>svlCudaLine</code>	One-dimensional memory defined by a width.
<code>svlCudaPitch</code>	Two-dimensional, padded memory defined by a width, height, and pitch (the byte distance between successive rows on the device, hence the name).
<code>svlCudaPitch3d</code>	Three-dimensional, padded memory defined by a width, height, pitch, and depth.

The `svlCudaMemHandler` class provides members for allocating and freeing all of these forms, and transferring data back and forth to the GPU. The allocation and deallocation functions are:

```
class svlCudaMemHandler {
    ...
    static svlCudaLine *allocLine(int w, const char *name = NULL);
    static svlCudaLine *allocPitch(int w, int h, const char *name = NULL);
    static svlCudaLine *allocPitch3d(int w, int h, int z, const char *name = NULL);
};
```

```

static void freeLine(svlCudaLine* &ln);
static void freePitch(svlCudaPitch3d* &p);
static void freePitch3d(svlCudaPitch3d* &p);
...
}

```

As seen above, all device memory can optionally be given a name. This is for debugging device-side memory leaks. As the `svlCudaMemHandler` must be invoked to allocate and free all memory, it can optionally be told to track those allocations, enabling it to print out all active allocations on demand. If any data structure has been allocated with a name, that name will be displayed with it when it appears in this list. An example use is given below:

```

svlCudaMemHandler *cm;
svlCudaPitch *untracked = cm->allocPitch(640, 480);

svlCudaMemHandler::setTracking(true); // Enables tracking of allocations.

svlCudaPitch *tracked_unnamed = cm->allocPitch(640, 480);
svlCudaPitch *tracked_named = cm->allocPitch(640, 480, "named pitch");

...svlCudaMemHandler::printAllAllocations();

```

This produces the output:

```

All device allocations:
pitch @ 21a0000: 1,228,800 bytes
pitch @ 22d0000: 1,228,800 bytes "named pitch"
Total device allocation: 2 structures for 2,457,600 bytes

```

The first pitch is untracked; the second is tracked but has no display name in the allocations list; and the final is tracked and named.

Tracking of course introduces overhead, so it should not be used when benchmarking code. Also, in the above example, if the pitch *untracked* is freed while tracking is still enabled, an error will be reported as the tracker did not have this pitch in its register. Tracking should be turned off before any untracked memory is to be freed.

2.5.2 Memory transfer

Memory is transferred to and from the GPU via `svlCudaMemHandler` functions of the name `{line,pitch,pitch3d}{To,From}GPU`. All pointers are treated as `float*` and so address four-byte data. Transferring `int` and `uchar` data to the GPU requires pointer casts to `float*s`, but this type punning does not affect the data transferred.

The transfer functions accept optional size parameters that will select a sub portion of the memory to transfer. For example, if a line and a pitch have been allocated with space for 1000 floats and 640x480 floats, respectively, then the calls:

```

// Transfer memory from src pointer to device.
svlCudaMemHandler::lineToGPU(line, src, 408); // width of transfer
svlCudaMemHandler::pitchToGPU(pitch, src, 320, 240); // width and height of transfer
// Transfer memory from device to dest pointer.
svlCudaMemHandler::lineFromGPU(dest, line, 408);
svlCudaMemHandler::pitchFromGPU(dest, pitch, 320, 240);

```

will only transfer the first 408 bytes of the line, and only the upper quadrant of the pitch (2D memory is stored in row-major order).

Regardless of how memory is sub-selected on the device, the result is always copied out linearly into the destination pointer (and also from the source pointer). Thus, in the above example, the *dest* pointer would need 320x240x4 bytes to store the memory returned from the `pitchFromGPU` call.

The functions `Ipl{To,From}GPU` are also provided to transfer `IplImage` structures to and from pitches.

2.5.3 Line integrals

`svlCudaMath` contains functions for computing cumulative sums of lines (also known as the “scan” algorithm). In these functions, integration is not performed over the total size of the data structure but over a block size, which is tiled over a desired range of the data structure. This block size must be a power of 2, and is identified as the argument `dim` in all integration functions. The function has a maximum `dim` value of 1024, dictated by the maximum number of threads (512 on most GPUs, and `dim` can be at most two times this number).

The function to piece-wise integrate a line is:

```
void integralLine(const svlCudaLine *n, svlCudaLine *integ, int w = -1,
int dim = 128, bool do_cumsum = false, int stride = -1);
```

The width argument `w` determines how many spaces to integrate over. If it is unspecified, the integration operation is applied over the entire width of `n`. The function of the additional arguments is described below.

With its default arguments, a pixel of `integ` will be computed as:

$$integ[i] = \sum_{j=\lfloor i/dim \rfloor}^{i-1} n[j]$$

I.e., each cumulative sum only proceeds back to the beginning of the current, `dim`-sized block.

Turning on `do_cumsum` makes the function perform like Matlab’s `cumsum` operation, which will sum up one more value in each pixel:

$$integ[i] = \sum_{j=\lfloor i/dim \rfloor}^i n[j]$$

The argument `stride` specifies the distance between blocks that will be integrated. If, for example, `dim = 128` and `stride = 200`, then the values from 0..128 will be integrated, those from 128..200 left unchanged, those from 200..328 integrated, and so on. Concretely,

$$integ[i] = \begin{cases} \sum_{j=\lfloor i/stride \rfloor}^{\lfloor i/stride \rfloor + dim - 1} n[j], & \text{if } i < \lfloor i/stride \rfloor + dim \\ integ[i], & \text{o.w., retains value} \end{cases}$$

Setting `stride` to be less than `dim` results in undefined behavior since each `dim`-sized integration is computed by a separate block of threads, which, according to the Cuda manual, can be executed in any order.

2.5.4 Integral images

The integral images function performs block-wise operations in both the row and column dimensions. The block-wise constraint here actually helps militate against precision errors as integrating an entire image can introduce significant numerical errors in the GPU’s native single-precision floating point.

The block integration function is given below:

```
void integralBlock(svlCudaPitch *pin, svlCudaPitch *integ, svlCudaPitch *tempint,
svlCudaPitch *integ2 = NULL, svlCudaPitch *tempint2 = NULL,
int img_w = -1, int img_h = -1, int dim = 128);
```

Only the integral image is required to be computed, with the computation of the integral-squared image being optional. A temporary pitch is required to perform each operation as the integration is implemented as a two-step (row-wise then column-wise) procedure. When computing both integral and integral-squared images, a temporary pitch is required for both (a single temporary pitch cannot simply be re-used) because

the inner operations of both the integral and integral-squared procedures are conflated so as to reduce the memory transfer of reading in the input pitch.

While the intuition behind block-wise integration is simple, the precise formula for its output is complex. Traditionally integral images are one row and one column larger than their input images as the formula for a pixel is $integ(r, c) = \sum_{i=0}^{r-1} \sum_{j=0}^{c-1} img(i, j)$. This we wanted to preserve with the Cuda version, but under block-wise integration this becomes confusing: if the block size is 32, each sub-block requires a 33-by-33 output. But, with the above formula, the first row and column of the integral image will contain all zeros; so likewise will the first row and column of each sub-block integration. Thus, in the block-wise integral images, each sub-block writes its last, “ $dim + 1$ ”st row and column over the first row and column of the next sub-block, which would have been zero otherwise. The formula for the block integral images is then:

$$integ(r, c) = \begin{cases} 0, & \text{if } r = 0 \vee c = 0 \\ \sum_{i=\lfloor r/dim \rfloor}^{r-1} \sum_{j=\lfloor c/dim \rfloor}^{c-1} img(i, j), & \text{if } mod(r, dim) \neq 0 \wedge mod(c, dim) \neq 0 \\ \sum_{i=\lfloor r/dim \rfloor - dim}^{r-1} \sum_{j=\lfloor c/dim \rfloor}^{c-1} img(i, j), & \text{if } mod(r, dim) = 0 \wedge mod(c, dim) \neq 0 \\ \sum_{i=\lfloor r/dim \rfloor}^{r-1} \sum_{j=\lfloor c/dim \rfloor - dim}^{c-1} img(i, j), & \text{if } mod(r, dim) \neq 0 \wedge mod(c, dim) = 0 \\ \sum_{i=\lfloor r/dim \rfloor - dim}^{r-1} \sum_{j=\lfloor c/dim \rfloor - dim}^{c-1} img(i, j), & \text{if } mod(r, dim) = 0 \wedge mod(c, dim) = 0 \end{cases}$$

When computing an integral image, one must first set the output pitches to 0 using the `svlCudaMemHandler::pitchSet(·, 0)` command. This is because none of the sub-blocks of the integration write their first rows and columns, which contain all zeros. Thus, if the first row and column of the integral image are not set to zero beforehand, they will contain uninitialized values even after block integration.

2.5.5 Convolution functions

Convolution functions are provided in the `svlCudaConvolution` class. Despite their name, these functions actually perform template matching rather than convolution, and so do not flip the kernel before computation.

The main convolution function is `conv2d`, which takes as input a pitch or a 3d pitch, and a template in the form of a `float*` with a width and a height and stored in row-major order, or an `IplImage*`:

```
void conv2d(svlCudaPitch *p_in, const float *patch, int pw, int ph,
           svlCudaPitch *p_out, unsigned int options = SVL_CONV_SAME,
           int img_w = -1, int img_h = -1);
void conv2d(svlCudaPitch *p_in, const IplImage *patch,
           svlCudaPitch *p_out, unsigned int options = SVL_CONV_SAME,
           int img_w = -1, int img_h = -1);
```

This performs the Matlab-style `filter2` operation, or the OpenCV `cvMatchTemplate` with the `CV_TM_CCORR` argument. Concretely:

$$p_out(r, c) = \sum_{i=0}^{patch_h-1} \sum_{j=0}^{patch_w-1} p_in(r+i, c+j) \cdot patch(i, j)$$

The function supports images of any size but only templates of size less than or equal to 16 in both dimensions. If a kernel up to size 32-by-32 is given, this is implemented as running up to four 16-by-16 convolutions behind the scenes, and summing the results.

