User Manual for Particle Swarm Optimization-based Matched Filtering on Gravitational Wave Data

Raghav Girgaonkar*

Department of Physics and Astronomy
The University of Texas Rio Grande Valley

April 2023

Abstract

This is a user manual for codes and scripts to run chirp-time and mass Particle Swarm Optimization (PSO) based matched-filtering using a Restricted-2PN waveform on user specified files for single-detector gravitational wave (GW) data realizations and Power Spectral Density (PSD) estimates. The codes are written in the `MATLAB` programming language and have been tested in `MATLAB` R2022b.

1 Introduction

The optimized scripts to run Chirp-time or Mass PSO [1] depend upon certain scripts, specifically `crcbpso.m` and `s2rv.m` from the GitHub repository SDMBIGDAT19. Before running the script the following steps should be ensured,

1. Install/ Clone the GitHub repository SDMBIGDAT19.
2. If SDMBIGDAT19 is installed at `$SDMBIGDAT19`, add the following path in `MATLAB` , `addpath("$SDMBIGDAT19/CODES")`
3. Ensure that `MATLAB` 's Parallel Computing Toolbox is installed.

The following subsections give a brief background on the waveform templates used and the matched filtering.

1.1 Restricted 2-PN Waveform

The signal polarization waveforms used for injection in these codes are expressed as follows in the Fourier domain using the stationary phase approximation:

$$\tilde{h}_{+}(f) = \frac{\mathcal{A}_f}{r}\frac{1+\cos\mathrm{i}}{2} f^{-\frac{7}{6}} \exp[-i\Psi_f] \tag{1}$$

$$\tilde{h}_{\times}(f) = \frac{\mathcal{A}_f}{r}\cos\mathrm{i}\, f^{-\frac{7}{6}} \exp[-i(\Psi_f + \frac{\pi}{2})] \tag{2}$$

where Ψ_f is the phase term,

$$\Psi_f = 2\pi f t_c - \phi_c - \frac{\pi}{4} + 2\pi f(\tau_0 + \tau_1 - \tau_{1.5} + \tau_2) + \sum_{j=0}^{4} \alpha_j \left(\frac{f}{f_*}\right)^{\frac{-5+j}{3}} \tag{3}$$

*Email: raghav.girgaonkar01@utrgv.edu

[1] Codes can be found here on GitHub

with,

$$\tau_0 = \frac{5}{256\pi} f_*^{-1} \left(\frac{GM}{c^3}\pi f_*\right)^{-\frac{5}{3}} \eta^{-1}, \tag{4}$$

$$\tau_1 = \frac{5}{192\pi} f_*^{-1} \left(\frac{GM}{c^3}\pi f_*\right)^{-1} \eta^{-1} \left(\frac{743}{336} + \frac{11}{4}\eta\right), \tag{5}$$

$$\tau_{1.5} = \frac{1}{8} f_*^{-1} \left(\frac{GM}{c^3}\pi f_*\right)^{-\frac{2}{3}} \eta^{-1}, \tag{6}$$

$$\tau_2 = \frac{5}{128\pi} f_*^{-1} \left(\frac{GM}{c^3}\pi f_*\right)^{-\frac{1}{3}} \eta^{-1} \left(\frac{3058673}{1016064} + \frac{5429}{1008}\eta + \frac{617}{144}\eta^2\right) \tag{7}$$

where,

$$M = (m_1 + m_2),\ \mu = \frac{m_1 m_2}{M},\ \eta = \frac{\mu}{M} \tag{8}$$

and,

$$\alpha_0 = 2\pi f_* \frac{3\tau_0}{5}, \tag{9}$$

$$\alpha_1 = 0, \tag{10}$$

$$\alpha_2 = 2\pi f_* \tau_1, \tag{11}$$

$$\alpha_3 = -2\pi f_* \frac{3\tau_{1.5}}{2}, \tag{12}$$

$$\alpha_4 = 2\pi f_* 3\tau_2 \tag{13}$$

Mass-Space PSO searches for the optimal solution in the binary component mass search space $[m_1, m_2]$ whereas Tau Space PSO searches for the optimal solution in the chirp time space $[\tau_0, \tau_{1.5}]$. τ_0 and $\tau_{1.5}$ are also used to characterize the signal, these two parameters can be used to derive M and μ:

$$\mu = \frac{1}{16 f_*^2} \left(\frac{5}{4\pi^4 \tau_0 \tau_{1.5}^2}\right)^{\frac{1}{3}} \left(\frac{G}{c^3}\right)^{-1}, \tag{14}$$

$$M = \frac{5}{32 f_*} \left(\frac{\tau_{1.5}}{\pi^2 \tau_0}\right) \left(\frac{G}{c^3}\right)^{-1} \tag{15}$$

Here, G and c are the gravitational wave constant and speed of light respectively, t_c is the arrival time of the signal, f_* is the lower cutoff frequency and ϕ_c is the phase at coalescence.

