# FAST-PT User Manual

Joseph E. McEwen
mcewen.24@osu.edu
Xiao Fang
fang.307@osu.edu

Our paper (arXiv:1603.04826 ) describes the FAST-PT algorithm and implementation. This paper should be cited when using FAST-PT in your research.

## 1 Versions

- Version 1.0 was released on March 15 2016.
- Version 1.1 was finalized on March 28 2016. This version included the following updates:
  - updated to be Python 3 and Python 2 compatible;
  - a bug was fixed in the power spectrum windowing routine. The previous version made power spectrum windowing obsolete;
  - the labeling within the plot example in FASTPT.py was changed to correctly indicated the output was $P_{22}(k) + P_{13}(k)$.

## 2 Software Requirements

FAST-PT makes use of NUMPY and SCIPY libraries. It is advised that you have newer versions of numpy and scipy. We have found that older versions of numpy and scipy can be problematic. To run the scripts that reproduce our plots you will also need a current version of matplotlib.

Our code was originally developed with Python version 2.7.10, NUMPY 1.8.2, and SCIPY 0.15.1. It is possible that newer version of Python and NUMPY and SCIPY can result in errors, due to software changes.

### 2.1 Python 3 issues

We have found a few issues related to Python 3 and corrected for in version 1.1 of our code.

- Python 3 does not allow the mixing of tabs and spaces. The best way we have found to handle this issue is to look for indentations errors by running python -tt <filename.py>. This will locate indentation errors and you can then correct. We have tried to locate all the indentation mixing, but a few may still be left.

- Python has changed the print statements from print "stuff" to print("stuff"). We have tried to change all the print statement to conform to Python 3 standards. If any Python 2 print statements have been left, this is an easy fix for the user.

- Division in Python 3 is different than in Python 2.X. To make compatible with Python 3 we have added the following to the top of the FASTPT.py script (it must be at the first line of the script).

```
from __future__ import division
```

  So, for floor division we use the "//" symbol and "/" for regular division.

# 3 Getting Started

Probably the first thing to do is to see if you can run `FASTPT.py`. The main file `FASTPT.py` contains a small script ( under the line [ if `__name__=="__main__"`: ] )to plot the 1-loop correction to the power spectrum. This script should serve as a template. A typical code snippet would look something like this:

```
import FASTPT

data=np.loadtxt('Pk_Planck15.dat')
# declare k and the power spectrum
k=d[:,0]; P=d[:,1]

# set the parameters for the power spectrum window and
# Fourier coefficient window
P_window=np.array([.2,.2]) ''' the windowing for the power spectrum is generally
    not needed, but included in this script for instructive purposes'''
C_window=.65

# bias parameter and padding length
nu=-2; n_pad=800

# initialize the FASTPT class
fastpt=FASTPT(k,nu,n_pad=n_pad)

# get the one-loop power spectrum
P_spt=fastpt.one_loop(P,P_window=P_window,C_window=C_window)
```

```
# update the power spectrum
P=P+P_spt
```

The windowing parameter `P_window=[0.2,0.2]` means that you start tapering the power spectrum at $\log k_{\rm min} + .2$ and $\log k_{\rm max} - .2$. The window parameters `C_window = .65` means that you begin tapering the Fourier coefficients $c_m$ at $|m| \geq 0.65 \times N/2$ (it will round to the nearest integer). One should chose windowing parameters wisely. You don't want to window away the majority of the function. The figure below illustrates the effect of applying the window function to the linear power spectrum and using the window function as a filter applied to the Fourier coefficients. In the left panel, one can see that the edges of the power spectrum are smoothly tapered to zero. The right panel displays a damping of the highest frequency Fourier modes.

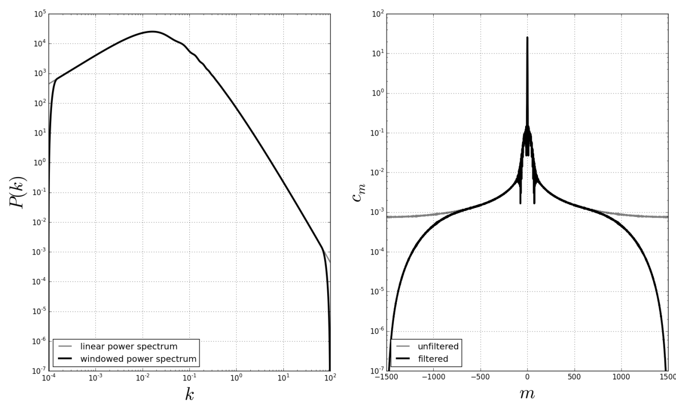


Figure 1: Smoothed power spectrum and filtered Fourier coefficients. Left panel compares the linear power spectrum to a windowed power spectrum. Right panel compares unfiltered Fourier coefficients to those that are filtered.

