ChirpLab 1.1 : User's Guide

Emmanuel Candès and Hannes Helgason
Applied and Computational Mathematics, Caltech, Pasadena, CA 91125

Philip Charlton
School of Science and Technology, Charles Sturt University, Wagga Wagga NSW 2678, Australia

April 2006

1 Introduction

ChirpLab is a collection of MATLAB-routines that can be used to detect chirping signals from very noisy data [1]. This guide explains how to use this software; it includes installation instructions together with a brief description of the code, and some demos. For example, ChirpLab will allow the user to reproduce all the numerical experiments presented in the companion paper [1]. We expect that the user will expand the library and tailor the software for her/his particular purpose. There is also little doubt that ChirpLab's performance and design may still be optimized. In summary, ChirpLab is a collection of core data structures and algorithms which may be used as a starting point, and which we expect will become richer in the future.

2 Installing ChirpLab

2.1 Requirements

The software requirements for ChirpLab are

- `tar` and `gunzip` to install the package on Mac OS X, Linux or Solaris, or `zip` for Windows.
- MATLAB version 6 or higher.

We have successfully compiled ChirpLab under the following operating systems and compiler versions:

- Solaris 9 (SunOS 5.9), `cc` 5.3
- Fedora Core 4 Linux, `gcc` 3.3.6
- Mac OS X

Compilation on other systems, such as Windows, should also work just fine.

2.2 Installing and starting up ChirpLab

ChirpLab is distributed as a compressed tar file for Unix-based systems or as a zip file for Windows systems. The current version is available on the ChirpLab home page [2]. To install,

1. Download the compressed tar or zip file.

2. Uncompress the archive at the desired location.
   For Linux and Mac OS X use

   ```
   tar -xzvf ChirpLab1_1.tar.gz
   ```

   For Solaris use

   ```
   gunzip -c ChirpLab1_1.tar.gz | tar xfv -
   ```

   In Windows, use `zip` to uncompress the zip file at the desired location.

This will create a directory tree rooted at `ChirpLab1_1/` containing the source code.

Starting up ChirpLab: Launch MATLAB, enter the ChirpLab root directory that you created when you uncompressed the archive and run:

```
>> ChirpPath
```

This will add the ChirpLab directories to your `MATLABPATH` during your MATLAB session. If you are using ChirpLab for the first time it will automatically compile the MEX source files if needed. If you run into trouble at this stage, please check that MATLAB is correctly configured to mex `.c` and `.cpp` files. If the problem persists, please contact us.

To test that ChirpLab is installed properly run (the output is shown):

```
>> FindBPDemo
--Running FindBPDemo.m--
Generating chirp...
Taking chirplet transform...
Generating chirplet graph and assigning costs... (this will take a while)
Running optimization routine for graph...
Done!
```

After running this demo, you should get two MATLAB figures looking similar to figures 2 and 3.

3 Chirplets and the Chirplet Graph: Brief Description

3.1 Multiscale chirplets

The methods in ChirpLab use a family of multiscale chirplets which provide good local approximations of a wide range of chirps. We assume we work in the time interval $[0, 1]$ and that our measurements are evenly sampled. For each $j \geq 0$, we let I denote the dyadic interval $I = [k2^{-j}, (k+1)2^{-j}]$, where $k = 0, 1, \ldots, 2^j - 1$. The multiscale chirplet dictionary is a family of functions of the form

$$f_{I,\mu}(t) := |I|^{-1/2} \, e^{\mathrm{i}(a_\mu t^2/2 + b_\mu t)} \, 1_I(t), \tag{1}$$

where $(a_\mu, b_\mu) \in \mathcal{M}_j$ is a discrete collection of offset and slope parameters which may depend on scale. One can think of the 'instantaneous frequency' of a chirplet as being linear and equal to $a_\mu t + b_\mu$ so that in a diagrammatic sense, a chirplet is a line segment in the time-frequency plane (see Figure 1).

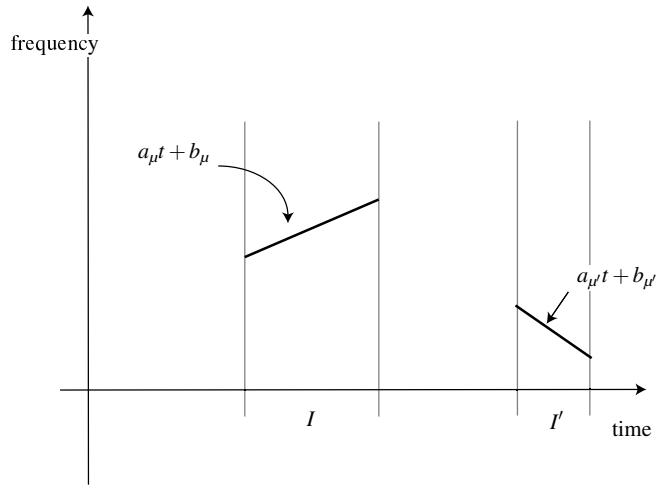

Figure 1: Diagrammatic representation of two chirplets in the time-frequency plane.

Remark: In ChirpLab, when we talk about the slope of a chirplet, we are referring to $a_\mu/(2\pi N)$ where N is the number of samples. The frequency offset (or simply the offset) of a chirplet refers to $b_\mu/(2\pi)$ and typically takes on integer values.

