PraatLib 0.3

Michael Feld, Michael.Feld@dfki.de
Gerald Friedland, fractor@icsi.berkeley.edu

Introduction

PraatLib 0.3 is a Praat C++ Library Wrapper that contains routines designed mainly for three purposes:

1) To provide access to the full set of Praat functions as a C++ library which can easily be integrated into other applications
2) To provide performance optimizations of Praat routines, including some that may have a trade-off between accuracy and speed (in which case a switch is available)
3) To have a feature-oriented wrapper around the Praat functions. The idea is to provide a simple-to-use interface that takes care of the computing of all eventually needed intermediate objects while avoiding redundant computations i.e., you only need to specify a sound file and the features to be extracted.

This library is based on Praat (http://www.praat.org) version 5.0.3.0 source code, which is available under GNU General Public License v2. As a result, PraatLib is also available under the GNU General Public License v2.

Technicalities at a Glance

The library is provided as a static library (`libpraat.a`) which is pre-compiled for Red Hat 4 Linux, as well as a header file (`PraatSound.h`) containing the required declarations. To use it, include the header file in your C++ source code and link the library `libpraat.a` to your application (e.g. `g++ myapp.o libpraat.a`).

- The provided binary library is compiled for Red Hat 4 Linux. For use with different architectures, the library might have to be recompiled using the supplied Makefile (see “Compiling the Library”).
- The wrapper has to be called from C++ code. The rest of the functions should also be callable from C code.
- Some Praat functions are dependent on dynamic system libraries; however, it is very unlikely that they will ever need to be called in the library use-case.
- Using Praat as a native Win32 library will not be possible without a number of modifications, MAC OS X might work but we did not test it.

Basic Usage

If you want to use the Praat functions directly, you can do this by simply including the corresponding header files from Praat and calling the functions.

The wrapper is contained of a single class named *PraatSound*. You will need to create one such object for each sound file that is to be processed. Do not forget to destroy the object using *delete* afterwards to free the memory required by the sound and all cached intermediate data structures.

The constructor of *PraatSound* will require the file name of the audio file as an argument (either as normal or wide characters). Only file formats readable by Praat are supported (e.g. WAVE and NIST Sphere). **To achieve maximum computational efficiency, the entire audio file is read into memory.**

If an error occurs within the wrapper library for some reason, an integer exception is thrown and an error message is printed on the console.

After the sound file has been loaded, you can request features from it. There are basically two methods for doing this:
1) Calling each feature extraction function separately.
2) Calling the "mu lti-extract" function with a list of features that are to be extracted in one call.

The names of the feature extraction functions have to be looked up in the `PraatSound.h` header file or in the list below (see "Feature Index"). The function for extracting multiple features is

```
double* GetMultiple(char** features, int num_features, int segment = -2)
```

It receives an array of feature names (see "Feature Index"), the number of elements in the array, and – optionally – a segment number (see below). The return value will be an array with one value per feature.
Note: The approach using `GetMultiple(...)` is slightly slower because of the string comparisons, but this should not be a concern except for very optimized applications.

Most feature computation functions have associated parameters. When you are using individual functions to compute the features, the parameters are specified as function arguments. For easier handling, each parameter has a reasonable default value (taken over from Praat), so you do not need to specify any of them. Parameters and default values can be looked up in the header file. For the multi-feature-extraction method, there are public fields in the *PraatSound* class that can be set to the default values. However, because some variables are re-used for multiple features (e.g. the minimum frequency is used for both `pitch_mean` and `pitch_stddev` features), there are some cases where you have to call the individual functions.

The library will compute the required intermediate objects as needed, caching them when possible. Yet, only one object is cached for each combination of parameters, so if you need to extract several features with different sets of parameters, try to group them by their parameters (i.e. `pitch_min` and `pitch_max` with parameter set A together, instead of `pitch_min` with parameter set A and B together). The return value is always of type `double`. Sometimes features cannot be computed e.g., pitch for unvoiced segments. In this case, the special value `INF` is returned.