1.2 Code Optimization

Details related to matched filtering and normlization of quadrature templates and signal injections is given in the `NoteonNormalization.pdf`. Reader is encouraged to read that document first. For optimization purposes,

1. The frequency magnitude term $A(f)$, the normalization factor N_f and the alpha terms in the waveforms (α_j where $j \in \{0, 1, 2, 3, 4\}$) are pre-calculated.

2. Since the quadrature templates $Q_0(f)$ & $Q_1(f)$ in fourier domain are related as $Q_1(f) = Q_0(f) \times \exp(-i\frac{\pi}{2})$, a vector corresponding to $\exp(-i\frac{\pi}{2})$ for all DTFT frequencies is pre-calculated.

3. Since the matched filtering and inner products between templates and the data realization are of the form of Equation 25, the frequency magnitude term $A(f)$ is multiplied (pointwise) to the fft of the data realization $\tilde{D}(f)$ and the PSD $S_n(f)$ is divided (pointwise) to create a variable `fftdataYbyPSD`

4. The waveform functions `gen2PNwaveform` and `gen2PNwaveform_tau` generate the phase term $\exp(-i\Psi(f))$ normalized by N_f as seen in Equation 25.

2 List of Functions

This section contains a brief description of the functions used, the code documentation can be found here.

1. `boundary_plot`

   Creates and returns the Chirp-time boundary plot as a figure. Used for plotting PSO-estimates in the search space. The assumed boundary in the mass-space is $[1, 30]M_{\odot}$.

2. `crcbgwpso_mass`

   Runs local-best PSO in the Mass Space on the data realization with user-specified parameters. It accepts a structure containing data and signal related parameters (`inParams`), a structure containing parameters for PSO (`psoparams`), the number of independent PSO runs (`nRuns`) and the sampling frequency of the data as input. Returns a structure containing PSO-estimated parameters for each independent run along with the best run estimates (`outResults`).

3. `crcbgwpso_tau`

   Runs local-best PSO in the Chirp-time Space on the data realization with user-specified parameters. It accepts a structure containing data and signal related parameters (`inParams`), a structure containing parameters for PSO (`psoparams`), the number of independent PSO runs (`nRuns`) and the sampling frequency of the data as input. Returns a structure containing PSO-estimated parameters for each independent run along with the best run estimates (`outResults`).

4. `createPSD`

   Script to create interpolated vectors of PSD for custom data lengths.

5. `gen2PNwaveform`

   Creates the normalized phase vector of the waveform in the Fourier domain. I.e the waveform (for example unit normalized quadrature template) can be written in the form

   $$Q_0(f) = N_f A(f) \exp\left(-i\Psi(f)\right)$$

   This function returns the DTFT vector of $N_f \exp\left(-i\Psi(f)\right)$. The phase term in this function is calculated using binary mass components m_1, m_2.

6. `gen2PNwaveform_tau`

   Creates the normalized phase vector of the waveform in the Fourier domain. I.e the waveform (for example unit normalized quadrature template) can be written in the form

   $$Q_0(f) = N_f A(f) \exp\left(-i\Psi(f)\right)$$

   This function returns the DTFT vector of $N_f \exp\left(-i\Psi(f)\right)$. The phase term in this function is calculated using chirp-time parameters $\tau_0, \tau_{1.5}$.

7. `innerprodpsd`

   This function calculates the inner product between a unit normalized quadrature template and the data realization. The inner product mentioned before can be written in Fourier domain as,

   $$< d(t)|q(t) >= \tilde{D}(f) N_f A(f) \frac{\exp\left(-i\Psi(f)\right)}{S_n(f)}$$

   Here,

   $$\texttt{fftdataXbyPSD} = \frac{A(f)\tilde{D}(f)}{S_n(f)}$$

and

$$\texttt{fftY} = N_f \exp\left(-i\Psi(f)\right)$$

8. `LIGOnoise`

   This function creates a colored noise realization using the design sensitivities specified in `iLIGOSensitivity.txt`. The design sensitivities are modified beyond [15,700] Hz. This function uses `statgaussnoisegen` to create the colored noise. Creation of the noise requires White Gaussian Noise (WGN) which can be created in runtime or specified from a .mat file containing pre-calculated WGN vectors for reproducibility of results.