3.2 The chirplet graph

An important concept in ChirpLab is the *chirplet graph* $G = (V, E)$ where V is the set of nodes (or vertices) and E the set of edges. Each node in the graph is a chirplet index $v = (I, \mu)$. Nodes corresponding to chirplets starting at time $t = 0$ are said to be start-nodes, and nodes corresponding to chirplets ending at time $t = 1$ are said to be end-nodes. The edges between nodes are selected to impose a certain regularity about the instantaneous frequency. We say that two chirplets are connected if:

1. they have adjacent supports in time,
2. the frequency offset at the juncture is small,
3. and the difference in their slopes is not too large.

The first release of ChirpLab implements the first two constraints, namely, 1 and 2. The third type of constraint will be included in a future release.

4 Description of Code

The code is divided in three main parts:

1. *Chirplet Transform* – Algorithms for calculating chirplet coefficient tables.
2. *Networks* – Network flow algorithms for solving optimization problems on the chirplet graph, e.g. the shortest path, the constrained shortest path, and the minimum cost-to-time ratio.
3. *Utilities* – Scripts for plotting chirplets, source code for simulating noise and generating signals.

An object or data structure called the *Chirplet Graph Parameters* holds various parameters which link these three parts together; these parameters determine the slope discretization, the scales which are used and other properties of the chirplet transform and the chirplet graph. Section 5 shows how these parameters can be adjusted. The code also includes functions for generating simulated gravitational wave signals; see their MATLAB documentation for instructions on what they do and how they should be used. Section 10.1 provides a simplified example of how ChirpLab can be used to search for gravitational wave signals.

4.1 Packages

In the ChirpLab root directory, you will find:

`ChirpletTrans/`	–	Functions to compute chirplet transforms.
`Data/`	–	Directory holding data from earlier simulations.
`Demos/`	–	Some examples demonstrating how to use this software.
`Documentation/`	–	Documentation for ChirpLab.
`Networks/`	–	Network flow algorithms for finding the minimum cost to-time ratio, the shortest path, and for solving the constrained shortest path problem.
`Inspiral/`	–	Functions for generating simulated gravitational wave signals.
`Utilities/`	–	Scripting utilities (for plotting chirplets etc.), code for generating simulated noise and signals.

5 Initializing the Chirplet Graph

The MATLAB-routine used to set the chirplet graph parameters is `GetChirpletGraphParam` and is located in the directory `ChirpletTrans`. For a detailed description of that function, type `help GetChirpletGraphParam` at the MATLAB command prompt. In a nutshell, the adjustable parameters are:

- Coarsest and finest scales included in the chirplet graph.
- Slope range [minslope,maxslope]; that is, the range of $a_\mu/(2\pi N)$ in equation (1).
- Slope stepsize.
- Minimum and maximum frequency. i.e. the range of b_μ in equation (1)
- Degree of the polynomials used to fit the amplitude (see the varying amplitude chirplet transform).

A note about the default configuration: Assume that the number of samples is dyadic, $N = 2^J$. In the default setting, the coarsest and finest scales are set to 2^{-j} with $j = 0$ and $j = J - 1$ respectively (the number of time points per interval is N at the coarsest scale and 2 at the finest scale). The frequency offsets $b_\mu/(2\pi)$—at all scales—are integers ranging from 0 to $N - 1$. The slope parameters a_μ range from $-\pi N$ to πN with a discretization at scale 2^{-j} of the form $a_\mu = 2\pi N(-1/2 + k \cdot m2^{j-J})$; $m = 1$, $k \in \{0, \ldots, 2^{J-j}\}$ which ensures that the frequency offsets at the endpoints are all integers. For a full description of the default settings, type `help GetChirpletGraphParam` at the MATLAB prompt.

5.1 Customizing the slopes

In applications, one would probably want to tune the discretization of the slope parameter depending upon the type of signals under study. For example, we might have bounds on the chirping rate of the unknown signal (the speed at which the instantaneous frequency is changing) or we might only be interested in chirping

signals whose instantaneous frequency is nondecreasing with time. In the latter case, one could restrict the graph to chirplets with nonnegative slopes.

Consider the cubic phase chirp signal in the example discussed in Section 8.1.2. The instantaneous frequency is strictly increasing and just as explained before, suppose we only consider chirplets with nonnegative slopes. The example below shows how to configure the chirplet graph parameters so that the slopes belong to the interval $[0, 0.5]$:

```
discrStepFactor = 4; % parameter for discretization step
minslope = 0;        % minimum slope in chirplet graph
maxslope = 0.5;      % maximum slope in chirplet graph
graphparam = GetChirpletGraphParam(N,[],[],discrStepFactor,[minslope maxslope]);
```

The two empty brackets imply the default setting at the finest and coarsest scale but these parameters can also be specified. For a full example which shows how to do this, see `Demos/FindBPposSlopesDemo.m`. If you run this m-file, you should get the same figures as with `FindBPDemo.m`.