Zero padding should be $\geq 2$ to ensure that the $k$ of interest is $\geq 2k_{\rm min}$. The output to `fastpt.one_loop` is equivalent to $P_{22}(k) + P_{13}(k)$ (in the above code snippet this is denoted as `P_spt`).

# 4 Files

## 4.1 `FASTPT.py`

This class takes two required inputs; an array $k$ (the wave vector) and a float $\nu$ (the biasing power), to initialize. An hard coded input is `param_mat`, which is the set of $\{\alpha, \beta, l\}$, with a fourth column set to 0 or 1. This last column corresponds to switching routines between calculating $J_{\alpha\beta l}(k)$ and $J_{\alpha\beta l,\mathrm{reg}}(k)$. The hard coded values for `param_mat` corresponds to those for our $P_{22,\mathrm{reg}}(k)$ run. A user has the option to input a `param_mat` array of their liking. Additional options include `n_pad`, the number of zeros to pad and an option to turn on verbose settings.

Upon initialization `FASTPT.py` will calculate all objects that depend only on the grid size (i.e. the number of points in the array $k$). These include all gamma function type evaluations and associated pre-factors. Putting grid specific calculations at initialization speeds up the recurring run time by avoiding repeated calculations.

Contained in `FASTPT.py` are the following functions:

- `J_k`: this is the workhorse function of FASTPT, it computes $J_{\alpha\beta l}(k)$ or $J_{\alpha\beta l,\mathrm{reg}}(k)$ (depending on user specifications). The required input is the power spectrum. Optional inputs are the window functions parameters (`P_window` and `C_window`) which are used to window the input power spectrum and/or the Fourier coefficients. `J_k` uses the `param_mat` file and `gamma` function evaluations that were set up upon initialization of the `FASTPT` class;

- `P_22`: this function adds up each Legendre component from the from `J_k` to construct $P_{22,\mathrm{reg}}(k)$;

- `one_loop`: this is the function most likely to be used. It requires the input power spectrum and has optional arguments for windowing parameters. It calls `P_22` and `P_13_reg` (which is located in the file `matter_power_spt.py`). The output for `one_loop` is $P_{22}(k) + P_{13}(k)$.

### 4.2 `gamma_funcs.py`

This module contains three functions that we use for our gamma functions type objects. They are:

- `log_gamma(z)`

- `g_m_vals(mu,q)`

- `gamsn(z)`.

#### 4.2.1 `log_gamma(z)`

This function calculates $\ln\Gamma(z)$ by calling `gamma` function in `scipy` and `log` function in `numpy`. This function returns the real part and the imaginary part together.

### 4.2.2 `g_m_vals(mu,q)`

This function calculates the $g_m$ values in the FAST-PT paper (Eqn.(B.2)).

$$g_m(\mu, q) = \frac{\Gamma(\alpha_+)}{\Gamma(\alpha_-)} = \frac{\Gamma\left(\frac{\mu+1+q}{2}\right)}{\Gamma\left(\frac{\mu+1-q}{2}\right)} \ , \tag{1}$$

where $\mu + 1$ is real and $q$ has an imaginary part that could be very large in magnitude, $|\Im(q)| > 200$.

The direct calculation of it works well for small $\Im(q)$, $|\Im(q)| \leq \text{cut} = 200$, and for $\mu + 1 - q \neq 0$. These $q$'s are called `q_good` in the code.

For $\mu + 1 - q \neq 0$, the gamma function in the denominator blows up, so that $g_m$ approaches zero.