9. `matchedfiltering`

   This function does FFT-based matched filtering and returns the matched-filtering timeseries. As seen before the expression for getting the matched-filtering timeseries is,

$$m(t) = \texttt{ifft}\left(\frac{\tilde{D}(f).*\tilde{Q}^*(f)}{S_n(f)}\right)$$

   using general expression for $Q(f)$, we have,

$$\texttt{fftdataXbyPSD} = \frac{A(f)\tilde{D}(f)}{S_n(f)}$$

   and

$$\texttt{fftY} = N_f \exp\left(-i\Psi(f)\right)$$

10. `mfgw_mass`

    This function generates a combined matched-filtering timeseries of $q_+(t)$ and $q_\times(t)$ with the data realization. Then the maximum value and the index of the maximum value of this timeseries is calculated and returned. These quantities correspond to the PSO-estimated time-of-arrival of the signal and the likelihood value (SNR) respectively. The quadrature templates in this function are calucated using the binary component masses m_1, m_2.

11. `mfgw_tau`

    This function generates a combined matched-filtering timeseries of $q_+(t)$ and $q_\times(t)$ with the data realization. Then the maximum value and the index of the maximum value of this timeseries is calculated and returned. These quantities correspond to the PSO-estimated time-of-arrival of the signal and the fitness/likelihood value respectively. The quadrature templates in this function are calucated using the chirp-time parameters $\tau_0, \tau_{1.5}$.

12. `preprocessing`

    This function calculates a few quantities before PSO-based matched filtering can be run. These quantities are constant and are used at every waveform generation call. They are the frequency magnitude vector $A(f) \propto f^{-\frac{7}{6}}$, the alpha terms α_j and the phase difference vector for quadrature templates.

13. `psofitfunc`

    This is a fitness function for Mass Space PSO. This uses `mfgw_mass` to calculate the fitness values for a specified location in the binary component mass search space. Also returns the index of the maximum value in the corresponding matched-filtering time series.

14. `psofitfunc_tau`

    This is a fitness function for Mass Space PSO. This uses `mfgw_tau` to calculate the fitness values for a specified location in the chirp-time search space. Also returns the index of the maximum value in the corresponding matched-filtering time series.

15. `rungwpso`

    Script to run PSO on a whitened HDF5 with a pre-computed transfer function. This will be explained in following sections.

16. `runpso`

    Script to run PSO on a whitened data segment supplied along with a pre-computed transfer function. This will be explained in following sections.

17. `launcherscript`

    Script to extract a segment of desired length, whitened it and pass it on to `runpso.m`.

18. `segdatacond`

    Script to whiten given data segment and creating a transfer function from the provided PSD.

19. `statgaussnoisegen`

    This function generates a colored noise realization using the design sensitivities given in the `iLIGOSensitivity.txt` file. It creates a transfer function based on these sensitivities and filters a vector of White Gaussian noise. For more details see the Wiener-Khinchin theorem.

20. `waveform`

    This function creates an unnormalized phase vector in the Fourier domain of the 2-PN waveform. I.e it creates $\exp(-i\Psi(f))$ for positive DTFT frequencies. This vector is creating using the binary component masses m_1, m_2.

21. `waveform_tau`

    This function creates an unnormalized phase vector in the Fourier domain of the 2-PN waveform. I.e it creates $\exp(-i\Psi(f))$ for positive DTFT frequencies. This vector is creating using the chirp-time parameters $\tau_0, \tau_{1.5}$.

3 `rungwpso.m`

`rungwpso.m` is one of the main scripts from which Chirp-time or Mass Space PSO can be run. It is the script that reads the data file and PSD specified by the user along with other relevant signal parameters. The signal, file and pso parameters can be changed/specified by editing the values in `signal.json`, `files.json` and `pso.json` respectively.

3.1 Running the script

Before running the script, the following steps must be ensured.