Specifying Segments

The library can extract features either on the full waveform or on segments. To accomplish this, each feature extraction function provides an optional "segment" argument, which is the first argument for all functions except `GetMultiple(...)`. If omitted or if `-1` is specified, it will default to using the entire sound file.

Segments are referred to by their index. To extract a feature for a specific segment, specify its index for the `segment` parameter. There two ways to define segments:

1) Call `SegmentsClear(int capacity)`with the maximum number of segments you want to add, then call `SegmentsAdd(double start, double end, WindowFunctions func)` to define each segment with its boundaries and (optionally) window function.
2) Call `SegmentsReadRTTM(char* filename, WindowFunctions func=PraatSound::wfRectangular)` to parse a NIST RTTM formatted file for segments. If a window function is specified, it will be applied to all segments.

You can also use the first approach to redefine your segments during usage.

The window function can be one of the following values defined inside *PraatSound*: `wfRectangular`, `wfTriangular`, `wfParabolic`, `wfHanning`, `wfHamming`, `wfGaussian1`, `wfGaussian2`, `wfGaussian3`, `wfGaussian4`, `wfGaussian5`, `wfKaiser1`, `wfKaiser2`. The value `wfRectangular` means essentially that no window function is applied (and corresponds to `wfNone` in earlier versions). Using a window function may require additional objects to be computed. This can increase the overall computation time by more than a factor of two if global features are also used.

When using the multi-feature-extraction function with segments, the default will be to compute the features for all segments, and the returned array will contain the feature vector for each segment. To force using the global waveform, specify `-1` for the `segment` parameter.

Important: Not all features can (or should) be computed on segments of any size (especially small segments may be problematic). For example, computing pitch on very small segments may return the `INF` value.

Performance notice: Features computed on segments may not always use cached objects created for features extracted on the full waveform. The exact behavior depends on the feature requested and on whether a non-rectangular window function is used.

Example

The directory `example` contains an example program that takes the filename of an audio file as argument and extracts some features.

To build the sample program, just run `make` from a shell inside the example directory. To run it, type `praatsample <audiofile>`, for example `praatsample test.wav`. Take a look at the `makefile` for the example program to find the external libraries needed for compiling applications that are using the Praat library for both static and dynamic linking.

You can change the features that are computed by editing the source code accordingly. You can also compute the features on segments and read from an RTTM file as opposed to the whole wave by the running praatsample <audiofile> <rttmfile>.

Compiling the Library

There is a directory src included in the package that contains the sources from which the Praat library (praat.a) is compiled. When you are making changes to the source code or when you want to port the library to another architecture or platform, you will need to recompile the code.

To build the library, run make from the src directory. The output (libpraat.a) will be copied to the parent directory.

Note: There may be some warnings when compiling the Praat sources. These warnings can usually be ignored.

Feature Index

This table lists the features that are available in the library. For more information on how they work and what they compute, please see the Praat manual (run the Praat binary, click menu "Help").

Function name	Name for GetMultiple()	Remark
GetPitchMean	f0_mean	
GetPitchMedian	f0_med	
GetPitchMin	f0_min	
GetPitchMax	f0_max	
GetPitchRange	f0_range	[1]
GetPitchStdDev	f0_stddev	
GetPitchMAS	f0_mas	[2]
GetPitchVoiced	f0_voiced	
GetPitchCandidatesMean	f0_cand_mean	
GetPitchCandidatesMedian	f0_cand_median	
GetPitchCandidatesMin	f0_cand_min	
GetPitchCandidatesMax	f0_cand_max	
GetPitchCandidatesRange	f0_cand_range	[1]
GetPitchCandidatesStdDev	f0_cand_stddev	
GetPitchStrengthMean	f0_str_mean	
GetPitchStrengthMedian	f0_str_med	