By default, the function performs Matlab ‘same’-style convolution, and so expects a return pitch p_out at least the same size as the input pitch, p_in . If the `option` argument `SVL_CONV_FULL` is used instead, however, it will return the Matlab ‘full’-style convolution result, and so will require an output pitch at least of size $(img_w + patch_w - 1, img_h + patch_h - 1)$.

Another convolution function provided is:

```
void ccoeffNormed(svlCudaPitch *p_in, const float *patch, int pw, int ph,
                 svlCudaPitch *p_out, int img_w = -1, int img_h = -1);
void ccoeffNormed(svlCudaPitch *p_in, const IplImage *patch,
                 svlCudaPitch *p_out, int img_w = -1, int img_h = -1);
```

This computes the normalized correlation coefficient in the manner of OpenCV's `cvMatchTemplate` with the `CV_TM_CCORR_NORMED` option, hence its name. Exactly,

$$p_{out}(r, c) = \frac{\sum_{i=0, j=0}^{h-1, w-1} p_{in}'(r + i, c + j) \cdot patch'(i, j)}{\sqrt{(\sum_{i=0, j=0}^{h-1, w-1} p_{in}'(r + i, c + j)^2)(\sum_{i=0, j=0}^{h-1, w-1} patch'(i, j)^2)}}$$

where

$$patch'(r, c) = patch(r, c) - \frac{1}{w \cdot h} \sum_{i=0, j=0}^{h-1, w-1} patch(i, j)$$

$$p_{in}'(r + i, c + j) = p_{in}(r + i, c + j) - \frac{1}{w \cdot h} \sum_{i'=0, j'=0}^{h-1, w-1} p_{in}(r + i', c + j')$$

These convolution functions pad the patch before transferring it to the GPU, which incurs host-to-device memory transfer overhead with each call. If one has a large number of templates, however, these can be pre-padded and transferred to the GPU using the `svlFinalPatchDictionary` class in the `svlCuda` library. Then there are versions of the convolution functions that can be called with `CachedPatches`, negating memory transfer overhead.

Larger convolutions

The convolution described above is an optimized version which uses device `constant` memory, but was originally limited to convolving with a kernel only up to 16-by-16 in size. Modifications were added to make the convolution work up to size 32-by-32, but only by tiling up to four individual 16-by-16 convolutions behind the scenes.

If even larger convolutions are desired, because of the limitations of `constant` memory, an entirely different implementation had to be written, and one that convolves `svlCudaPatches` with each other. The signature for this method is:

```
static void cudaPatchConv(const svlCudaPatch *pin, const svlCudaPatch *patch,
    svlCudaPatch *pout, const svlCudaPatch *orig_patch = NULL,
    int img_w = -1, int img_h = -1, int out_w = -1, int out_h = -1);
```

(if the various image size arguments are not specified, they are pulled from the patches' sizes directly) The `pin`, `pout`, and `patch` arguments are obviously for the input, output, and patch pitches. The `orig_patch` member, however, requires special discussion.

To optimize this new convolution for speed, it was necessary to restrict the convolution patches (the `patch` member) to have a width divisible by 16. This does not apply to the *height* of the `patch` member, nor are the widths or heights of the other input pitches restricted in any way. But this requires the `patch` member to be properly padded before calling the function. But, since this function performs "valid"-style convolution by default, extra pixels to the side of the image will not be computed (e.g., if the original patch was 100 pixels wide, and was padded to 112, the final 12 columns of the convolution will not be computed). To fix this, one can supply a pointer to the original, unpadded pitch as `orig_patch`, with the proper width, which makes the function properly compute all columns that would have been lost to padding. If this pointer is not specified, `patch`'s width is taken as the original, unpadded width.

2.5.6 Decision tree set

The `svlCudaDecisionTreeSet` is a subclass of `svlVision`'s `svlDecisionTreeSet`. Thus it inherits all of the same I/O functionality for reading from XML files, and a Cuda DTS can be instantiated from the same file that instantiates a regular DTS. What the Cuda class adds is functionality for device-side evaluation of the decision trees on an array of features.

The function for evaluating the decision trees is deceptively involved. The signature is:

```
void evaluate(svlCudaLine *features, svlCudaLine *invalid, int feature_len,
             int feature_stride, int n_fvs, int execs_per_block = 100) const;
```

The parameter *features* is obviously the vector of feature vectors. Each vector has a length of *feature_len*, while the spacing between individual feature vectors is *feature_stride*. These last two quantities need not be equal, and, in fact *feature_stride* can be greater than or equal to *feature_len* (meaning there is padded space at the end of every feature vectors). *n_fvs* refers to the number of feature vectors, and *execs_per_block* is a parameter which can be tuned for optimal evaluation time. *invalid* is an output vector, which should have at least *n_fvs* space, which will contain a non-zero entry if that feature vector contained invalid entries (NaNs or infinities).

There are several steps to evaluating the decision trees when this function is called. First, the decision trees are evaluated individually, with their results stored *in-place* over the features, consuming the features values (so the values must be copied beforehand if one does not want them lost). Because of this, there has to be enough space between each feature vector to accommodate not only the number of features but the number of decision trees. If there are only 45 features, but 200 decision trees, say, then *feature_stride* must be at least 200.

Second, after the individual decision trees are evaluated, they have to be summed to generate the ensemble response. This is accomplished by using calls to the line integral functions above. However, because these all sum over distances that are powers of two, there has to be enough padding in the decision tree outputs to accommodate this. If there are 45 features and 200 decision trees, then the feature stride cannot be 200 but has to be 256, the nearest power of 2 above 200, and these extra spaces also need to be initialized to zeros so that the decision tree sums are correct. After this, the summed decision tree response is placed in the first entry of every feature vector.

Because of the intricacy of this procedure, the class `svlCudaFeatureLine` is provided to hide most of this functionality. Its initialization method is:

```
svlCudaFeatureLine(int n_features, int nTrees, bool need_objects = true);
```

The crucial arguments are the number of features in each vector and the number of trees in the decision tree. If these numbers are 45 and 200, as in the above example, the `svlCudaFeatureLine` will correctly calculate a feature stride of 256, and then calling the `void alloc(int n)` method will allocate space for that many feature vectors as the class member `svlCudaLine *features`.

Evaluating a decision tree set on the features is also much easier, accomplishable with:

```
void evaluate(const svlCudaDecisionTreeSet *tree);
```

Assuming the given tree has the same number of nodes as the `svlCudaFeatureLine` was initialized with, this call will perform all of the steps enumerated above for evaluation. In the end, the compacted results will appear in the class member `svlCudaLine *results`, which will have a length of *n*, as above. Care will still have to be taken to make sure that the proper feature stride is used when writing into the memory allocated by `svlCudaFeatureLine`, but allocation and evaluation are much simpler with this class. Examples of its use are given in the `cudaVerify` application.

2.5.7 Sliding window detector

The Cuda sliding window detector ports all of the functionality of the `svlSlidingWindowDetector` in `svlVision`. However, both classes take slightly different inputs. They are both initialized by the function:

```
bool load(const char *extractorFilename, const char *modelFilename = NULL);
```

The first of these, the “extractor,” refers to the patch dictionary. Both classes require a patch dictionary, only the `svlVision` SWD reads the dictionary with `svlPathDictionary`, while the `svlCuda` SWD reads the dictionary with `svlCudaPatchDictionary`. The former is a subclass of the latter, though, and only adds additional functionality for device-side evaluation of the patch responses.

The second of these, the “model,” refers to the classifier that makes binary decisions off of the computed features. Here the two classes take different inputs. The `svlVision` SWD uses an OpenCV boosted decision tree as its input, while the `svlCuda` SWD uses an `svlCudaDecisionTreeSet`, which is a Cuda-capable subclass of `svlDecisionTreeSet`, which is an ensemble of boosted decision trees. Examples on how to instantiate a `svlCudaDecisionTreeSet` are given in the `cudaVerify` application, which tests the `svlCudaDecisionTreeSet`.

2.5.8 Verification application

Because of the great intricacy of writing GPU code, a `cudaVerify` application is provided in the `svl/apps/cuda` directory. This subjects a great number of the library functions to random numerical tests to ensure their correctness. If interested in using the Cuda portion of the library, one should first compile and run this function to ensure that all the verified functions are working properly.

Ideally all library functions should be tested in the verification application, but, as the library is still under development, only a fraction of them are. All of the functions listed above, however, have been verified. Also, each library header file has its verified functions listed first and marked clearly, with older and un-verified functions listed later. Just because a function has not been verified does not mean it contains bugs. All functions were tested before being placed in the library, but the specific manner of testing in the `cudaVerify` application was a late addition, and all of the old tests have not yet been ported over to this new method.