1. Change value of the sampling frequency `sampling_freq` and data segment length in seconds `T_sig_len` in the file `signal.json` accordingly. For example, typically for a LIGO HDF5 data file, the data segments are 4096 seconds long sampled at 4096 Hz, thus in that case `sampling_freq` = 4096 and `T_sig_len` = 4096.

2. If injecting custom CBC signal, injection parameters can also be changed in `signal.json`. Depending on the search space for PSO, this includes the mass parameters, the time of arrival (in seconds), the coalescence phase and the signal strength. In the case of injected signals, the user only needs to specify the **component masses**. In the case of chirp-time PSO, the chirp-time parameters τ_0 and $\tau_{1.5}$ will be calculated in script.

3. In `signal.json`, one can also set the search range for the PSO for mass space (`rmin` and `rmax`) or for chirp-time space (`rmin_tau` and `rmax_tau`). By default these values are [1,30] for mass space and [0,90] for τ_0 and [0,2] for $\tau_{1.5}$.

4. The high frequency and low frequency cutoffs for the waveforms can also be specified (`fmin` and `fmax`) in `signal.json`.

5. In `files.json` specify the **full paths** of the data file (`datafile`) and the output struct that stores the PSO estimates and results of each independent run (`output_struct_location`) and plot names `bestfitplot` `psoresultplot` and `bestlocplot`.

6. If using custom PSD file provide corresponding file path (`psdfile`) in `files.json`. If creating colored noise from custom noise realizations, specify full path for the noise file (`noisefile`).

7. Change PSO type type in `pso.json` to either `tau` or `mass` for Chirp-time and Mass PSO respectively.

8. Change the number of PSO iterations (`maxSteps`) and/or number of independent PSO runs (`nruns`) in `pso.json`.

9. The information present in the files `signal.json`, `files.json` and `pso.json` are fed in to the script through a master json file called `allparamfiles.json`. This file must have the full paths to the corresponding json files if one has them in different directories.

Once the above steps are ensured, the script can be run in the `MATLAB` command line by,

```
rungwpso allparamfiles.json
```

3.2 Script Components

The sequence of steps in the script are as follows,

1. Read the signal, file and pso parameters from the json files. From these values infer parameters such as the total number of samples, the DTFT positive frequency vector and in the case of chirp-time PSO, the τ_0 and $\tau_{1.5}$ values from the user specified mass components.

```
 N = floor(num*T_sig_len*Fs);

 %% Positive Frequency Vector
datalen = N/Fs;
fpos = (0:floor(N/2))*(1/datalen);

%% Tau coeffs as phase parameters
if pso.type == "tau"
    m1 = m1*Msolar;
    m2 = m2*Msolar;
    M = m1 + m2;
    u = m1*m2/(m1 + m2);
    n = u/M;
    tau0 = (5/(256*pi))*(1/fmin)*((G*M*pi*fmin/c^3)^(-5/3))*(1/n);
    tau1p5 = (1/8)*(1/fmin)*((G*M*pi*fmin/c^3)^(-2/3))*(1/n);
    type = 1;

```

2. Pre-calculate quantities such as the frequency magnitude vector $A(f) \propto f^{-\frac{7}{6}}$, the alpha terms α_j and the phase difference vector for quadrature templates using the `preprocessing` function.

3. Read the input data file and the PSD file. This can have two subcases,

   (a) Both the data and the PSD are in a single HDF5 file. In this case the convention followed is that the data be already whitened once i.e it has already been filtered with a transfer function corresponding to $\frac{1}{\sqrt{S_n(f)}}$. and the transfer function is specified in the same file under the `strain` group as `condTF`. For example,

```
dataY = h5read(files.datafile,'/strain/Strain')';
TF = h5read(files.datafile,'/strain/condTF')';

```

   A row vector convention is maintained throughout the script.