Further customization. Although `GetChirpletGraphParam` offers various slope discretization adjustments, it is still possible—with a little bit of extra work—to make your own discretization scheme by using the function `ChirpletTrans/GetSlopes.m` and then overwriting the slope parameters returned by `GetChirpletGraphParam`. Type `help GetSlopes` at the MATLAB prompt for information. The MATLAB code below gives an idea of how this can be achieved.

```
graphparam = GetChirpletGraphParam(....);
...
set up input parameters for GetSlopes
...
slopeparam = GetSlopes(...); % generate slope parameters
graphparam{3} = slopeparam; % overwrite slope parameters

... take chirplet transform and find the optimal path...
```

Connectivities in the graph. If a chirplet ends at a frequency offset that is not an integer, it will be connected to a chirplet starting at the nearest integer offset.

6 Chirplet Transforms

Three different chirplet transforms for calculating chirplet costs are supported in `ChirpLab 1.1`, see [1] for a definition of each chirplet cost:

(i) white noise and constant amplitude

(ii) white noise and polynomial amplitude

(iii) colored noise and constant amplitude (the noise spectrum needs to be provided)

In a future release, we will also include support for computing costs with time-varying amplitudes in colored noise. We will also provide code for computing real-valued chirplet costs.

The main function for calculating chirplet transforms is `ChirpletTrans/ChirpletTransform.m`. A typical call to this function is

```
cc = ChirpletTransform(sig,param)
```

where the variable `sig` stores a signal of dyadic length $N = 2^J$, $J \geq 1$, and the variable `param` stores the graph parameters as returned by `GetChirpletGraphParam` (see Section 5). The chirplet coefficients are stored as a cell array. There is, in general, no need to worry about how the parameters are stored since the cell array is passed directly to the network algorithms which we use to calculate the statistics of interest.

The Chirplet Coefficient Data Structure. Each element in the cell array is a table of chirplet coefficients corresponding to a specific dyadic time interval $[k2^{-j}, (k+1)2^{-j}]$, labeled (j,k) where $k = 0, 1, \ldots, 2^j - 1$, j is a scale index which can range from 0 to $J-1$. A table corresponding to a dyadic interval (j,k) is indexed by $2^j + k$ and is an n-by-(#slopes at scale j) matrix whose values are described as follows:

- Each column in the table corresponds to a specified slope. The kth column corresponds to the kth slope.
- Each entry in a column corresponds to a specified frequency offset. In each column, the mth entry corresponds to the frequency offset $2\pi(m-1)/N$.

Explaining the data structure with an example. This example assumes that the demo script `FindBPDemo` has been previously executed and that the MATLAB variables `graphparam` and `cc` storing the graph parameters and the coefficient table are still in the workspace. To get the slopes, we retrieve the third entry of the parameter data structure:

```
>> slopesAtAllScales = graphparam{3}
slopesAtAllScales =

Columns 1 through 4

  [1x513 double]    [1x257 double]    [1x129 double]    [1x65 double]

Columns 5 through 8

  [1x33 double]    [1x17 double]    [1x9 double]    [1x5 double]

Column 9

  [1x3 double]
```

Column k in `slopesAtAllScales` has the vector of slopes used at scale 2^{-k+1}. To retrieve the slopes at scale 2^{-j} where $j = 7$ do

```
>> j = 7;
>> slopesForj7 = slopesAtAllScales{j+1}

slopesForj7 =

   -0.5000   -0.2500         0    0.2500    0.5000
```

Now suppose that we would like to retrieve all the chirplet coefficients in the time interval $[k2^{-j}, (k+1)2^{-j}]$, where $k = 65$ and $j = 7$. We can use the function `node` as follows:

```
>> j=7; k = 65;
>> ccAtDyadInt = cc{node(j,k)};
```

Finally, we get the chirplet coefficient with a slope equal to 0.25—the third slope in `slopesForj7` above—and a frequency offset obeying $m = 100$ by typing

```
>> m = 100; slopeNo = 3;
>> ccoeff = ccAtDyadInt(m,slopeNo)
ccoeff =

   0.0768 + 0.0437i
```

6.1 Chirplet costs with time-varying amplitudes

Refined detection methods correlate the data with chirplets which—in addition to having a quadratic phase function—also have polynomial amplitudes (rather than constant amplitude). The current version of ChirpLab allows these polynomials to be of degree 0, 1 or 2; higher degrees are of little use here since we are interested in chirping signals for which the phase varies much more rapidly than the amplitude. One can invoke the function `ChirpletTransform` to compute coefficients against chirplets with time-varying amplitudes. One simply needs to set the parameter `xttype` in `GetChirpletGraphParam` to the char array `'VARAMP'`. At each scale, one can also select the degree of the polynomial one wishes to use. The following MATLAB script snippet shows how to invoke this function; the coarsest and finest scales are 2^{-s} with $s = 0$ and $s = 6$ respectively.

```
% we assume the data is stored in a vector y and is of length N=2^10=1024
csc = 0; % coarsest scale
fsc = 6; % finest scale
xttype = 'VARAMP';
degreeArray = [2 2 1 1 1 1 1]; % degrees for polynomials, coarsest to finest scale
graphparam = GetChirpletGraphParam(N,csc,fsc,[],[],[],[],xttype,degreeArray);
cc = ChirpletTransform(y,graphparam);
```

The scales in the graph are 2^{-s}, with $s = 0, 1, \ldots, 6$ so the total number of scales is 7. The array of polynomial degrees is also of length 7 enumerating scales from the coarsest to the finest. Thus in our example, we use quadratic polynomials at the two coarsest scales and linear amplitudes at all the other scales. The MATLAB script `Demos/VarAmpDemo.m` applies a time-varying chirplet transform.