For large $|\Im q|$, the `gamma` function from `scipy` does not work well. We therefore derive an asymptotic formula for $g_m$ at $\Im q > \text{cut}$. We choose cut $= 200$ in our code. The asymptotic formula derived from the Stirling formula (Eqn.(B.5) in FAST-PT paper) is given by:

$$\begin{aligned} \ln g_m(\mu, q) &= \ln(\Gamma(\alpha_+)) - \ln(\Gamma(\alpha_-)) \\ &\simeq (\alpha_+ - 0.5)\ln(\alpha_+) - (\alpha_- - 0.5)\ln(\alpha_-) - q + \frac{1}{12}\left(\frac{1}{\alpha_+} - \frac{1}{\alpha_-}\right) + \frac{1}{360}\left(\frac{1}{\alpha_-^3} - \frac{1}{\alpha_+^3}\right) \end{aligned} \tag{2}$$

### 4.2.3 `gamsn(z)`

This function calculates $\Gamma(z)\sin\left(\frac{\pi}{2}z\right)$ using formula

$$\Gamma(z)\sin\left(\frac{\pi}{2}z\right) = \frac{\sqrt{\pi}}{2}2^z\frac{\Gamma\left(\frac{1}{2}+\frac{z}{2}\right)}{\Gamma\left(1-\frac{z}{2}\right)} = \frac{\sqrt{\pi}}{2}2^z g_m(0.5, z-0.5) \ . \tag{3}$$

## 4.3 `fastpt_extr.py`

This module contains a set of "extra" routines that are used within FAST-PT. These include our window functions for power spectrum and Fourier coefficients. The routines to pad the power spectrum with zeros (to the left or right of the array). The routine to calculate the $n_{\text{eff}} = \frac{d\ln P}{d\ln k}$.

## 4.4 `matter_power_spt.py`

This file contains three functions. Two of them, are left overs form a previous version of the code and are now implemented within the `FASTPT.py` class (`P_22_reg` and `one_loop`). The function `P_13_reg` calculates the $P_{13,\text{reg}}(k)$ by convolution as shown in the paper. It takes two inputs, $k$ and the power spectrum $P$. Usually the input power spectrum is the inverse Fourier transform of $c_m$ (this is procedure hard coded in the `one_loop` functions).

### 4.5 `J_k.py`

This is an older version of the code. It is now fully contained in the class `FASTPT.py`, but could be used on its own if a user wanted to.

### 4.6 `RG_RK4.py`

This function calculates the renormalization group results by integrating Eq. 3.1 in the paper using a 4th order Runge-Kutta integrator. The inputs to this routine are the output file name, the vector $k$, the power spectrum $P$, the integration step size $\Delta\lambda$, the maximum $\Lambda$ to integrate to, the number of zeros to pad with, the parameters for the power spectrum window, and the parameter for the Fourier coefficient window. This integration routine is not well suited for $k_{\max} > 1$. The output file saved is an array with the following structure

$$
\begin{bmatrix}
0, & \text{the wave vector } k \\
0, & \text{the linear power spectrum} \\
0, & \text{the one-loop power spectrum, i.e. } P_{\text{lin}} + P_{22} + P_{13} \\
\lambda(i=0), & \text{power spectrum at first lambda step} \\
\vdots, & \vdots \\
\lambda(i=N), & \text{power spectrum at last lambda step}
\end{bmatrix} . \tag{4}
$$

### 4.7 `RG_STS.py`

This function calculates the renormalization group results by integrating Eq. 3.1 in the paper using a super time step method (see the appendix to the paper). The inputs are the same as `RG_RK4.py`. The parameters for super time stepping are $\mu = .1$, $\Delta\lambda_{CFL} = .001$ and the number of stages is set to 10. These can all be changed by the user and are found at the begging of `RG_STS.py`, right before the function `RG_STS`. This integration routine is well suited for $k_{\max} > 1$. The output is in the same format as `RG_RK4.py`.

### 4.8 `RG_RK_filt.py`

This function calculates the renormalization group results by integrating Eq. 3.1 in the paper using a 4th order Runge-Kutta integrator with a digital filter applied to each stage. This is an old routine. It was a method developed to maintain stability. It is still useful and can be used for $k_{\max} > 1$, particularly for sparsely sampled power spectra. The inputs to this routine are the output file name, the vector $k$, the power spectrum $P$, the integration step size $\Delta\lambda$, the maximum $\lambda$ to integrate to, the number of zeros to pad with, the parameters for the power spectrum window, and the parameter for the Fourier coefficient window. The output is in the same format as `RG_RK4.py`.

### 4.9 xxx_example.ini

These are ini files that our RG integrators use.

### 4.10 `RG_ani.py`

Use the file `RG_ani.py` to makes animations of the RG output. If you have an output with a lot of frames, you should downsample in `RG_ani.py`, or else it will take a long time to run. The inputs are the same as the output for `RG_RK4.py`.