   (b) The data and the PSD are specified in two different mat files. In this case the convention follows that PSD vector for the positive DTFT frequencies be specified and the data be un-whitened. For example,

```
S = load(files.datafile);
dataY = S.dataY;
%% Load PSD
E = load(files.psdfile);

```

4. Create the `fftdataYbyPSD` vector for both cases. This creates the vector corresponding to

$$\texttt{fftdataXbyPSD} = \frac{A(f)\tilde{D}(f)}{S_n(f)}$$

   and also generate the general normalization factor N_f given by Equation 21. For example in the HDF5 input file case,

```
%Create entire Transfer Function vector
 TFtotal = [TF, TF((kNyq-negFStrt):-1:2)];
 AbysqrtPSD = A.*TFtotal;

 %% Create General Normalization Factor
   % Scalar factor of 1/N is due to Parseval's theorem
   dataLen = N;
   innProd = (1/dataLen)*(AbysqrtPSD)*AbysqrtPSD';
   genNormfacSqr = real(innProd);
   genNormfac = 1/sqrt(genNormfacSqr);

   %% Data Products
   fftdataY = fft(dataY);
   fftdataY = fftdataY.*A;%Pre-multiply frequency magnitude vector A for
optimization
   fftdataYbyPSD = fftdataY.*TFtotal;

```

5. Generate parameter structure, this contains all relavant signal, file and pso parameters to run matched-filtering, generate waveforms and output plots. For example,

```
   %% Input Parameters:
   inParams = struct('dataX', dataX,...
                     'fpos', fpos,...
                     'dataY', dataY,...
                     'fftdataYbyPSD', fftdataYbyPSD,...
                     'frange', [fmin,fmax],...
                     'datalen',datalen,...,
                     'initial_phase', initial_phase,...
                     'N', N,...
                     'A', A,...
                     'phaseDiff', phaseDiff,...
                     'normfac', genNormfac,...
                     'avec', avec,...
```

```
                    'T_sig', T_sig,...
                           'rmin',rmin,...
                           'rmax',rmax,...
                           'Fs',Fs);

```

6. Run PSO-based matched-filtering based on user-specified search space. In the case of signal injection, the fitness value at the true parameter location is calculated and stored.

```
        maxSteps = pso.maxSteps;
        if type
            original_fitVal = -1*mfgw_tau([tau0, tau1p5], inParams);
            outStruct = crcbgwpso_tau(inParams,struct('maxSteps',maxSteps),nRuns,Fs);
            bestFitVal = -1*outStruct.bestFitness;
        else
            original_fitVal = -1*mfgw_mass([m1, m2], inParams);
            outStruct = crcbgwpso_mass(inParams,struct('maxSteps',maxSteps),nRuns,Fs);
            bestFitVal = -1*outStruct.bestFitness;
end

```

7. Generate output plots and display estimated parameters.

Depending on the use case, one can change the script to suit the input/output and processing conventions.

3.3 Output

This script can produce the following figures,

1. `bestfitplot`

   This is a figure that plots the evolution of the best fitness values of each independent run over the specified iterations. An example of this is shown as follows,

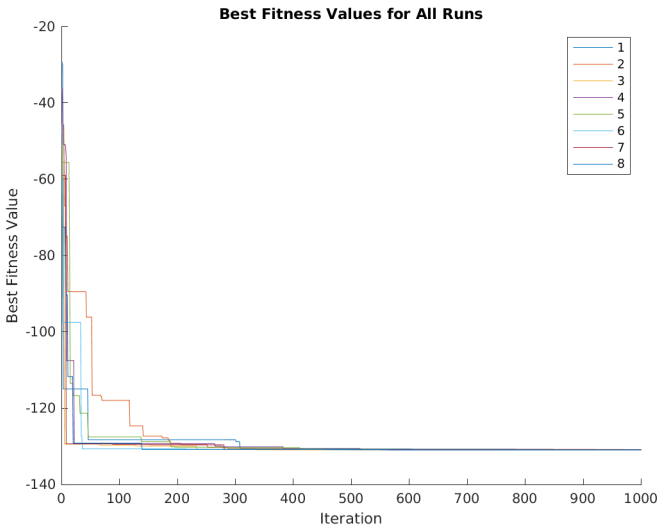

Figure 1: Best Fitness Value evolution for each independent run for Mass Space PSO

2. `psoresultplot`

This is a figure that has an overlay plot of the original data realization and the signal corresponding the the PSO-estimated parameters. In the case where a custom CBC signal is injected, it also plots the original injected signal.