The names of the tests can be listed by running `./cudaVerify -names`, and all tests can be run with the command `./cudaVerify -all`, or they can be switched on individually with `./cudaVerify -on SWD`. A list of the “named” tests and the functions they evaluate are given below:

NAME	DESCRIPTION
<code>alloc</code>	Tests allocation and transfer of memory between host and device.
<code>math</code>	Tests general math operations, including basic math, adding in quadrature, and image region means.
<code>minmax</code>	Tests the image region minimum and maximum operations.
<code>integral</code>	Tests in the integration functions over lines and pitches.
<code>lines</code>	Tests utility function with <code>svlCudaLines</code> , including complex memory access patterns.
<code>conv</code>	Tests convolutions with <code>IplImages</code> and <code>float*s</code> up to size 16-by-16.
<code>conv32</code>	Tests convolutions with <code>IplImages</code> and <code>float*s</code> up to size 32-by-32.
<code>convsuper</code>	Tests convolutions with <code>svlCudaPitches</code> up to size 128-by-128.
<code>filter</code>	Tests functions in the <code>svlCudaFilter</code> class, or currently just the Sobel soft edge filter.
<code>dtree</code>	Tests the Cuda decision tree, including reading in a test ensemble and evaluating it on random features.
<code>SWD</code>	Tests the Cuda sliding window detector by computing features on random images with a fixed dictionary.

2.5.9 Benchmarking application

As one of the principle reasons for using Cuda is to speed-up code, the `cudaBenchmark` function provides a small set of battery tests were `svlCuda` functions are pit against OpenCV implementations of the same. Currently only integral images and convolution are compared, but the results are given in easy-to-read table formats:

```

*** Benchmarking CCORR convolution with 640-by-480 image ***

OpenCV [ms]:
Height\width      4      8      12      16
  4  38.200  38.799  40.599  45.599
  8  38.599  36.799  38.400  35.599
 12  46.599  42.200  42.200  42.200
 16  56.599  38.000  37.200  38.400

Cuda 'same' [ms (speedup)]:
Height\width      4      8      12      16
  4  0.335 (113.9x)  0.415 ( 93.3x)  0.577 ( 70.3x)  0.568 ( 80.2x)
  8  0.471 ( 81.7x)  0.626 ( 58.7x)  0.942 ( 40.7x)  0.936 ( 37.9x)
 12  0.611 ( 76.2x)  0.840 ( 50.2x)  1.318 ( 32.0x)  1.308 ( 32.2x)
 16  0.755 ( 74.9x)  1.052 ( 36.0x)  1.676 ( 22.1x)  1.676 ( 22.9x)
Speedup min, mean, max: 22.1x — 57.7x — 113.9x

Cuda 'full' [ms (speedup)]:
Height\width      4      8      12      16
  4  0.345 (110.4x)  0.425 ( 91.0x)  0.594 ( 68.2x)  0.586 ( 77.7x)
  8  0.474 ( 81.2x)  0.638 ( 57.6x)  0.960 ( 39.9x)  0.958 ( 37.1x)
 12  0.616 ( 75.6x)  0.854 ( 49.3x)  1.340 ( 31.4x)  1.351 ( 31.2x)
 16  0.759 ( 74.4x)  1.078 ( 35.2x)  1.732 ( 21.4x)  1.729 ( 22.2x)
Speedup min, mean, max: 21.4x — 56.5x — 110.4x

```

As with `cudaVerify`, all benchmarking trials can be run with `./cudaBenchmark -all`, or they can be run individually *à la* `./cudaBenchmark -on conv`, which produced the above output.

2.5.10 Custom application

Compiling an application that uses Cuda strictly through the `svlCuda` interface involves merely including “`svlCuda.h`”. But, as interacting with all device-side memory requires a kernel function, when building a custom application, one may need to write their own `cu` files. As there are several intricacies to getting such projects to build correctly, they are detailed here, as well as presented in example in the `cudaCustom` application.

There are three files in the application: `cudaCustom.{cpp,h,cu}`. The `cpp` is compiled as the executable, while the `cu` compiles the custom kernel functions, and the `h` is a linker between. To call kernel functions from plain C++ code, one must have a C++ wrapper function for them, and this prototype must be `extern`’ed so as to be visible from the `cpp`. However, one cannot put the `extern` definitions in the `cpp` file directly. They must be placed in and included from a separate `h` file. This is done with the `cudaCustom.h` file.

To use the `svlCuda` library intermixed with custom kernel functions, one may wish to use the built-in `svlCuda` data structures. To do this, one cannot include “`svlCuda.h`” from the kernel functions. First, because this will include many classes in the library, and also inevitably link to `svlVision`, `svlML`, and `svlBase`, which will require adding the paths for all of these to the NVIDIA compiler. Second, even if one adds all of these paths to the compiler to get it to compile, it will report all warnings in the compilation of the `SVL`, and, as the NVIDIA compiler is more fastidious about reporting warnings than `gcc`, this will result in a great number of them, greatly slowing computation. While linking in all of these extra files is possible, still, because of the extra warnings and delays in compiling, it should only be done for single files at a time when it is strictly necessary.

The compromise is to include only “`svlCudaCommon.h`” from the kernel functions, as this file contains only the data structure definitions in the library, as well as the `extern`’ed definitions of all of the other kernel functions, allowing one’s custom `cu` files to use library functions as well. If one still wants to use high-level library functions defined in the classes (such as from “`svlCudaMath.h`”), one can include them from the `cu` file as one takes care to add the proper paths to the NVIDIA compiler. In the `cudaCustom` application, the `h` file is used to include “`svlCudaCommon.h`” while the `cu` file include the `h`.

Chapter 3

SVL Applications and Scripts

3.1 GUI Applications

3.1.1 Point Cloud Viewer

The `pointCloudViewer` application allows you to visualize 3D point clouds. You can navigate around the point cloud using the arrow keys (for moving in the xz -plane) and keys `a` and `z` (for up and down). You can also zoom in and out and rotate by dragging the mouse (holding down the left and right buttons, respectively). Points can be viewed in their true color (back-projection), or colored by depth, height, or user-assigned weight. You can also load a triangulated mesh to view textured 3D surfaces over the point cloud.

The default format for a point cloud (for `File|Open...`) is a 10-element (whitespace delimited) vector per point $(x, y, z, n_x, n_y, n_z, r, g, b, w)$ where (x, y, z) is the location of the point in space, (n_x, n_y, n_z) is the (surface) normal associated with the point, (r, g, b) is the color of the point (normalized between 0 and 1), and w is a weight associated with the point. Unused fields can be set to zero. In addition (x, y, z) points can be imported using `File|Import...`. Here 3-element vectors will be read (you can change the data width in the `Options` menu). When you import points, the points are added to the existing point cloud. Use the `Options` menu to change the color of the imported points.

Tip: *If you get lost while exploring your point cloud, hit the `x` key to take you home.*

The `pointCloudViewer` will also render textured 3D meshes given a (separate) file of (whitespace delimited) triangle indices. The triangle indices are zero-based and reference vectors in the point cloud file. Thus, the triangle $(0, 1, 2)$ refers to the first three points in the point cloud.

3.1.2 Image Sequence Labeler

The `imageSequenceLabeler` application allows you to label (and view) the location of objects in images. Usually objects are labeled by placing a bounding box around the object and annotating the box with the object's name. The `imageSequenceLabeler` will produce an XML file (see Section 2.4.4) which can be read by the various object detection applications to train and evaluate object detector models.

Tip: *Holding down the `Shift` key when resizing an object to make the bounding box square.*

Tip: *Press `'-` or `'+'` to decrease/increase the size of the active bounding box.*

Tip: *Press the `Insert` key to copy all objects from the previous image to a current empty image.*

Using the `imageSequenceLabeler` application you can also view the output from object detection (see 3.6.1 below).

3.1.3 Region Labeler

Applications such as multi-class image labeling—the assignment of classes to every pixel in an image—require ground truth data for training and evaluation. The `regionLabeler` tool lets you generate this ground truth data by labeling every pixel in an image with a class label, e.g., sky, building, road. The tool is configured by loading a set of region labels from an XML file, e.g.,

```
<regionDefinitions version="1">
  <region id="0" name="background" color="0_0_0" />
  <region id="1" name="foreground" color="255_0_0" />
</regionDefinitions>
```

The region labels can be loaded using the `File|Region Definitions...` menu option, or as a command line argument, e.g.,

```
`${CODEBASE}/bin/regionLabeler -d `${CODEBASE}svl/scripts/msrcRegions.xml &
```

After loading an image, you can then use paint-like tools to color regions in the image. The figures below show three different views (image, overlay, and segments) from the tool.



Figure 3.1: `regionLabeler` image, overlay and segment views.

The underlying labels are stored in text format as a matrix, the same size as the image, where each element is an integer indicating the pixel class. A negative value indicates unknown or void.

Tip: A useful trick for labeling images is to use the paint brush to trace the outline of a contiguous region and then holding the `CTRL` key, click inside the region. This will flood-fill the region with the current class label.

Use `tab` to iterate through the labels, `d` to iterate through the drawing modes, and `v` to iterate through the viewing modes. Holding down `shift` iterates in the other direction.

3.2 Machine Learning Applications

3.2.1 Classifiers

The machine learning applications `trainClassifier` and `evalClassifier` provide file-level interfaces to the `SVL` multi-class logistic and boosted decision tree classifiers. The `trainClassifier` application lets you train a from labeled training instances. The program takes as input a file containing all a set of training feature vectors (one per row) and a file containing the corresponding target labels (integers starting from 0). The boosted decision tree classifier learns one-versus-all binary classifiers for each class. Negative target labels (e.g., -1) are considered *unknown* and will not be used for training (or evaluation).

Novel data instances can be classified using the `evalClassifier` application. If ground truth labels are available, the code can produce a confusion matrix and overall accuracy for the learned model. Negative ground truth entries indicate *unknown* and are ignored during evaluation.