6.2 Colored Noise

To calculate chirplet costs which assume colored noise, the parameter `xttype` in `GetChirpletGraphParam` has to be set to the char array '`COLOREDNOISE`.' Concretely,

```
xttype = 'COLOREDNOISE';
param = GetChirpletGraphParam(N,[],[],[],[],[],[],xttype);
```

The empty brackets imply default values for the respective parameters but they can be changed. Once the chirplet graph parameters have been set up, call `ChirpletTransform`:

```
C = ChirpletTransform(sig,param,S,Cnorm);
```

where `S` is the spectrum as before and `Cnorm` is the normalization cell array as returned by the function `ChirpNormColoredNoise.m`. It is possible to invoke the function without the variable `Cnorm`:

```
C = ChirpletTransform(sig,param,S);
```

Typically, one will compute many chirplet transforms with the same type of noise (the same noise spectrum) and the same chirplet discretization. In such cases, it is more efficient to precompute the normalization cell array and write it to a `mat`-file for future use. Here is an example:

```
Cnorm = ChirpNormColoredNoise(S,param);
```

The MATLAB-script `Demos/ColoredNoiseDemo.m` shows how to compute chirplet costs adjusted for colored noise.

7 Network Flow Algorithms for the Chirplet Graph

In the current ChirpLab release, three different optimization routines are implemented. Below, c_v is the cost/value assigned to the chirplet indexed by v which is a function of the corresponding coefficient (possibly adjusted to deal with time-varying amplitudes and colored noise). In the simplest situation, white noise + constant amplitude chirplets, $c_v = -|\langle y, f_v \rangle|^2$ where y is the data vector and f_v the chirplet (1). Let W be any path in the chirplet graph and $|W|$ be the number of chirplets in the path.

- Shortest Path
$$\min_W \sum_{v \in W} c_v. \tag{2}$$
- Best Path
$$\min_W \sum_{v \in W} c_v, \quad \text{s.t.} \quad |W| = 1, 2, \ldots, \ell, \tag{3}$$
where ℓ is an integer.
- Minimum Cost to Time Ratio
$$\min_{W \in \mathcal{W}_k} \frac{\sum_{v \in W} c_v}{|W|}, \tag{4}$$
where for each k, $\mathcal{W}_k$ is a subset of all paths in the chirplet graph. In ChirpLab, $\mathcal{W}_0$ could be chosen to be the set of all paths, $\mathcal{W}_1$ be the set of paths which cannot use chirplets at the coarsest scale, $\mathcal{W}_2$ be the set of paths which cannot use chirplets at the two coarsest scales, and so on.

The motivation behind these statistics and the algorithms used for efficient computations are discussed in [1]. Sections 8.1.1, 8.1.2 and 8.1.3 show how to compute these statistics in ChirpLab.

8 Tutorials

Below are some instructions for getting started with `ChirpLab 1.1`. A good idea is to go through the examples below or to check/run the scripts in the folder `ChirpLab/Demos`. For detailed description about each function, simply type `help <name of function>` in MATLAB.

8.1 Demos: finding the best paths in the chirplet graph

The steps below are required to compute the main statistics of interest:

1. Make a signal.
2. Select the chirplet graph/discretization parameters and apply the chirplet transform.
3. Initialize the graph with the chirplet costs and compute a test statistic.
4. Plot the chirplet paths.

Steps 1 and 2 are independent of steps 3 and 4. They only share the graph parameters.

Below is our first example (step 1 and 2):

```
% STEP 1
>> N = 2^9;
>> [y,omega]= MakeChirp('CubicPhase',N);

% STEP 2
>> graphparam = GetChirpletGraphParam(N);
>> cc = ChirpletTransform(y,graphparam);
```

A cubic phase chirp signal of length $N = 512$ is generated in step 1 and stored as `y`. The variable `omega` is the 'instantaneous frequency' (first derivative of the phase) of the chirp and is not used in any calculations; it may be interesting to plot it though, and compare how chirplet paths 'track' the instantaneous frequency. In step 2, one retrieves the default chirplet graph parameters and calculate the chirplet coefficients table. In Section 5, we saw how to configure the chirplet graph by passing other arguments to `GetChirpletGraphParam`. Note that one can omit the graph parameters and calculate the chirplet coefficients with the default setting by typing:

```
>> cc = ChirpletTransform(y);
```

However, it is recommended to always use the graph parameters especially since they have to be passed to the optimization routines and to the plotting utilities.

8.1.1 Shortest Path

To calculate the Shortest Path through the graph, one sets the second argument of the function `CalculateStatistic` to the string '`SP`.' Step 3 is as follows:

```
>> cnetwork = GetChirpletNetwork(cc,graphparam);
>> [costpath,shortestpath] = CalculateStatistic(cnetwork,'SP');
```

The variable `costpath` returns the sum over the chirplet costs along the shortest path. The variable `shortestpath` is a vector of numbers corresponding to the chirplets in that path. One can plot the shortest path by issuing the following command:

```
>> DisplayChirpletPath(shortestpath,graphparam);
```

Check out the demo-file `Demos/FindSPDemo.m`.