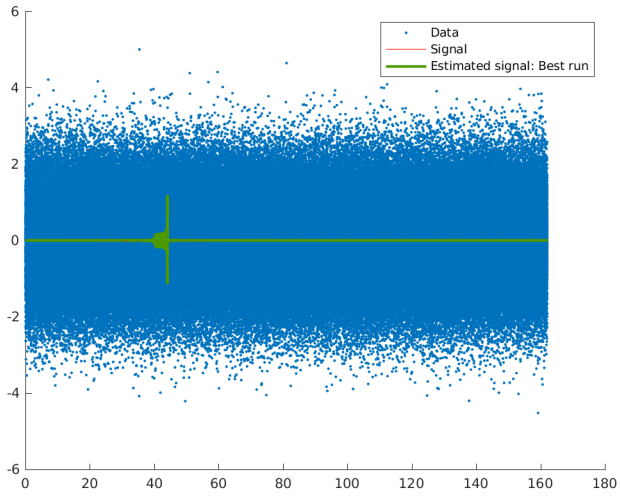

Figure 2: The PSO-estimated signal (green), the original injected signal (red) and the original data realization (blue) for a Tau Space PSO run. In this case the original injected signal and the estimated signal are very similiar and lie almost on top of each other.

3. `bestlocplot`

This is a figure that plots the evolution of the best run location found by the independent runs over the PSO search space. In the case of a custom CBC injection, the original parameter location is also plotted. This plot is useful in the case of Tau Space PSO as it shows the physical search space boundary compared to the PSO run locations. For example,

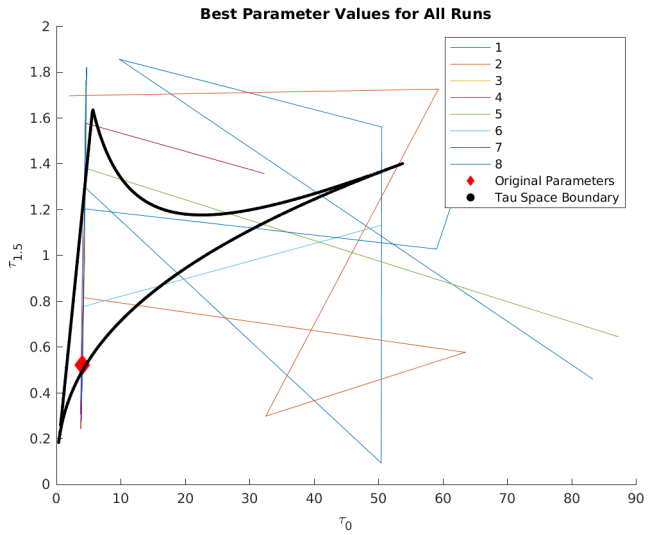

Figure 3: Best run location evolution in the PSO search Space for each independent PSO run. The boundary (black) is generated by the `boundary_plot` function. The location of the injected parameters is plotted as the red dot.

Along with this the script also generates command line output displaying the PSO-estimated parameters in comparison to the original, injected parameters. One such sample output can look like,

```
        Original parameters: tau0= 29.6373; tau1p5= 1.1045; m1= 2; m2= 2; A = 8; phi = 0;
    t_a = 57; FitVal = 314820591.1893
Estimated parameters: tau0=29.6225; tau1p5=1.0967; m1= 1.9868-0.21416i; m2= 1.9868+0.21416i;
     A = 8.3707; phi = 0.19236; t_a = 57.0112; FitVal = 321155053.9049

```

In the case where custom signals are not injected, one can comment out the lines that display the original parameters.

4 `runpso.m`

`runpso.m` is a general script that can run PSO on a specified data segment of any size. It takes as inputs, the whitened segment vector, the transfer function and other components such as the parameter json file and the segment length etc. Once generally would create a wrapper script for this function which would first load the data segment vector, whiten it and pass it through to this function. One such example is `launcherscript.m`. `runpso.m` is a more general script and should be used along with a companion wrapper function.

4.1 Script Components

The main script components are explained in this section.

```
%% Pre-processing
[A,avec, phaseDiff] = preprocessing(fmin,fmax, fpos, datalen, N);

dataY = whtnddata;

AbysqrtPSD = A.*TFtotal;


```

This part of the script creates common vectors needed for waveform generation and matched filtering once before PSO is running rather than generate them at every iteration. The vectors generated are the frequency magnitude vector `A`, $(f^{-7/6})$, the phase difference vector `phaseDiff` for create $h_\times$ from h_+ and `avec`, which creates the α terms for the 2PN waveforms.

```
        %% Create General Normalization Factor
% Scalar factor of 1/N (check NoteonNormalizaton.pdf for more details)
dataLen = N;
innProd = (1/dataLen)*(AbysqrtPSD)*AbysqrtPSD';
genNormfacSqr = real(innProd);
genNormfac = 1/sqrt(genNormfacSqr);

%% Data Products
fftdataY = fft(dataY);
fftdataY = fftdataY.*A;%Pre-multiply frequency magnitude vector A for optimization

%% Get FFT of data by total PSD
fftdataYbyPSD = fftdataY.*TFtotal;

