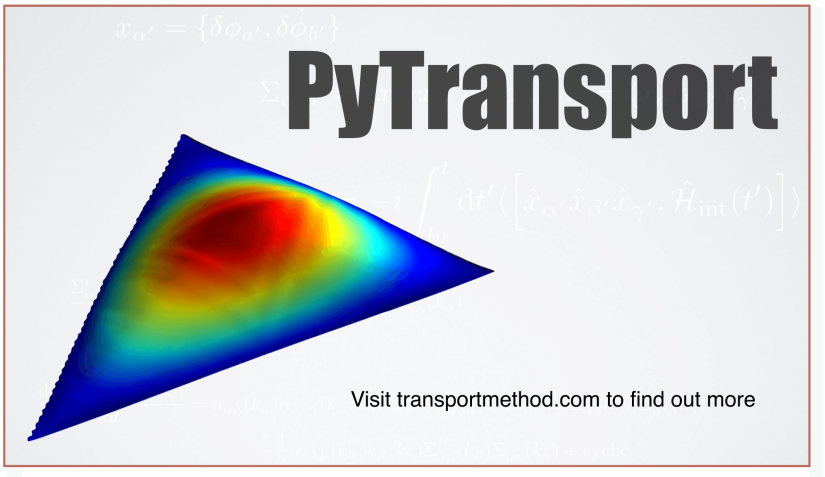
PyTransport: A Python package for the calculation of inflationary correlation functions

David J. Mulryne
Astronomy Unit, School of Physics and Astronomy,
Queen Mary University of London, Mile End Road, London, E1 4NS, UK[*]
(Dated: September 5, 2016)

PyTransport constitutes a straightforward code written in C++ together with Python scripts which automatically edit, compile and run the C++ code as a Python module. It has been written for Unix-like systems (OS X and Linux). Primarily the module employs the transport approach to inflationary cosmology to calculate the tree-level power-spectrum and bispectrum of user specified models of multi-field inflation, accounting for all sub and super-horizon effects. The transport method we utilise means only coupled differential equations need to be solved, and the implementation presented here combines the speed of C++ with the functionality and convenience of Python. At present the code is restricted to canonical models. This document details the code and illustrates how to use it with a worked example.



I. INTRODUCTION

PyTransport is distributed under a GNU GPL licence. The most recent version can be obtained by visiting transportmethod.com. If you use PyTransport you are kindly asked to cite Ref. [1] as well as the archive version of this user guide in any resulting works.

The main purpose of this document is to teach those interested how to use, and if so desired adapt, the PyTransport package. The philosophy behind the implementation is simplicity and ease of use. Python was selected as the language though which to interact with the code because it enables rapid scripting and provides a flexible and powerful platform. In particular, it has many readily available tools and packages for analysis and visualisation, and for tasks such as parallelisation (using for example Mpi4Py). As an interpreted language, however, Python can be slow for some tasks. This is circumvented here by using C++ code, which is compiled into a Python module, to perform numerically intensive tasks with the result that the speed of the package is nearly indistinguishable from pure C++. The C++ code itself is kept as simple and clean as possible and can therefore easily be edited if required. PyTransport has been developed on OS X using Python 2.7. We have also performed limited testing on Linux systems, and attempted to ensure compatibility with versions of Python 3. It seems likely it could also be adapted to Windows systems, but this has not yet been attempted.

[*]d.mulryne@qmul.ac.uk

The code is intended to be a reusable resource for inflationary cosmology. It enables users to quickly create a complied Python module(s) for any given model(s) of multi-field inflation. The primary function of the complied module is to calculate the power-spectrum and bi-spectrum of inflationary perturbations produced by multi-field inflation. To this end, the module contains a number functions that can be called from Python and that perform tasks such as calculating the background evolution of the cosmology, as well as the evolution of the two and three point functions. We also provide a number of further functions written in Python that perform common tasks such as calculating the power spectrum or bispectrum over a range of scales by utilising the compiled module. The true power of the approach, however, is that users can rapidly write their own scripts, or adapt ours, to suit their own needs.