Example input data can be found in the `tests/input` directory. The following demonstrates how to train and test a classifier by generating 5 random random partitions of the data for training, and testing on the hold out set.

```

${CODEBASE}/svl/scripts/generateCrossValidationFolds.pl -f 5 \
  ${CODEBASE}/tests/input/iris.data.txt ${CODEBASE}/tests/input/iris.labels.txt

foreach i ('seq 0 4')
  ${CODEBASE}/bin/trainClassifier -c LOGISTIC -o model.${i} \
    iris.data.txt.${i}.train iris.labels.txt.${i}.train > /dev/null
  ${CODEBASE}/bin/evalClassifier -l iris.labels.txt.${i}.test model.${i} \
    iris.data.txt.${i}.test | grep accuracy
end
```

3.2.2 Precision-recall curves

The `analyzePRcurve` is a quick application for finding the area and the highest F-score of a precision-recall curve stored in a file. See Section 2.3.11 for more information about working with `svlPRcurves`.

3.3 Matlab (mex) Applications

The **STAIR Vision Library** provides some Matlab applications that wrap the core machine learning and probabilistic graphical models libraries. These applications are not compiled by default: to build them under Linux you should add the line `BUILD_MEX_APPS = 1` to your `make.local` file.

Warning: *mex applications are not yet supported under Windows.*

Tip: *To add SVL applications to Matlab's search path run the `addSVLPaths.m` script.*

3.3.1 Probabilistic Graphical Model Inference

`mexFactorGraphInference` provides a Matlab interface for running general Graphical Model inference algorithms on factor graphs. The following provides an example using the Rosetta protein design dataset [27].

```

% add STAIR Vision Library to Matlab path
run('svl/scripts/addSVLPaths');

% load Rosetta protien design problem definition
load('1bx7.dee.mat');

nVariables = length(Ei);
varCards = []; for i = 1:nVariables; varCards(i) = length(Ei{i}); end;
maxCard = max(varCards);

[i, j] = find(adjMatrix);
pairwiseCliques = [i, j];
pairwiseCliques = pairwiseCliques(i < j, :);
nPairwiseCliques = length(pairwiseCliques);

% build factors
factors = repmat(struct('vars', [], 'cards', [], 'data', []), ...
  nVariables + nPairwiseCliques, 1);

for i = 1:nVariables,
```

```

factors(i).vars = i - 1;
factors(i).cards = varCards(i);
factors(i).data = exp(-Ei{i});
end;

for i = 1:nPairwiseCliques,
    phi = Eij.cells{Eij.indMat(pairwiseCliques(i, 1), pairwiseCliques(i, 2))};
    factors(i + nVariables).vars = pairwiseCliques(i, :) - 1;
    factors(i + nVariables).cards = varCards(pairwiseCliques(i, :));
    factors(i + nVariables).data = exp(-phi(:));
end;

% run inference
options = struct('infAlgorithm', 'ASYNCMAXPRODDIV', 'maxIterations', 1000, 'verbose', 1);
output = mexFactorGraphInference(factors, options);
mapAssignment = [output(1:nVariables).value] + 1

```

3.3.2 General CRF Learning and Inference

The following code builds a templated-MRF model over binary variables with (one) singleton and (two different) pairwise terms. There are no features in this model.

```

% construct CRF model
model.weights = log([1.0 0.5 0.9 0.1 0.2 0.8]);

model.templates(1).cards = [2];
model.templates(1).entries(1).wi = [0];
model.templates(1).entries(2).wi = [1];
[model.templates(1).entries(1:2).xi] = deal(-1);

model.templates(2).cards = [2 2];
model.templates(2).entries(1).wi = [2];
model.templates(2).entries(2).wi = [3];
model.templates(2).entries(3).wi = [3];
model.templates(2).entries(4).wi = [2];
[model.templates(2).entries(1:4).xi] = deal(-1);

model.templates(3).cards = [2 2];
model.templates(3).entries(1).wi = [4];
model.templates(3).entries(2).wi = [5];
model.templates(3).entries(3).wi = [5];
model.templates(3).entries(4).wi = [4];
[model.templates(3).entries(1:4).xi] = deal(-1);

```

The following code builds a large grid-structured MRF instance for the above model.

```

% construct CRF instance
nRows = 160;
nCols = 120;
nVars = nRows * nCols;

instance.cards = 2 * ones(nVars, 1);
instance.values = [];
for i = 1:nVars,
    instance.cliques(i).Cm = [i - 1];
    instance.cliques(i).Tm = 0;
end;

k = 1;
for i = 1:nRows,
    for j = 1:nCols,
        v = nCols * (i - 1) + j;
        if (i < nRows),
            instance.cliques(nVars + k).Cm = [v - 1, v + nCols - 1];

```

```

        instance.cliques(nVars + k).Tm = 1 + (rand(1) > 0.9);
        k = k + 1;
    end;
    if (j < nCols),
        instance.cliques(nVars + k).Cm = [v - 1, v];
        instance.cliques(nVars + k).Tm = 1 + (rand(1) > 0.9);
        k = k + 1;
    end;
end;
end;

% no features
[instance.cliques(:).Xm] = deal([]);

```

The following code runs inference on the above CRF instance.

```

% run inference
tic;
output = mexCRFInfer(model, instance, struct('maxIterations', 1000));
toc;
output.marginal

```

3.3.3 Classifier Training and Evaluation

The `mexTrainClassifier` and `mexEvalClassifier` Matlab applications let you train and evaluate multi-class logistic or boosted decision tree classifiers. The applications takes as input Matlab 2D matrix containing all a set of training (or test) feature vectors (one per row) and a matrix containing the corresponding target labels (integers starting from 0). The weights vector weights the data instances, and has to be of the same length as the number of rows in data. In the example below, if we had a weights vector, it would be of length 150. Regularization of the weights can be controlled through `options.lambda`.

The code below demonstrates typical usage for training and evaluating classifiers. It should be run from the base **SVL** directory.

```

run('svl/scripts/addSVLPaths');

labels = dlmread('tests/input/iris.labels.txt');
data = dlmread('tests/input/iris.data.txt');

options = struct('maxIterations', 1000, 'nClasses', 3, 'lambda', 1.0e-6);
parameters = mexTrainClassifier(data, labels, [], options);

output = mexEvalClassifier(parameters, data);
[confidence, predicted] = max(output, [], 2);
predicted = predicted - 1;
accuracy = sum(predicted == labels) / length(labels)

```

3.4 Matlab (.m) Scripts

A few basic Matlab scripts are included in the `svl/scripts` directory. These include scripts for plotting precision-recall curves (`plotPR.m`), computing confusion matrices (`confusionMatrix.m`), and combining multiple images into a single big image (`combineImages.m`).

Tip: Run the `addSVLPaths.m` script to add **SVL** .m files to Matlab's search path.

Other scripts include the XML utilities `parseXML.m` and `writeXML.m`. These can be tested by asking them to parse and regurgitate an existing XML file:

```

writeXML('tests_test.xml', parseXML('../tests/tests.xml'), 'Pretty');

```

The files `writeipl.m` and `readipl.h` are also for writing or reading `IplImages` that have been output in binary format using the `svlVision` utilities `svlBinaryWriteIpl` and `svlBinaryReadIpl` (located in `svlVisionUtils.h`). These functions make it easy to pass `IplImages` in between Matlab and library applications.

3.4.1 Library Examples

While the library contains many applications and example uses of its classes, many of these are high-level, and so it is hard to separate out and observe the functioning of a small part of the library. Thus several example projects with walkthrough M files are provided in the `projects/examples` directory, which instantiate and run just one class at a time. This makes it easy to see how to initialize use the class, and also provides graphical output with which to verify the class's functionality.

There are currently three examples implemented: one for testing the `svlDecisionTree` (as `exampleLogistic.m`), one for the multi-class logistic classifier in `svlLogistic` (as `exampleLogistic.m`), and one for the boosting implementation in `svlNativeBoostedClassifier` (as `exampleBoosting.m`). Each of these comes with a C++ application that must be compiled first, for which a Makefile is provided in the `projects/examples` directory.

The example M files always contain five “cell” sections:

1. Initializing the path and some system variables.
2. Creating and displaying the data the test will be run on.
3. Executing a `system` command to run the test application.
4. Reading in the results of the functions and displaying them.
5. Deleting any data files created by the test.

For running the M files, one should set their Matlab working directory to SVL's `bin` directory, and add the `./projects/examples` path. Since the M files run outside applications, they contain instructions for setting whether one's OS is Windows or Linux, so as to execute the proper system calls. On Linux, the system application calls may not execute, in which case one can copy the command `system` is trying to execute and run it manually, producing the function output the rest of the M file requires.

3.5 Test Applications

A number of test applications (in `svl/apps/test`) are provided for testing different functionality of the library. The source code for these tests is a useful resource for learning how to use the code. These applications, in addition to some standard applications, are the basis of the regression tests in the `tests` directory. The regression tests compare the output from various applications with pre-computed output. A difference indicates a failed test.

Warning: *Some of the regression tests are sensitive to numerical precision and will fail on some systems (depending on architecture and compiler version). This known issue will be fixed in later releases of the code.*

To run the regression tests simply execute the command `runTests.pl test.xml` from the `tests` directory. You will need to have Perl (and either `XML::Parser` or `XML::SAX`) installed to run the tests. Executing `runTests.pl` without command line arguments will list some test options.