```

This part of the script creates the normalization factor as explained in Section 3.1 and creates the `fftdataYbyPSD` vector.

```
        %% Input Parameters:
inParams = struct('dataX', dataX,...
                  'fpos', fpos,...
                  'dataY', dataY,...
                  'fftdataYbyPSD', fftdataYbyPSD,...
                  'frange', [fmin,fmax],...
                  'datalen',datalen,...,
                  'initial_phase', initial_phase,...
                  'N', N,...
                  'A', A,...
                  'phaseDiff', phaseDiff,...
                  'normfac', genNormfac,...
                  'avec', avec,...
              'T_sig', T_sig,...
                  'rmin',rmin,...
                  'rmax',rmax,...
                  'Fs',Fs);

```

Create input parameter structure.

```
        maxSteps = pso.maxSteps;
if type
    if rmin(1) >= 0 && rmin(2) >= 0
      original_fitVal = -1*mfgw_tau([tau0, tau1p5], inParams);
      outStruct = crcbgwpso_tau(inParams,struct('maxSteps',maxSteps),nRuns,Fs);
    else
  original_fitVal = -1*mfgw_tau_negative([tau0, tau1p5], inParams);
      outStruct = crcbgwpso_tau_negative(inParams,struct('maxSteps',maxSteps),nRuns,Fs)
    ;
    end
    bestFitVal = -1*outStruct.bestFitness;
else
    original_fitVal = -1*mfgw_mass([m1, m2], inParams);
    outStruct = crcbgwpso_mass(inParams,struct('maxSteps',maxSteps),nRuns,Fs);
    bestFitVal = -1*outStruct.bestFitness;
end


```

Run PSO, with provisions to run negative chirp-time space PSO.

```
        candidates = [segnum, t0, t1p5, real(est_m1), imag(est_m1), real(est_m2), imag
    (est_m2), est_ta, est_SNR];
%candidates = [segnum, est_m1, est_m2, est_ta, est_SNR];
fileID = fopen(files.candidatefile,'a');
fprintf(fileID,'%d\t%f\t%f\t%f + %fi\t\t%f + %fi\t\t%f\t\t%f\n',candidates);
%fprintf(fileID,'%d\t%f\t%f\t%f + %fi\t\t%f + %fi\t\t%f\t\t%f\t%f\n',candidates);
%fprintf(fileID,'%d\t%f\t%f\t%f\t%f\n',candidates);
fclose(fileID);

```

In the case the Chirp-time PSO is run, this part of the code outputs the estimated parameters to an output text file that is specfied in the `files.json` file.

5 Running the Script on LoneStar6

Running PSO-based matched-filtering on large data files may not be feasible on local machines due to the significant computational costs. This script can be run on the Texas Advanced Computing Center's supercomputer LoneStar6 (Ls6) [2] Ls6 makes use of Simple Linux Utility for Resource Management (SLURM) for batching and scheduling jobs. For this, a *slurm* file must be created.

5.1 SLURM File

An example *slurm* file is present in the `examples` folder of the GitHub repository. This file has the following structure.

```
      #!/bin/bash
#----------------------------------------------------
# Sample Slurm job script

#SBATCH -J sample           # Job name
#SBATCH -o /path/sample.o%j       # Name and Path of stdout output file
#SBATCH -e /path/sample.e%j       # Name and Path of stderr error file
#SBATCH -p normal          # Queue (partition) name
#SBATCH -N 1               # Total # of nodes (must be 1 for serial)
#SBATCH -n 1               # Total # of mpi tasks (should be 1 for serial)
#SBATCH -t 10:00:00        # Run time (hh:mm:ss)
#SBATCH --mail-type=all    # Send email at begin and end of job
#SBATCH --mail-user=username@utrgv.edu

module load matlab
matlab -batch  "addpath($SDMBIGDAT19/CODES); cd /working_dir; rungwpso  /path/to/jsonfiles/
    allparamfiles.json"

```

The file has the following components:

1. `#SBATCH -J sample`

   This sets the job name to "sample". This can be changed according to the job being run.