Function name	Name for GetMultiple()	Remark
GetPitchStrengthMin	f0_str_min	
GetPitchStrengthMax	f0_str_max	
GetPitchStrengthRange	f0_str_range	[1]
GetPitchStrengthStdDev	f0_str_stddev	
GetPitchEnergyMean	f0_en_mean	
GetPitchEnergyMedian	f0_en_med	
GetPitchEnergyMin	f0_en_min	
GetPitchEnergyMax	f0_en_max	
GetPitchEnergyRange	f0_en_range	[1]
GetPitchEnergyStdDev	f0_en_stddev	
GetPitchTierNumSamples	f0_samples	[2]
GetPitchTierMean	f0_mean_curve	
GetPitchTierStdDev	f0_stddev_curve	
GetPointProcessNumSamples	pp_samples	[2]
GetPointProcessNumPeriods	pp_periods	
GetPointProcessPeriodMean	pp_period_mean	
GetPointProcessPeriodStdDev	pp_period_stddev	
GetJitterRAP	jitt_rap	
GetJitterPPQ5	jitt_ppq5	
GetJitterLocal	jitt_l	
GetJitterLocalAbs	jitt_la	
GetJitterDDP	jitt_ddp	
GetShimmerAPQ3	shim_apq3	[2]
GetShimmerAPQ5	shim_apq5	[2]
GetShimmerAPQ11	shim_apq11	[2]
GetShimmerLocal	shim_l	[2]
GetShimmerLocalDb	shim_ldb	[2]
GetShimmerDDA	shim_dda	[2]
GetHarmonicityMean	harm_mean	
GetHarmonicityMedian	harm_med	[2]
GetHarmonicityMin	harm_min	
GetHarmonicityMax	harm_max	
GetHarmonicityRange	harm_range	[1]
GetHarmonicityStdDev	harm_stddev	
GetFormantMean	f1_mean, ..., f9_mean	
GetFormantMedian	f1_med, ..., f9_med	
GetFormantMin	f1_min, ..., f9_min	

Function name	Name for `GetMultiple()`	Remark
`GetFormantMax`	`f1_max, ..., f9_max`	
`GetFormantRange`	`f1_range, ..., f9_range`	[1]
`GetFormantStdDev`	`f1_stddev, ..., f9_stddev`	
`GetFormantsDispMean`	`form_disp_mean`	[1]
`GetFormantsDispMedian`	`form_disp_med`	[1]
`GetFormantsDispMin`	`form_disp_min`	[1]
`GetFormantsDispMax`	`form_disp_max`	[1]
`GetFormantsDispRange`	`form_disp_range`	[1]
`GetEnergyMean`	`en_mean`	
`GetEnergyMedian`	`en_med`	
`GetEnergyMin`	`en_min`	
`GetEnergyMax`	`en_max`	
`GetEnergyRange`	`en_range`	[1]
`GetEnergyStdDev`	`en_stddev`	
`GetLtasEnergyMean`	`ltas_mean`	[2]
`GetLtasEnergyMin`	`ltas_min`	[2]
`GetLtasEnergyMax`	`ltas_max`	[2]
`GetLtasEnergyRange`	`ltas_range`	[1] [2]
`GetLtasEnergyStdDev`	`ltas_stddev`	[2]
`GetLtasEnergySlope`	`ltas_slope`	[2]
`GetLtasEnergyLocalPeakHeight`	`ltas_lph`	[2]

Remarks:

[1] This feature is based on other features which are *not* cached, but are (a) fast to compute and (b) easy to combine. It is included to provide a more complete list of features. You may be able to get a minimal performance gain by manually computing the feature.

[2] This feature is always computed on full sound objects. If you are using segments without a window function and compute this feature on a segment, there is some additional overhead of copying the sound and re-creating intermediate objects. This means that if you are using only a single feature of this type under the aforementioned conditions, consider dropping it if it's not essential when you want to improve performance.