3.6 Computer Vision Applications

3.6.1 Standard Object Detection pipeline

Object detection in the **STAIR Vision Library** is implemented based on the work of Torralba et al. [23], briefly described in Section 2.4.10. The separate applications used in the pipeline are described below, followed by scripts which combine some of them in a useful way.

buildTrainingImageDataset

Used to build the training dataset for the object classifier. Given a set of labeled scenes and the name of the object of interest, cropped positive and negative training examples are extracted. Most of the functionality is encompassed in the `svlTrainingDatasetBuilder` class of the vision library; see Section 2.4.7 and Appendix B for details. The `-parallel` flag is used for distributing jobs on the cluster.

buildPatchDictionary

Given the cropped positive images of the object of interest, often obtained from the `buildTrainingImageDataset` application, builds a dictionary of features as discussed in Section 2.4.10 (**building the dictionary** paragraph).

buildWindowFeatureCache

Given a set of training images and the feature dictionary, computes the feature vector for each training example as described in Section 2.4.10. The output is either a set of text files, one per input image, in the output directory, or, if the `-binary` flag is set, a single binary file with all the feature vectors. The `svl/lib/vision/svlCacheOutputUtils.h` contains the functions for reading and writing the feature vectors from files that are reused by different applications.

filterPatchDictionary

Trims the patch dictionary down to a more manageable size, eliminating features that are not very informative on the training set. The input is a patch dictionary and the feature vectors of the training examples, precomputed as described above. There are two types of filtering currently supported:

1. The first method uses the `svlPatchFeatureSelector` described in section 2.4.11. It uses the classification F-score as well as the correlation of patch-based features to perform the filtering.
2. The second method uses the mutual information between the features and the class labels (i.e. positive or negative example), as implemented in the `svlMIFeatureSelector` class and described in Section 2.3.10. It does not take into consideration the fact that the features are patch-based; it just considers their values and attempts to choose the most informative features that way. The algorithm was presented for object detection in [25]. This method can be turned on with the `-useMI` flag.

See Appendix B for configuration options.

trainObjectDetector

Trains a boosted classifier given the cached training example feature vectors. Note that if you used `filterPatchDictionary` to trim down the dictionary size, you will have to re-cache the examples again using `buildWindowFeatureCache` with the new dictionary to get the desired speed-up. The `trainObjectDetector` application can optionally perform cross-validation to determine the optimal number of boosting rounds to use (set with the `-cv` or `-cvn` flags). For more details about boosted classifiers, refer to Section 2.3.1.

Warning: *The code currently wraps the OpenCV boosting implementation which limits the size of the training sets. Refer to the scripts mentioned below, namely `svl/lib/trainObjectDetector.pl` and `svl/lib/fullyTrainObjectDetector.pl` for ways to work around it.*

trainClassifierPipeline

Encompasses the `buildPatchDictionary`, `buildWindowFeatureCache`, and `trainObjectDetector` from above, and is useful for running small randomized object classification trials. Given some positive and negative training images along with the `-numTrials` to run and the `-numPos` and `-numNeg` examples to use in each trial, it will repeatedly randomly select a training set of that size from the provided images, build a dictionary of features, train a classifier, and evaluate on the remaining images. This application is not currently optimized to filter the dictionary, and thus can be quite slow and memory-inefficient; however, it is very useful for running multiple trials with small datasets.

Tip: *This application is not recommended for use with large datasets.*

classifyImages

Given a dictionary to use for feature extraction, a trained classifier, and a set of images, runs the sliding window object detection and outputs the `svlObject2dSequence` of detections to a file. See Section 2.4.12 for details about the sliding windows algorithm. Note that it also has the option to immediately score the detections it produces (thus avoiding the `scoreDetections` application described below). It can either output a precision-recall curve directly or a summary file of detections at various thresholds. These summary files from different runs of the `classifyImages` program can later be combined to create a joint precision-recall curve. Output the detections is still recommended even if evaluation is being done on the spot, to be able to visualize them later (and re-run the evaluation as needed).

scoreDetections

Given pairs of object detection and groundtruth files, evaluates the detections against the labeled objects. See Section 2.4.13 for details about the `svlObjectDetectionAnalyzer` class. The application can optionally print out a precision-recall curve for each object of interest.

scoreSummaryDetections

Given the summary detections files from multiple runs of the `classifyImages` application on multiple images, creates a joint precision-recall curve. Note that this assumes that only one object was being detected.

Scripts

The main pipeline is provided for convenience in the script `svl/scripts/trainObjectDetector.pl`. Given positive and negative training image examples it will train a binary classifier for the object of interest using the `buildPatchDictionary`, `buildWindowFeatureCache`, and `trainObjectDetector` applications. It does so in two stages to avoid memory overflow when the number of training examples is too large. First, it runs through the pipeline and trains a boosted detector with just a subset of the training examples but with a full dictionary. Then it trims the dictionary using the `svl/scripts/trimDictionary.pl` script to only consider the features that were chosen by the boosting algorithm, and re-runs the pipeline with the full dataset.

The `svl/scripts/fullyTrainObjectDetector.pl` provides even more functionality, including building the training dataset with `buildTrainingImageDataset`, multiple ways of filtering the dictionary including `filterPatchDictionary`, evaluating the object detector on both the training and test images with `classifyImages` and `scoreDetections`, and retraining the classifier with false positive detections.

3.6.2 Pixel-level features

buildPixelDictionary

This application builds a vector-quantized feature dictionary as described in section 2.4.10. This dictionary can then be passed to the `buildWindowFeatureCache` and `trainObjectDetector` applications described in section 3.6.1, just as a dictionary produced from `buildPatchDictionary` can be. Note that sometimes vector quantization is not required (e.g. if you want to classify pixels individually and based on the full feature vector). In this case this application can still be used to create the `svlPixelDictionary` based on just raw features. It will simply output a basic XML file specifying the dictionary type which can then be read in by the applications described below.

buildPixelFeatureCache

Similar to `buildWindowFeatureCache` from section 3.6.1, it creates a feature cache with one feature vector per pixel in the image (sampled densely or using an interest point detector). `trainObjectDetector` can be used as before with these feature vectors to learn to distinguish two classes of pixels.

classifyPixels

This is an application for classifying all pixels within an image given an `svlPixelDictionary` and a `svlBinaryClassifier`. It mimics `classifyImages` described in section 3.6.1. The output can be a binary map image (classifying each pixel as positive or negative) or a heat map image representing the probability of each pixels being positive. It can also output the probabilities in a text file.

3.6.3 Object Detection utilities

trimFeatureExtractor

Removes unused features from a feature extractor description file. This is analogous to `trimDictionary.pl` but works on all types of feature extractors, not just the patch dictionary. This application requires the latest svn copy of OpenCV and will only work correctly if the line

```
OPENCV_DEVEL=1
```

is set in the `make.local` file at compile time.

visualizeDetections

Displays images with detections superimposed. Optionally creates an HTML page displaying all such images in a table. This can be useful for identifying the kinds of mistakes that a classifier makes.

Tip: Press '+' and '-' to increase or decrease the threshold for displaying detections

visualizePatchDictionary

Displays all the features from a patch dictionary. After training the boosted classifier, it is useful to verify that the chosen features visually seem reasonable for the object of interest.

Other Vision Utilities

See section 4.2 for more useful applications available in the `visionUtils` project.

3.6.4 Multi-class Image Segmentation

Multi-class image segmentation (or pixel labeling) aims to label every pixel in an image with one of a number of classes (e.g., grass, sky, water, etc). Since classifying every pixel can be computationally expensive, many state-of-the-art methods first over-segment the image into *superpixels* (or small coherent regions) and classify each region. The **SVL** implementation is a slight variant of the baseline method described in [10] and was also used in [12].¹ The following figure shows an example of an image, its over-segmentation, and region labels.



Figure 3.2: Multi-class image segmentation examples.

The basic method proceeds as follows:

- First, we extract appearance (color and texture), geometry and location features for each superpixel region.
- We then learn boosted classifiers over these features for each region class.
- Finally, we learn a CRF or logistic model using the output of the boosted classifiers as features.
- A step-by-step guide to the process is provided below.

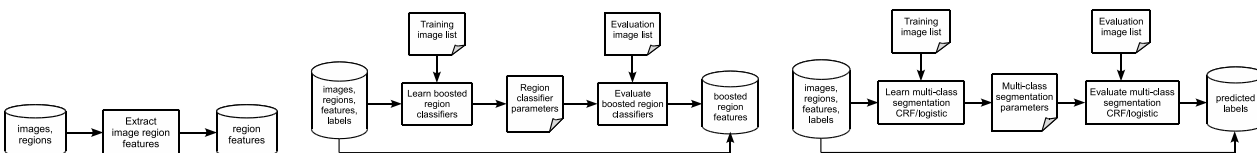


Figure 3.3: Multi-class image segmentation pipeline.

Data Preparation

The multi-class image segmentation algorithms that we describe take as input an image (`<base>.jpg`) and corresponding over-segmentation (`<base>.seg`) and ground-truth pixel labels (`<base>.txt`). The images are usually resized to approximately the same dimensions (e.g., 320-by-240). Over-segmentation and groundtruth labeling are described below.

Groundtruth Labeling: The groundtruth file (`<base>.txt`) corresponding to an image is a text file containing one number (integer) per pixel and arranged in a matrix the same size as the image. The integers correspond to different class labels and are zero-based. You can use the `regionLabeler` application to generate or view groundtruth files.

Over-Segmentation: The over-segmentation file (`<base>.seg`) corresponding to an image is a text file containing one number (integer) per pixel and arranged in a matrix the same size as the image. The

¹The relative location extension described in these works is not implemented in this version of the code.

integers correspond to different superpixels and are zero-based. Free online code such as that provided by Greg Mori [18] or Pedro Felzenszwalb [6] can be used for this purpose. Note that in our models each pixel is assigned to one and only one superpixel.

Extracting Region Features

Once the image regions/superpixels have been defined, we extract appearance (color and texture), geometry and location based features for each superpixel. These are cached in features files (`<base>.features.txt`) which contain one feature vector per superpixel. If the features are to be used directly as input to the logistic/CRF multi-class segmentation model, then a bias term (constant feature) can be appended which helps to model prior class preferences.