8.1.2 Best Path

Next we show how to calculate the BP test statistic and plot the chirplet paths which gives the best cost for a given length. We initialize the graph/network as before

```
>> cnetwork = GetChirpletNetwork(cc,graphparam);
```

(there is no need to do this again provided that the variables are still in the workspace). To calculate the Best Path statistic, we set the second argument of `CalculateStatistic` to `BPFORPLOTTING` and add a third parameter which sets the maximum number of chirplets that should be used in a path:

```
>> STATTYPE = 'BPFORPLOTTING';
>> maxLength = 5;
>> [costpaths,bestpaths] = CalculateStatistic(cnetwork,STATTYPE,maxLength);
```

The variable `costpaths` stores the costs of the best paths of length 1,...,`maxLength` and the variable `bestpaths` stores the corresponding chirplet paths as vectors in a MATLAB cell array of length `maxLength`. To plot the best path of length k, type

```
>> DisplayChirpletPath(bestpaths{k},graphparam);
```

The demo-file `Demos/FindBPDemo.m` shows how to calculate the BP statistic. If you execute this file, MATLAB should return plots similar to Figures 2 and 3. The chirplet graph in this demo is restricted to nonnegative frequencies by using the following commands

```
minfreq = 0;
maxfreq = N/2-1;
graphparam = GetChirpletGraphParam(N,[],[],[],[],minfreq,maxfreq);
```

Note that in the demo, the parameters `minfreq` and `maxfreq` are set at the beginning of the file.

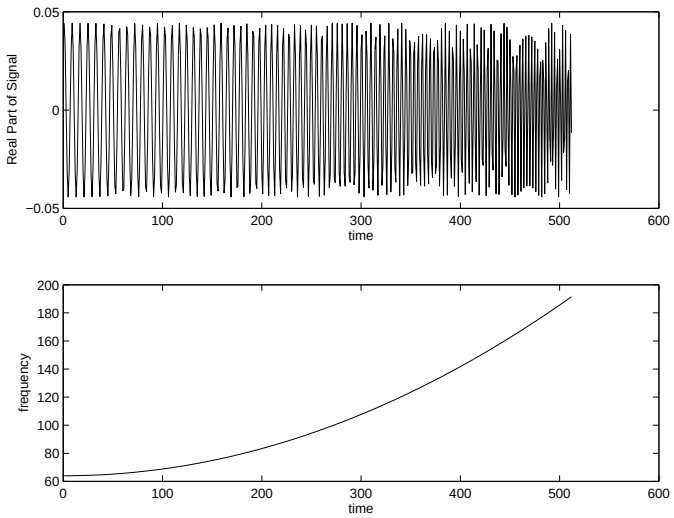

Figure 2: The first figure in the sample script `FindBPDemo.m`.

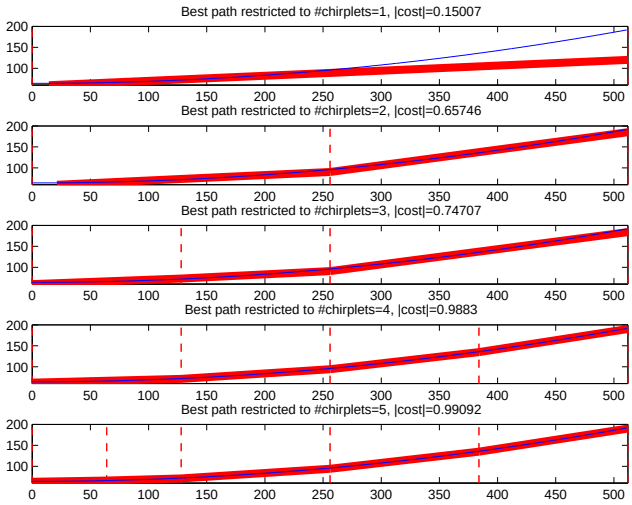

Figure 3: The second figure in the sample script `FindBPDemo.m`.

8.1.3 Minimum Cost to Time Ratio

To calculate the minimum cost to time ratio (MCTR), an additional argument needs to be passed to the function `GetChirpletNetwork`. This parameter sets the coarsest scale allowed in a path. In the example, the coarsest scale in the path is set to 2^{-s} with $s = 3$.

```
>> coarsestScale = 3;
>> cnetwork = GetChirpletNetwork(cc,graphparam,coarsestScale);
>> [mctrCost,mctrPath,nchirplets] = CalculateStatistic(cnetwork,'MCTR');
```

The variable `mctrCost` holds the value of the MCTR, i.e. the optimal value of (path cost)/(number of chirplets). The variable `nchirplets` holds the number of chirplets for the optimal path so that the actual cost of the path is `nchirplets*mctrCost`. One can plot the path in exactly the same way as before.

```
>> DisplayChirpletPath(mctrPath,graphparam);
```

Check out the demo file `Demos/FindMCTRDemo.m`.

8.2 Running Monte Carlo simulations with noisy data

ChirpLab includes routines for running simulations which illustrate the performance of the detection strategies introduced in [1].