2. `#SBATCH -o /path/sample.o%j` and `#SBATCH -e /path/sample.e%j`

   This sets the name and location of the output and error files respectively. The `%j` denotes the job id assigned by the system to distinguish between two jobs with the same name. These names can also be changed if needed, however keeping the `%j` at the end of the name is recommended.

3. `#SBATCH -p normal # Queue (partition) name`

   This specifies the queue to which the job will be submitted to. Depending on job size and time needed, one can choose the appropriate queue. More information on queues on Ls6 can be found here.

[2] For more details see here.

4. `#SBATCH -N 1 # Total # of nodes (must be 1 for serial)`

   This is the total number of nodes that the job needs. Ls6 has 550 compute nodes with 128 cores on each node. Typically, for a PSO-run on a single data file 1 node is enough, for 8 independent PSO runs the job will use 8 cores. If multiple parallel jobs are run for example with `launcher` [3], this number should be set appropriately.

5. `#SBATCH -n 1 # Total # of mpi tasks`

   This specifies the total number of mpi tasks needed for the job. For a single data file PSO-run, this can be set to one. If multiple parallel jobs are to be run on a single node, this number should be changed to the number of jobs to be run on that node. [4]

6. `#SBATCH -t 10:00:00`

   This sets the time limit for the job in `hh:mm:ss`. For a typical LIGO HDF5 file containing 4096 seconds of data sampled at 4kHz, a PSO-run of 500 iterations takes 7-10 hours. To account for any delays in the job, this number should be greater than the estimated time needed for the job(s).

7. `#SBATCH --mail-type=all` and `#SBATCH --mail-user=username@utrgv.edu`

   `--mail-type=all` specifies that the user will get mails alerting the start and end of the job. The users email id should be given after `--mail-user=`

```
1
2           module load matlab
            matlab -batch  "addpath($SDMBIGDAT19/CODES); cd /working_dir; rungwpso  /
    path/to/jsonfiles/allparamfiles.json"
3
```

   The first line loads `MATLAB` into the job environment. On line 2, the path corresponding to `$SDMBIGDAT19/CODES` (on Ls6) is loaded. Then the working directory (where `rungwpso.m`) is present is loaded and the script is ran.

The *slurm* job can be submitted through the command line as follows,

```
1
2       $> sbatch sample.slurm
```

5.2 Useful Slurm Commands

1. `squeue -u <username>` Shows the status of all running/pending jobs associated with specified username.
2. `scancel -u <username>` Cancels all jobs associated with specified username.
3. `scancel <jobid>` Cancels a specific job based on specified job id.

6 Example

In the `examples` folder there is a `rungwpso_bns.m` script to run PSO-based matched-filtering on a cleaned, whitened and bandpassed HDF5 data file from LIGO Livingston `L-L1_LOSC_CLN_4_V1-1187007040-2048_dtrndWhtnBndpss.hdf5`. This is the file that has the GW170817 binary neutron star event towards the end of the file. The corresponding json files and the slurm batch file to run the script are also present in the directory. The steps to run this script are,

[3] `launcher`

[4] For example, if one wishes to run 50 independent PSO-based matched-filtering jobs (say on 50 different data files and with 8 independent PSO runs per job) at once using `launcher`, then N = 5 and n = 10. This is because each job will use 8 cores per node and each node has 128 cores. Thus the maximum number of jobs one can run on a node is 16. However, for uniform distribution of jobs this is set to 10.

1. Clone the repository,

```
git clone https://github.com/RaghavGirgaonkar/Accelerated-Network-Analysis.git
```

2. One can make changes to the PSO parameters as explained in Section 3.1.

3. If SDMBIGDAT19 is installed at `$SDMBIGDAT19`, add the following path in `MATLAB` ,

```
addpath("\$SDMBIGDAT19/CODES")
```

4. **To run on local machine**: simply `cd` into `examples` directory and run

```
rungwpso_bns allparamfiles.json
```

5. **To run on Ls6**, follow steps 1 and 2 on Ls6. Edit the `BNS.slurm` file as needed according to Section 5.1 and submit the job as

```
$> sbatch BNS.slurm
```

After 500 PSO iterations, one can expect output similar to as follows,

```
Estimated parameters: tau0=55.1758; tau1p5=1.6327; m1= 0.85849; m2= 2.3175; A = 25.7199; phi = -0.17383; t_a = 1787.2678; FitVal = 668.4991
```