During this step we also extract the groundtruth label for each superpixel. This is done by finding the maximum occurring pixel label within each superpixel. The labels are written to corresponding labels file (`<base>.labels.txt`). This can be skipped if groundtruth labels are not available (such as when using a previously trained multi-class segmentation model to label new images).

The following command processes all `.jpg` (and corresponding `.seg`) files in `$IMAGEDIR` and output feature and label files to `$OUTPUTDIR`.

```
bin/segImageExtractFeatures -v -o $OUTPUTDIR $IMAGEDIR
```

Building Boosted Region Classifiers

Performance can be greatly improved if learn a boosted classifier for each class instead of using the raw appearance, geometry and location features described above. In this step we learn a one-versus-all classifier for each groundtruth label using the cached superpixel features and labels from the previous step. Once the classifiers are learned new feature files (`<base>.boosted.txt`) are created containing the output score for each of the learned classifiers on each superpixel in an image. Bias terms are also included so that these features files can be used directly by the logistic/CRF models.

```
bin/segImageTrainBoostedFeatures -v -featuresDir $OUTPUTDIR -o ${OUTPUTDIR}/demo \  
-rounds 200 -splits 2 $TRAININGLIST  
bin/segImageEvalBoostedFeatures -v -featuresDir $OUTPUTDIR -i ${OUTPUTDIR}/demo \  
-includeBias $NUMCLASSES $ALLIMAGESLIST
```

`$ALLIMAGESLIST` is the name of the file containing a list (one per line) of base names (image file names without the `.jpg` extension) to be processed. The file can be created using the following shell commands:

```
foreach i ($IMAGEDIR/*.jpg)  
  echo $i:r >> $ALLIMAGESLIST  
end
```

Learning the Multi-class Image Segmentation Model

Training involves learning the parameters of the logistic or CRF model. The logistic model assumes that each superpixel is independent and learns a multi-class logistic classifier based on the (raw or boosted) features. The CRF model adds a pairwise term between neighboring superpixels which acts to smooth the predicted labels.

```
bin/segImageTrainModel -v -o ${OUTPUTDIR}/demo.crf.model \  
-imgDir $IMAGEDIR -featuresDir $OUTPUTDIR \  
-model "CRF" -regNodes 1.0e-9 -regEdges 1.0e-2 $TRAININGLIST
```

The regularization parameters (`regNodes` and `regEdges`) should be set by cross-validation on the training set.

Evaluating the Multi-class Image Segmentation Model

After training, the accuracy of a model can be evaluated by comparing predicted labels against groundtruth labels on a hold out set of data (i.e., data that was not used during the training process). The model can also be used to label new images that will be used as input for some other vision task.

```
bin/segImageEvalModel -imgDir $IMAGEDIR -featuresDir $OUTPUTDIR \  
-model "CRF" ${OUTPUTDIR}/demo.crf.model $EVALLIST
```

3.6.5 Video Processing Utilities

A few very basic utilities are provide for dealing with video data. The `dumpVideoFrames` application will extract frames from a video file and save them as images. Going in the other direction, `images2video` takes a directory of images files and create an AVI video. The images are loaded in lexicographical order. Under Windows you will be prompted for the codec to use.

3.6.6 Depth Superresolution

The `mrfDepthSmooth` application produces superresolution depth maps by projection a point cloud into an image and applying a superresolution smoothing MRF [4, 9]. Concretely, let the image pixel intensities be $\{x_{i,j} \mid (i,j) \in \mathcal{I}\}$, the laser depth measurements be $\{z_{i,j} \mid (i,j) \in \mathcal{L}\}$ and the reconstructed/inferred depth for every pixel be $\{y_{i,j} \mid (i,j) \in \mathcal{I}\}$ where \mathcal{I} indexes the image pixels and $\mathcal{L} \subseteq \mathcal{I}$ indexes the laser measurements (projected onto the image plane). Two MRF potential functions are defined—the first penalizes discrepancy between measured and reconstructed depths, while the second encodes a preference for smoothness:

$$\Phi_{ij}(\mathbf{y}, \mathbf{z}) = h(y_{i,j} - z_{i,j}; \lambda) \quad (3.1)$$

$$\Psi_{ij}(\mathbf{x}, \mathbf{y}) = w_{ij}^v h(2y_{i,j} - y_{i,j-1} - y_{i,j+1}; \lambda) + w_{ij}^h h(2y_{i,j} - y_{i-1,j} - y_{i+1,j}; \lambda) \quad (3.2)$$

where $h(x; \lambda)$ is the Huber penalty function, and $w_{ij}^v = \exp\{-c\|x_{i,j-1} - x_{i,j+1}\|^2\}$ and $w_{ij}^h = \exp\{-c\|x_{i-1,j} - x_{i+1,j}\|^2\}$ are weighting factors indicating how unwilling we are to allow smoothing to occur across vertical and horizontal edges in the image as in [4]. The super-resolution MRF as

$$p(\mathbf{y} \mid \mathbf{x}, \mathbf{z}) = \frac{1}{\eta(\mathbf{x}, \mathbf{z})} \exp \left\{ -k \sum_{(i,j) \in \mathcal{L}} \Phi_{ij} - \sum_{(i,j) \in \mathcal{I}} \Psi_{ij} \right\} \quad (3.3)$$

where k specifies the trade-off between measurement reconstruction and smoothness, and $\eta(\mathbf{x}, \mathbf{z})$ is the normalization constant (partition function).

The following command line shows typical usage:

```
set CAMERAPARAMETERS = ...  
set BASENAME = ...  
bin/mrfDepthSmooth -camera ${CAMERAPARAMETERS} -rescale 0.5 \  
-o ${BASENAME}.reconstructed ${BASENAME}.jpg ${BASENAME}.txt
```

Chapter 4

Projects

4.1 Building Your Own Projects

Projects using the **SVL** are typically developed in subdirectory under the **projects** directory. The following is a prototype for a project **Makefile** for Linux. Application source code (i.e., **.cpp** files with a **main()** function) should be listed in **APP_SRC**, other source code such as shared classes that are linked into multiple applications should be listed in **OTHER_SRC**.

```
# STAIR VISION PROJECT MAKEFILE
# Stephen Gould <sgould@stanford.edu>

LASIK_PATH := $(shell pwd)/../..

USE_OPENCV = 1
USE_EIGEN = 1
USE_WX = 0

-include $(LASIK_PATH)/make.mk

#####

APP_SRC =

OTHER_SRC =

#####

APP_PROG_NAMES = $(APP_SRC:.cpp=)
APP_OBJ = $(APP_SRC:.cpp=.o)
OTHER_OBJ = $(OTHER_SRC:.cpp=.o)

.PHONY: clean
.PRECIOUS: $(APP_OBJ) $(OTHER_OBJ)

all: depend ${addprefix ${BIN_PATH}/, $(APP_PROG_NAMES)}

${BIN_PATH}/%: %.o $(OTHER_OBJ) $(LIBSVL)
    ${CCC} $*.o -o $@ (:o=) $(OTHER_OBJ) $(LFLAGS)

$(LIBSVL):
    @echo "**_YOU_NEED_TO_MAKE_THE_SVL_LIBRARIES_FIRST_**"
    false

.cpp.o:
    ${CCC} ${CFLAGS} -c $< -o $@
```

```

depend:
    g++ ${CFLAGS} -MM ${APP_SRC} ${OTHER_SRC} >depend

clean:
    -rm $(APP_OBJ)
    -rm $(OTHER_OBJ)
    -rm ${addprefix ${BIN_PATH}/, $(APP_PROG_NAMES)}
    -rm depend

-include depend

```

Warning: *If you choose to write your own Makefile (e.g., for inclusion in other projects), then make sure the **SVL** libraries are included in reverse dependency order, i.e., `svlVision.a`, `svlML.a`, `svlPGM.a`, `svlBase.a`.*

You can have the build system automatically make your projects by adding them to the `SVL_PROJECTS` variable in your `make.local` file. For example, the following `make.local` will cause projects `projects/helloworld` and `mydir/myproject` to build when `make` is invoked:

```
SVLPROJECTS := projects/helloworld mydir/myproject
```

Under Windows you can set up a new project by following these instructions:

- Create a Microsoft Visual Studio solution and add one Visual Studio project for each application. The projects should be created as **Visual C++ | General | Empty Project** types.
- Add your source files to the appropriate projects.
- For each project, select **Project | Properties (Alt-F7)** and set the following options:
 - Output Directory: `..\..\bin`
 - Additional Library Directories: `..\..\bin; "C:\Program Files\OpenCV\lib"; "C:\Program Files\OpenCV\otherlibs\highgui"`
 - Additional Dependencies: `svlBase.lib svlML.lib svlPGM.lib svlVision.lib xmlParser.lib lbfgs.lib svm.lib cv.lib cxcore.lib highgui.lib ml.lib kernel32.lib user32.lib`
 - Additional Include Directories: `..\..\include; ..\..\external; ..\..\external\eigen; "C:\Program Files\OpenCV\cv\include"; "C:\Program Files\OpenCV\cxcore\include"; "C:\Program Files\OpenCV\otherlibs\highgui"; "C:\Program Files\OpenCV\ml\include"`

If you are developing GUI applications then you will also need to link to the `wxWidgets` libraries.

4.2 Vision Utilities

The `visionUtils` project provides various applications that are useful when setting up and analyzing computer vision experiments.

4.2.1 combineDictionaries

Combines multiple patch dictionaries into one. Each input dictionary must specify a single channel from which its features were extracted (-1 to keep the original channels).

4.2.2 computeDetectionStatistics

Given a file of object detections and a corresponding image sequence, outputs for each type of object the number of detections, the average ratio, the minimum size of a detection, etc. It is useful for quickly getting the feel for a new labeled dataset or to sanity-check the output of `classifyImages`.