8.2.1 Running simulations for pure noise and signal+noise

Suppose we wish to estimate the null distribution of a given test statistic and consider the following example:

```
% set signal length and initialize chirplet graph using the default settings
N = 2^6;
graphparam = GetChirpletGraphParam(N);
```

```
% Use the best path statistic with lengths 1,2,3,4
statType = 'BP';
statParam = 4;

numSims = 10; % number of simulations

% wrap the experiment setup
expSetup = ExperimentSetup(graphparam,statType,statParam,numSims);

% file name to store the data
fname = 'nullRunSimsDemo.mat';

% run the simulations
T = RunSimulations(expSetup,fname);
```

After execution, the variable T holds 10 realizations of the BP statistic with the cost of the best chirplet paths of lengths 1,2,3 and 4. Here the two key functions are:

- `ExperimentSetup.m` – This wrapper is meant to make experiments with the same setting easier to repeat. The settings are saved in in one `mat`-file. For simulations with noise only, one should of course omit the signal.
- `RunSimulations.m` – This runs the simulations as described in the experiment setup. The optional argument is a file name for saving the data to a file. In case, the signal is not provided, the routine will simulate the null distribution of the test statistic. In each realization, the data is of the form $z = z_r + \mathrm{i}z_i$, where z_r and z_i are two independent vectors with i.i.d. $N(0,1)$ entries.

  In case a signal and a SNR are provided, the function normalizes the signal such that

$$\mathrm{SNR} = \frac{\|s\|_{\ell_2}}{\sqrt{E\|z\|^2}},$$

  where $y = s + z$ is the simulated data, s is the normalized signal of length N. Note that $E\|z\|^2 = 2N$.

For further information about these functions, see the MATLAB documentation using `help`.

Running the demo script `RunSimsDemo.m` which includes the commands from the previous example gives

```
>> RunSimsDemo
--Running RunSimsDemo.m--
Initialize chirplet graph...
Run the simulations (this might take awhile)...
Done!
>> T

T =

  -18.4163  -19.5320  -23.4282  -27.0571
  -16.5960  -24.9926  -28.4492  -32.5346
  -18.9530  -22.6076  -25.1558  -28.6932
  -14.2429  -17.8198  -22.4245  -25.1301
  -20.2356  -22.0379  -28.2136  -31.5691
  -18.7165  -23.2990  -31.6998  -34.3740
```

```
  -10.5744  -15.5506  -18.2989  -23.4716
  -10.9623  -16.1651  -19.7368  -23.9579
  -19.7387  -21.7211  -32.1896  -36.1324
  -18.5750  -22.4598  -27.2383  -31.3057
```

After running the script, the variables `T`, `expSetup` and the random seed used to generate the data have been saved to the file `nullRunSimsDemo.mat` in the current directory. What does it contain?

```
>> load('nullRunSimsDemo');
>> who

Your variables are:

T         expSetup  seedUsed

>> expSetup

expSetup =

     graphParam: {[64 6]  [0 5]  {1x6 cell}  [0 63]  []  'PLAIN'}
         statType: 'BP'
      statParam: 4
        numSims: 10
                  sig: []
                 snr: []
    description: ''

>> seedUsed

seedUsed =

        1234
```

Note that the last three parameters in `expSetup` are empty since they were not set.

To run simulations with the signal of your choice, simply make a signal, select a SNR and add it to the experiment setup as shown in the following MATLAB snippet:

```
sig = MakeChirp('CubicPhase',N);
snr = 0.5;
% wrap the experiment setup
expSetup = ExperimentSetup(graphparam,statType,statParam,numSims,sig,snr);
```

The file `Demos/RunSimsAltDemo` does just this. It gives the following output:

```
>> RunSimsAltDemo
--Running RunSimsAltDemo.m--
Initializing chirplet graph...
Running the simulations (this might take awhile)...

T =
```

```
  -27.7749  -29.5230  -35.6481  -42.1786
  -28.9956  -29.8739  -34.7797  -38.5646
  -24.1944  -31.5743  -38.3908  -40.9896
  -25.9676  -34.4569  -37.1174  -45.3329
  -23.8026  -29.5673  -32.8835  -39.0411
  -22.8067  -42.3826  -48.9427  -54.8580
  -30.2571  -43.7777  -51.8028  -55.5910
  -30.7068  -47.2637  -52.0503  -53.4561
  -26.4720  -40.3218  -49.7803  -50.8223
  -24.5752  -41.4134  -42.4127  -49.2123

Done!
```

8.2.2 Estimating minimum P-values and calculating detection rates

In [1], one decides to reject the null hypothesis by checking the minimum P-value and comparing it with a threshold. ChirpLab has a routine called `EstimateMinPvalues` for estimating the minimum P-values given realizations of multivariate random variables.

Assume the variable `nullT` is a $B \times k$ matrix where each row corresponds to one realization of a k-dimensional random vector $T = [T_1, \ldots, T_k]$. B is the number of realizations. You might think about the random vector T as the best path costs corresponding to chirplet path lengths $1, \ldots, k$. To estimate the minimum P-value of the random vector $T' := [T_1, T_2, T_4, T_8, T_{16}]$, a subset of the entries in the full vector T, you can run

```
whichCoord= [1 2 4 8 16];
pmin = EstimateMinPvalues(nullT,[],whichCoord);
```

This estimates the minimum P-value for each of the B realizations of T' in `nullT`.

For a particular realization $T' = t'$, we estimate its minimum P-value by comparing it to an estimate of the distribution of T' done from the other $B - 1$ realizations.

Using `pmin`, one can estimate the distribution of the minimum P-value and find thresholds for given significance levels. The function below will compute these tresholds.

```
alpha = 0.05;
thresh = GetThreshold(pmin,alpha);
```

Here, this gives the threshold for a fixed type I error equal to 5%.

Suppose we wish to detect a signal in additive noise. Assume we have a variable `sigT` which is a $B' \times n$ matrix where each row corresponds to one realization of an n-dimensional random vector $y = s + z$, where z is a random vector of the same form as in `nullT` and s is a deterministic signal. B' is the number of realizations. To estimate the probability of detection for a fixed type I error, type

```
estPow = EstimateDetRate(nullT,sigT,thresh);
```

which returns estimated power curves/detection rates. For further information, check See the MATLAB documentation for `EstimateDetRate`.

The script `Demos/AnalyzeDataDemo.m` uses all these routines. It uploads results from simulations to save time. These data may be found in the directory `Data/ForDemos`.

9 Further releases

Some selected features under development that will be included in future releases of ChirpLab:

- Routines for calculating chirplet costs with time-varying amplitude chirplets and colored noise.
- Adjust the chirplet costs to handle real-valued data gracefully.
- Implement connectivities to impose a curvature constraint upon the instantaneous frequency along a chirplet path.
- Develop software for estimating chirps from noisy data (estimation as opposed to mere detection).

Stay tuned. Comments are welcome!

10 Appendix

10.1 An example with a gravitational wave signal

One application of note in conjunction with these methods is the problem of detecting gravitational waves in data from laser interferometric detectors such as LIGO [3]. The inspiral of two massive bodies (such as a pair of neutron stars) is a good candidate for a detectable source of gravitational waves. The signal detected by a LIGO-type detector is a sinusoidal wave

$$h(t) \quad = \quad A_+(t)\cos\phi(t) + A_\times(t)\sin\phi(t) \tag{5}$$

here expressed in terms of the two polarisations h_+ and $h_\times$. The strain $h(t)$ is the fractional change induced in the arm length of the interferometer by the gravitational wave. That is, if the arm length of the interferometer "at rest" is L, then $h(t) = \Delta L/L$. For a binary inspiral signal, the post^2-Newtonian approximation to the instantaneous frequency of $h(t)$ is of the form

$$f(t) = \dot{\phi}(t)/2\pi \quad = \quad a_0(t_c - t)^{-3/8} + a_1(t_c - t)^{-5/8} + a_2(t_c - t)^{-3/4} + a_3(t_c - t)^{-7/8} \tag{6}$$

where t_c is the time when the two bodies coalesce.

We have found that the Best Path statistic is most suitable for detecting signals of this form. Since the LIGO noise spectrum is strongly colored, we must also supply a spectrum for weighting the chirplet coefficients. The code `Demos/BinaryInspiralDemo.m` gives an example of how to set up the detection problem with a simulated signal and simulated LIGO noise. First we obtain a LIGO noise spectrum via

```
>> S = MakeLIGO1Psd(N, Fs, true);
>> S = Fs*S;
```

Here N is the number of samples in the signal (512 in the demo), the sampling rate `Fs` is 2048 Hz and the final parameter is a flag which tells the function to limit the spectrum's magnitude at low frequencies. Since LIGO power spectra are normally given in units of strain^2/Hz we multiply by `Fs` to get a spectrum with units of strain^2.

The gravitational wave signal is generated using the `inspiral` function:

```
>> [h, p, fr] = inspiral(t, m1, m2, tc);
```

where m_1 and m_2 are the masses of the two bodies (here both are taken to be 14 solar masses) and t is the vector of time values at which h is to be calculated. The function also returns the phase `p` and the instantaneous frequency `fr` in Hz (see Fig 4). We normalise h with respect to S by scaling it so that

$$\frac{1}{N}\sum_{n=0}^{N-1}\frac{|\tilde{h}_n|^2}{S_n} = 1 \tag{7}$$

where $\tilde{h}$ is the discrete Fourier transform of h.

The next step is to prepare the parameters for the chirplet graph. Since the data is real, we can restrict the search to positive frequency indices only, i.e. frequency indices from 0 (DC) to N/2 (Nyquist). This is set by specifying the 5th and 6th parameters of `GetChirpletGraphParam`:

```
>> transformType = 'COLOREDNOISE';
>> fmin = 0;
>> fmax = N/2;
>> graphparam = GetChirpletGraphParam(N,csc,fsc,[],[],fmin,fmax,transformType);
```

Since the spectrum is colored we also need to calculate the norm of each chirplet in the chirplet dictionary:

```
>> Cnorm = ChirpNormColoredNoise(S,graphparam);
```

This only needs to be done once in advance and the resulting norms, stored in `Cnorm`, can be re-used for each new instance of data.