4.2.3 `countSlidingWindows`

Counts the number of sliding windows that would be analyzed in a given image with the given `svlSlidingWindowDetector` settings.

4.2.4 `dir2imageSeq`

Takes an image directory and converts all the images to an `svlImageSequence` object, which is then written to a file. Most object detection applications expect an image sequence as input.

4.2.5 `dumpVideoFrames`

Dumps undistorted video frames as JPEGs to disk. Can also be used for extracting objects out of video frames.

4.2.6 `extractorCat`

Concatenates feature extractor definitions. For example, given a patch dictionary file and a file describing a set of HAAR features, `extractorCat` makes a new file that describes an `svlCompositeFeatureExtractor` that extracts both the patch response features and the HAAR features.

4.2.7 `labelMe2ObjectSequence`

Converts a directory containing LabelMe XML-formatted annotations to an `svlObject2dSequence` file. The LabelMe polygons are converted to tight bounding boxes.

4.2.8 `processDetections`

Provides a set of utilities for processing detection XML files (corresponding to `Object2dSequence` classes). New features are still being added, but some of the current ones include scaling all detections by some factor, removing all detections less than a certain size, removing all detections from frames that are not in a given `svlImageSequence`, expanding all detections by a specified number of pixels, and so on.

4.2.9 `processImageSequence`

Provides a set of utilities for processing image sequence XML files. As with `processDetections`, new features are still being added, but the current ones include randomizing the sequence of images, splitting it up into a specified number of folds, adding in another image sequence, or, on the other hand, removing images that are present in another image sequence.

4.2.10 `splitImages`

Allows one to work with very large images by splitting them into more manageable-sized regions and saving those as separate images to be read in and analyzed by the various vision applications. Alternatively, it can also combine a set of small images back into one large image.

4.2.11 `summarizeExtractor`

Prints the contents of a feature extractor definition. For example,

```
[[11360 intensity patches, 4968 depth patches][1 Haar features]]
```

One use of this application is to determine which features were selected by boosting, by comparing the output of `summarizeExtractor` before and after running `trimFeatureExtractor`. This can also be useful for debugging. For instance, if `buildPatchDictionary` rejects several of one kind of patch, then that channel of your data might not be formatted correctly.

4.2.12 Scripts

Two relevant and useful scripts can be found in `svl/lib/scripts`. They address the fact that detection XML files are often produced by running the sliding windows detectors and end up being extremely large if the threshold for outputting detections is set too low. As a result, parsing the entire XML file might take a ridiculous amount of time. `splitDetectionsByFrame.pl` takes a file corresponding to an `Object2dSequence` with multiple frames and splits it into multiple `Object2dSequence` files, one per frame. These can then be analyzed individually, e.g. with the `processDetections` application, and merged back together using `combineDetectionFrames.pl`.

Warning: *These scripts don't check the input and assume that it is correct. All they do is copy lines between files. Please use carefully.*

Appendix A

Coding Guidelines

Software is written for people, not for machines.

The key to a successful (large) programming project, like any other project, involves planning, management, and testing. The amount of time, effort and emphasis placed on each of these three components depends on the size of the project and experience of the people involved. Consistency in design and implementation is also key to success and is what we will address in this document by outlining some (fairly standard) coding conventions. These conventions will make the project more manageable over time especially when many different people are involved.

It is absolutely guaranteed that some users will not like the style guidelines, and that others will even hate them. Everyone has their own style which they prefer to use on their own projects. But even though this is so, please understand that your team-mates, as well as yourself, will benefit greatly from the uniformity which they offer.

Finally, since the **STAIR Vision Library** is designed to be platform-independent, it is essential that you make don't implement any platform specific functionality. Following these guidelines will help reduce the amount of incompatibility introduced by developing in multiple environments. Also, before writing a new class or function, check to see whether one already exists that does what you want, or nearly what you want. Can that function be generalized to meet your needs?

A.1 Source Control

- Always use a source/revision control system. Some good packages include svn (free) and Perforce (commercial). In this project we use svn.
- Check-in all code and configuration files necessary for rebuilding a project from a clean environment. Do not check-in files that can be regenerated.
- In direct contradiction to the above, it is sometimes useful to check-in (non-standard) external packages that the project requires.
- Don't forget to add new source and header files before doing a commit.
- The latest code checked into the repository is assumed to be correct. Always make sure you merge your code with the latest revision before checking in. If you're working on something experimental then create a separate branch which you can merge later.
- Always make sure your code compiles and passes any regression tests before checking-in. Remember if you break something it's likely to affect a lot of other people.
- Check-in code regularly. Don't be afraid to check-in small changes.
- Add comments when you check-in code. This will help people understand what you were trying to do when they check-out your code and it doesn't work.

A.2 Structure

- Always include a header comment at the top of each file. The header should include the name of the project, name of the file, sometimes a copyright notice, and most importantly your name and email address.
- Group the declaration of public and private members. Separate the declaration of methods from variables.
- Likewise, group common header files together, starting from standard headers (e.g., `stl`) through to project specific headers.
- Declare constructors and destructors before other methods.
- Declare like-functions together.
- Always implement functions in the *same* order in which they are declared in header files or in the prototype section at the top of the file.
- Code should be implemented in `.cpp` files not `.h` files. Exceptions are templated and short inline functions.
- Name a file the same as the class that it implements. It is okay to implement multiple classes in the same file if they logically belong together. In this case find a filename that is appropriate to the group of classes.

A.3 Variable and Object Naming

- Use all-caps for constants and macros.
- All variables should be named starting with a lowercase letter.
- Prepend an underscore to private data members.
- Prepend 'b' to boolean types, 'g' to global types.
- Descriptive variable names are strongly preferred to compactified names. Exceptions are allowed for standard technical equations. For example, to evaluate a quadratic equation it is acceptable to write $y = a * x * x + b * x + c;$
- Single letter variables should be restricted to either loop iterators (preferably i, j, or k), or terms in very local computations (i.e., it is okay to use 'x' in a computation only if the scope of 'x' is less than a few dozen lines of code at the most).
- All **STAIR Vision Library** classes should begin with `sv1`.

A.4 Comments

- **It is a waste of time to write software without comments!** Your comments don't need to be lengthy, but they should be informative.
- Make sure you update your comments whenever you change your code.

A.5 Portability and Maintainability

- Never use variable or object names that could be keywords under different systems (e.g. `min`, `max`, `win`, `file`, `interface`).
- Use standard libraries (available on all platforms)—in particular, the `stl`. Don't reinvent the wheel.
- On a similar note, use standard file formats (e.g., XML). Put version numbers in parameter/model files so that you can read them back even if you change the format later.
- Keep object interfaces short and simple, it will make it much easier for other people to learn and use.
- When composing `stl` datatypes make sure you put a space between the `>` characters, for example `vector<vector<double> >`. Some compilers will (correctly) interpret `>>` as an operator and will generate an error.

- Some compilers like to have a blank line at the end of all source files. If you're working in multiple environments then it is good practice to do this.
- Set your editor to replace tabs with spaces.
- Use `svlCompatibility.h` to define symbols that are available on one platform but not another.
- Do not have two (or more) file names that differ only in their character case—for one thing this will confuse SVN under Windows.
- Do not have code with side-effects inside `assert` statements. Often `asserts` get commented out for release builds and you do not want the behaviour of your code to change.

A.6 Performance

- Don't copy large data structures around. Rather pass by reference (&) or by pointer (*).
- Use `const` whenever you can.
- Avoid allocating and deallocating memory in tight loops—rather allocate all the memory you need outside of the loop, but don't forget to deallocate the memory eventually.
- Don't use `printf`'s (or output stream operators) in tight loops.
- Use `reserve` to allocate memory to vectors and other stl datatypes before populating.

A.7 Miscellaneous

- Read the first chapter of “The Mythical Man-Month” [3] and remember that, although it was written in 1975, it still applied today.
- Avoid using `#define` when you can use a `const` variable or `enum` instead.
- Use structures, unions and classes to keep related variables together.
- Limit the use of global and static variables.
- Enclose conditionally executed code in braces (e.g., after an `if` or `for` statement) even if the code is only a simple statement. This will prevent bugs later on when you modify the code.

A.8 Testing

- Write and use regression tests.
- Make sure your code compiles without any compiler warnings.
- If you discover a (non-trivial) bug, first write a simple test that exposes the bug, before debugging. Then add the test to your regression test suite.
- Use lots of `assert()`'s. You can always compile them out for speed later. However, beware not to have side-effects inside the `assert`.
- Run you code while watching system memory (Task Manager under Windows or `top` under Linux) to identify memory leaks.
- If you have a bug, first think about where in the design the bug could be, before jumping into the code.

Appendix B

Configuration

The following table specifies the standard configuration attributes that are available in the **STAIR Vision Library**. For many application you can use the option `-config` by itself on the command line to get a list of the registered modules and their usage.

MODULE	ATTRIBUTE	DESCRIPTION
svlBase.svlCodeProfiler	enabled	Enable code profiling.
svlBase.svlLogger	logLevel logFile	Set the log verbosity level. Can be “ERROR”, “WARNING”, “MESSAGE” (default), “VERBOSE” or “DEBUG”. Append messages to a log file with the given filename.
svlBase.svlThreadPool	threads	Sets the maximum number of threads allowed. A value of 0 will cause all multi-threaded code to run in the main thread.
svlML.svlBoostedClassifier	boostMethod boostingRounds trimRate numSplits pseudoCounts quietTraining	Sets the boosting method. Can be “GENTLE” (default), “DISCRETE” or “LOGIT”. Number of rounds of boosting. Rate at which least weighted samples are ignored. Number of splits in each weak learner’s decision tree. Pseudo-counts for each class. Training verbosity.
svlML.svlConfusionMatrix	colSep rowBegin rowEnd	String to place between columns in the confusion matrix, e.g., “</td><td>”. String to place at the start of each row, e.g., “<tr><td>”. String to place at the end of each row, e.g., “<td></tr>”.
svlML.svlFeatureSelector	minSize maxSize	Minimum number of features to choose Maximum number of features to choose
svlML.svlLogistic	maxIterations eps	Sets the maximum number of iterations during training. Sets the training convergence threshold.