For a single simulation we use S to generate an instance of simulated LIGO noise. The data used as the input to the chirplet transform is the sum of the noise and the normalised h multiplied by the signal strength:

```
>> n = MakeRealNoise2(S);
>> y = n + SNR*h;
```

Following this step we generate the chirplet transform of y, the chirplet graph, and the list of best paths as in Section 8.1.2:

```
>> cc = ChirpletTransform(y,graphparam,S,Cnorm);
>> cnetwork = GetChirpletNetwork(cc,graphparam);
>> STATTYPE = 'BPFORPLOTTING';
>> maxLength = 8;
>> [costpaths,bestpaths] = CalculateStatistic(cnetwork,STATTYPE,maxLength);
```

10.2 Compiling the MEX source using `make` (optional)

It is possible to compile the MEX sources outside MATLAB before you start up ChirpLab the first time. From a terminal, try running `make` from the ChirpLab directory:

```
make
```

If this returns errors make sure that your system fullfills the following requirements:

- a C/C++ compiler capable of creating MEX files (if you have MATLAB installed, you should be able to create MEX files from the MATLAB command prompt.

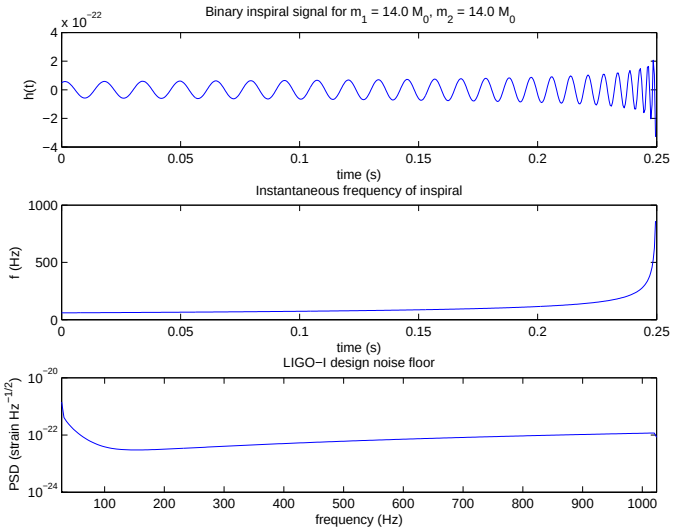

Figure 4: The first figure in the sample script `BinaryInspiralDemo.m`.

- GNU `make` (compilation will not work using non-GNU versions of `make`). On Solaris, aliasing "`make`" to "`gmake`" will usually work.

Some hand editing of the `Makefile.include` in the top level directory of ChirpLab may be required to set the correct extension for compiled MEX files.

10.3 Permanently adding the ChirpLab directories to the `MATLABPATH` (optional)

Path settings for ChirpLab are set in the file `ChirpPath.m`. It is possible, although not necessary, to edit the file and change the variable `CHIRPLABPATH` to the directory where ChirpLab is installed. Note that the directory separator character should be the same as what is normally used on your OS.

To permanently add the ChirpLab directories to your `MATLABPATH`, you can add the following commands to your Unix profile For csh-derived shells use:

```
setenv CHIRPLAB <root directory of ChirpLab>
setenv MATLABPATH ${MATLABPATH}:${CHIRPLAB}/mex/src/Networks:
${CHIRPLAB}/ChirpletTrans:${CHIRPLAB}/Data:${CHIRPLAB}/Data/ForDemos:
${CHIRPLAB}/Networks:${CHIRPLAB}/Utilities:${CHIRPLAB}/Inspiral
```

For Bourne-shell derived shells use:

```
CHIRPLAB=<root directory of ChirpLab>; export CHIRPLAB
MATLABPATH=${MATLABPATH}:${CHIRPLAB}/mex/src/Networks:
${CHIRPLAB}/ChirpletTrans:${CHIRPLAB}/Data:${CHIRPLAB}/Data/ForDemos:
${CHIRPLAB}/Networks:${CHIRPLAB}/Utilities:${CHIRPLAB}/Inspiral;
export MATLABPATH
```

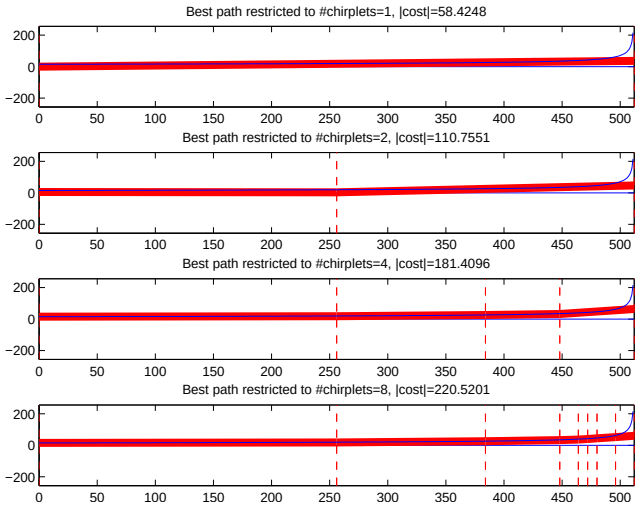

Figure 5: The second figure in the sample script `BinaryInspiralDemo.m`.

References

[1] E. J. Candès, P. Charlton, and H. Helgason. "Detecting highly oscillatory signals by chirplet path pursuit", `http://arxiv.org/gr-qc/0604017`.

[2] `http://www.chirplab.org`.

[3] A. Abramovici, W. E. Althouse, R. W. P. Drever, Y. Gursel, S. Kawamura, F. J. Raab, D. Shoemaker, L. Sievers, R. E. Spero, and K. S. Thorne. "LIGO – The Laser Interferometer Gravitational-Wave Observatory". *Science*, 256:325–333, April 1992.