MODULE	ATTRIBUTE	DESCRIPTION
svlML.svlMIFeatureSelector	threshold	Minimum required joint MI increase for a feature to be added (overridden by svlFeatureSelector's minSize)
svlML.svlSVM	kernel degree epsilon posWeight C gamma numFolds	LINEAR (default), POLY, or RBF degree of the POLY kernel (default: 3) convergence criteria (default: 0.0001) weight for the positive class (default: 1; ≤ 0 means the weight will be set s.t. total weight of positive and negative examples will be the same) regularization parameter (default: 0; see section 2.3.1 for cross-validation configs) width of RBF kernel (default: $1.0/\text{numFeatures}$; see section 2.3.1 for cross-validation configs) number of cross-validation folds (default: 2)
svlPGM.svlFactorOperations	cacheIndexMapping useSharedIndexCache	Cache factor index mappings for operators (faster but uses more memory). Share cached indices (significant memory reduction for regular MRFs).
svlPGM.svlDualDecompositionInference	eps	Duality gap convergence threshold.
svlVision.svlImageLoader	channels defaultExtension resizeWidth resizeHeight depthMapWidth depthMapHeight useMask maskExtension ignoreExtensions	Single string of space-separated or colon-separated channel-extension pairs e.g. 'INTENSITY .jpg EDGE .jpg', 'INTENSITY: .jpg:EDGE: .jpg' or 'INTENSITY - EDGE -' with defaultExtension (default) Default: .jpg Resizes each image after loading Resizes each image after loading Required if using depth maps Required if using depth maps Mask out a region of each loaded image specified in the file named [imageName][maskExtension] Default: .occ Loader ignores files with these extensions, e.g. '.txt .bmp'
svlVision.svlObjectDetectionAnalyzer	areaRatio useNonmax includeAllFrames	Amount of overlap with the groundtruth for a detection to be considered positive (default: 0.5) Suppresses non-maximal detections (default: true) Includes all frames from the groundtruth files, even the ones not present in the detection file (default: false)

MODULE	ATTRIBUTE	DESCRIPTION
svlVision.svlPatchFeatureSelector	fThreshold ccThreshold	required minimum F-score of patches to be added (default: 0.75; overridden by svlFeatureSelector's minSize) threshold for cross-correlation of patches (default: 0.9)
svlVision.svlSlidingWindowDetector	deltaX deltaY deltaScale threshold rawScores	Shift in pixels in the x-direction between successive windows (default: 4) Shift in pixels in the y-direction between successive windows (default: 4) The change in scales between successive levels (default: 1.2) The minimum probability required for a detection to be returned (default: 0.5) Output raw detector scores (unnormalized log-probabilities) (default: false)
svlVision.svlTrainingDatasetBuilder	objects overlapThreshold texture allWindows baseScaleOnly skipPos posBestWindow skipNeg falsePositivesOnly detectionsFilename threshold includeOtherObjects numNegs	String of space-separated object names to be considered positive, e.g. ‘‘mug monitor’’ Minimum ‘‘overlap’’ with the groundtruth for a detection to be considered positive (default: 0.5) ‘‘Overlap’’ between a window W and a groundtruth box G is now defined as $\frac{area(W \cap G)}{area(W)}$ (default: $\frac{area(W \cap G)}{area(W \cup G)}$) Returns all sliding windows as (positive or negative) examples (default: false) Only the sliding windows at the base scale are considered (default: false) <i>Related to positive examples:</i> Skip positive examples (default: false) For each positive detection returns the sliding window with the biggest overlap (default: false; incompatible with allWindows) <i>Related to negative examples:</i> Skip negative examples (default: false) Only includes false positives from detectionsFilename (default: false) svlObject2dSequence file with object detections Threshold at which a detection from detectionsFilename is considered positive (default: 0.0) Include all objects from the groundtruth file whose names don't match objects as negative examples (default: false) Total number of negative examples to produce per scene (default: 100)

MODULE	ATTRIBUTE	DESCRIPTION
	negHeight	Desired height of negative examples (default: random, ≥ 32)
	negAspectRatio	Desired aspect ratio (w/h) of negative examples (default: random s.t. $w, h \geq 32$ or <code>resizeWidth/resizeHeight</code> if those are specified)
	<i>Related to writing out examples:</i>	
	baseDir	Base directory to write to (default: .)
	useWinRefs	Write the example references in <code>baseDir/imageSeq.OBJECT.xml</code> files rather than outputting individual training image files
	createDirs	Create output directories within <code>baseDir</code> , one with <code>negSubdirName</code> , and one per positive object with the object name (default: false)
	negSubdirName	Name of the subdirectory for negative examples (default: negative)
	posPrefix	Prefix for final name of positive images
	negPrefix	Prefix for final name of negative images
	includeFlipped	Include horizontally flipped examples (default: false)
	resizeWidth	Resize examples before writing
	resizeHeight	Resize examples before writing

Appendix C

License

Copyright (c) 2007-2010, Stephen Gould
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of Stanford University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ‘‘AS IS’’ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Bibliography

- [1] Stephen Boyd and Lieven Vandenbergh. *Convex Optimization*. 2004. 36
- [2] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. In *ICCV*, 1999. 27
- [3] Fred Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975. 79
- [4] J. Diebel and S. Thrun. An application of markov random fields to range sensing. In *NIPS*. 2006. 52, 72
- [5] Gal Elidan, Ian McGraw, and Daphne Koller. Residual belief propagation: Informed scheduling for asynchronous message passing. In *UAI*, 2006. 27
- [6] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient graph-based image segmentation. *IJCV*, 2004. 71
- [7] V. Franc and V. Hlavac. A novel algorithm for learning support vector machines with structured output spaces. *CTU-CMP-2006-04*, 2006. 36
- [8] Amir Globerson and Tommi Jaakkola. Fixing max-product: Convergent message passing algorithms for MAP LP-relaxations. In *NIPS*, 2007. 27
- [9] Stephen Gould, Paul Baumstarck, Morgan Quigley, Andrew Y. Ng, and Daphne Koller. Integrating visual and range data for robotic object detection. In *ECCV Workshop on Multi-camera and Multi-modal Sensor Fusion Algorithms and Applications (M2SFA2)*, 2008. 72
- [10] Stephen Gould, Jim Rodgers, David Cohen, Gal Elidan, and Daphne Koller. Multi-class segmentation with relative location prior. *IJCV*, 80(3):300–316, 2008. 70
- [11] Stephen Gould, Fernando Amat, and Daphne Koller. Alphabet SOUP: A framework for approximate energy minimization. In *CVPR*, 2009. 27
- [12] Jeremy Heitz, Stephen Gould, Ashutosh Saxena, and Daphne Koller. Cascaded classification models: Combining models for holistic scene understanding. In *NIPS*, 2008. 70
- [13] Andrew Johnson and Martial Hebert. Using spin images for efficient object recognition in cluttered 3d scenes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(1):433–449, May 1999. 52
- [14] Nikos Komodakis and Nikos Paragios. Beyond pairwise energies: Efficient optimization for higher-order MRFs. In *CVPR*, 2009. 27
- [15] S. Lazebnik, C. Schmid, and J. Ponce. A sparse texture representation using local affine regions. *Pattern Analysis and Machine Intelligence*, 27(8):1265–1278, Aug. 2005. 51
- [16] D.C. Liu and J. Nocedal. On the limited memory method for large scale optimization. In *Mathematical Programming B*, volume 45, pages 503–528, 1989. 16

- [17] K. Mikolajczyk and C. Schmid. Scale and affine invariant interest point detectors. *International Journal of Computer Vision*, 60(1):63–86, 2004. [47](#)
- [18] Greg Mori, Xiaofeng Ren, Alexei A. Efros, and Jitendra Malik. Recovering human body configurations: combining segmentation and recognition. In *CVPR*, 2004. [71](#)
- [19] J. Nocedal. Updating quasi-newton matrices with limited storage. In *Mathematics of Computation*, volume 35, pages 773–782, 1980. [16](#)
- [20] Oren Papageorgiou and Poggio. A general framework for object detection. In *ICCV*, 1998. [47](#)
- [21] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988. [27](#)
- [22] D. Sontag, T. Meltzer, A. Globerson, T. Jaakkola, and Y. Weiss. Tightening LP relaxations for map using message passing. In *UAI*, 2008. [27](#)
- [23] A. Torralba, K.P. Murphy, and W.T. Freeman. Sharing features: efficient boosting procedures for multiclass object detection. In *CVPR*, 2004. [48](#), [67](#)
- [24] R. J. Vanderbei. Lqoq: An interior point code for quadratic programming. In *Technical Report SOR-94-15*, 1998. [36](#)
- [25] M. Vidal-Naquet and S. Ullman. Object recognition with informative features and linear classification. In *ICCV*, 2003. [37](#), [67](#)
- [26] J. Winn, A. Criminisi, and T. Minka. Categorization by learned universal visual dictionary. In *ICCV*, 2005. [46](#)
- [27] C. Yanover, T. Meltzer, and Y. Weiss. Linear programming relaxations and belief propagation—an empirical study. *JMLR*, 2006. [63](#)