The transport approach to inflationary perturbation theory that the code employs can be seen as the differential version of the integral expressions of the In-In formalism. It is helpful numerically because it provides a set of ordinary differential equations for the correlation functions of inflationary perturbations. The code solves these equations from deep inside the horizon until some desired time after horizon crossing using a standard variable step size ordinary differential equation (ODE) routine with error control. Such off the shelf routines are extremely well tested, and provide an easy way to change the required accuracy. This is helpful in order to check convergence of the numerical solutions, or to respond to needs of models with very fine features. Details of the transport method itself that the code is based on can be found in the recent paper [1]. We highly recommend reading this guide in combination with that paper.

In this guide, we first we give some brief background and motivation for the code, much more can be found in Ref. [1], before giving an overview of its structure and how it can be set up. In the appendices we give some more detail about the structure of the underlying C++ code, give full details of all the functions the complied module provides, and all the functions provided by Python scripts which accomplish common tasks. The best way to learn how to use the package, however, is by example. We present an extended example below spread between the "Getting going" and "Examples" sections, complete with screen shots of the code in use. Other examples that come with the distribution are discussed in the "Examples" section. Throughout, familiarity with Python and to some extent C++ is assumed, though in reality users can just probably get a long way by looking at the examples and modifying to their needs.

Finally, we would also like to refer readers to the complementary package developed by David Seery in tandem with the work in Ref. [1] and with PyTransport: CppTransport [2]. This is a platform for inflationary cosmology developed fully in C++. In comparison with PyTransport it has more external dependancies (in the sense that the dependancies of PyTransport are mainly Python modules), but provides more sophisticated parallelisation and data management capabilities. In limited testing it is also found to be marginally faster. For users with modest aims in terms of CPU hours and data generation, however, it is likely to have a higher overhead in getting started, but may well be beneficial for intensive users. PyTransport is intended to be more lightweight with users encouraged to utilise the power of Python in combination with PyTransport to achieve their specific aims and data management needs.

II. BACKGROUND

Calculations of the correlation functions of perturbations produced by inflation are now extremely mature. In the single field context, the In-In formalism is routinely used to calculate the equal time correlation functions of the curvature perturbation, ζ, as wavelengths cross the cosmological horizon [3–6] where they become constant [7, 8]. For many models this calculation can be accurately performed analytically, while for others, such as models with features, a numerical implementation is required [9–12]. If additional fields are added, the problem becomes even more complex. ζ is no longer necessarily conserved after horizon crossing, and the evolution of all isocurvature modes needs to be accounted for – all the way from the initial vacuum state until such time as the system becomes adiabatic, or until the time at which we wish to know the statistics (see for example Ref. [13] for a discussion of adiabaticity). While analytic progress can be made in some circumstances using the In-In formalism and/or so called "super-horizon" techniques such as the δN formalism, in general for multiple field models numerical techniques become even more important.

The code documented here accounts for all tree-level effects present in multi-field inflation. This includes the super-horizon evolution of ζ, which can occur in models with multiple light fields, as well as the effect of features in the multidimensional potential, and the effect of quasi-light or heavy fields orthogonal to the inflaton (which are important if the inflationary trajectory is not straight). As discussed above the code utilises the transport approach to inflationary correlation functions [14–19]. This approach can be viewed as a differential version of the integral expressions of the In-In formalism, and evolves correlations of inflationary perturbations from their vacuum state on sub-horizon scales until we wish to evaluate the statistics. We note for clarity that in its original form the "moment transport" method was restricted to super-horizon scales, but it was later shown how it could be extended to include sub-horizon scales in Ref. [19]. A recent paper studies this extension further and develops it into a working algorithm

[1] with many additional details provided. The present document details the code PyTransport which is discussed in that paper.

At the background level an inflationary cosmology is completely determined by the evolution of the fields, ϕ_i, and their associated velocities (the rate at which the fields change with cosmic time), $\dot{\phi}_i$, as a function of the number of e-folds (the logarithm of the scale factor, $N = \ln(a)$) which occurs. At the perturbed level, the key objects are the correlations of the perturbations in these fields, and correlations of other perturbative quantities. Here we have used the label i to run over the number of fields present. The code numerically solves the equations of motion for the background fields, and the equations of motion for the evolution of correlations of the field and field velocity perturbations defined on flat hyper-surfaces. Ultimately the quantities probably of most interest for observations are the statistics of the curvature perturbation ζ – in particular the power spectrum and bispectrum – which the code calculates from the field space correlations. Defining the array $\mathbf{X} = \{\delta\phi, \delta\dot{\phi}\}$, where the components are labelled X^a and a now runs over the total number of fields and field velocities, we recall the following definitions for later clarity:

$$\langle\zeta(\mathbf{k_1})\zeta(\mathbf{k_2})\rangle = (2\pi)^3\delta(\mathbf{k}_1+\mathbf{k}_2)P_\zeta(k_1)\,, \tag{2.1}$$

$$\langle X^a(\mathbf{k_1})X^b(\mathbf{k_2})\rangle = (2\pi)^3\delta(\mathbf{k}_1+\mathbf{k}_2)\Sigma^{ab}(k_1)\,, \tag{2.2}$$

$$\langle\zeta(\mathbf{k_1})\zeta(\mathbf{k_2})\zeta(\mathbf{k_3})\rangle = (2\pi)^3\delta(\mathbf{k}_1+\mathbf{k}_2+\mathbf{k}_3)B_\zeta(k_1,k_2,k_3)\,, \tag{2.3}$$

$$\langle X^a(\mathbf{k_1})X^b(\mathbf{k_2})X^c(\mathbf{k_3})\rangle = (2\pi)^3\delta(\mathbf{k}_1+\mathbf{k}_2+\mathbf{k}_3)B^{abc}(k_1,k_2,k_3)\,, \tag{2.4}$$

$$\frac{6}{5}f_{\mathrm{NL}}(k_1,k_2,k_3) = B_\zeta(k_1,k_2,k_3)/\left(P_\zeta(k_1)P_\zeta(k_2)+P_\zeta(k_2)P_\zeta(k_3)+P_\zeta(k_1)P_\zeta(k_3)\right)\,, \tag{2.5}$$

where P_ζ and B_ζ are power spectrum and bispectrum of ζ respectively, and Σ and B are the equivalent functions for the correlations and cross correlations in field space, and f_{NL} is the reduced bispectrum. Σ and B together with the background values of the fields and field velocities are the objects directly evolved by the code using the equations detailed in section 5 of Ref. [1] with initial conditions detailed in section 6 of Ref. [1][1]. As discussed in that paper, B and Σ can then readily be converted to give P_ζ and B_ζ through the use of the "N" tensors with components N_a and N_{ab}, described in section 7 of Ref [1] (see also [20]). The equations of motions for the correlations are given in Eqs. (5.5) and (5.16) of Ref. [1], the initial conditions in Eqs. (6.2) and (6.7), and the conversion to ζ in Eq. (7.4).

It is worth briefly commenting on how our code compares with existing ones. A number of publicly available codes exist to calculate the power spectrum from canonical multi-field inflation. For example Pyflation [21] and MultiModeCode [22] employ a method originally used by Salopek and Bond [23] in which the mode functions of the QFT of inflationary perturbations are evolved. These codes are in written Python[2] and Fortran respectively. Moreover, a Mathematica code which implements the transport method for curved field space metric models, but is restricted to the power-spectrum, is also available [24]. At the level of the three-point function a number of authors have undertaken numerical work directly utilising the In-In formalism, for example in Refs. [9–12, 25]. The only publicly released code we are aware of, however, is BINGO [11] which is restricted to single field models. No general multi-field codes have been undertaken until now.

A further advantage of PyTransport (and CppTransport) over previous codes is that it leaves little for the user to calculate analytically. It needs the user only to provide the inflationary potential. Then all the derivatives are automatically calculated using symbolic Python (SymPy) and written automatically into the C++ code which is then compiled. Compared to Fortran or pure C++ implementations PyTransport has the advantage of easy access to the extensive and easy to use modules available to Python, and compared to a pure Python or Mathematica implementation we have the advantage of speed.

III. CODE OVERVIEW

The code structure should become familiar though the extended example we provide, but here we give a brief summary.

The code is distributed in a folder called *PyTransportDist/* , which also contains a copy of this document (possibly updated compared with the arXiv version) in the *PyTransportDist/docs/* folder. The base code for PyTransport is written in C++ and has a simple object orientated structure. This code can be found in the *PyTransportDist/PyTransport/CppTrans* folder and we provide a few more details about its structure and functionality in appendix 1. The C++ code is deliberately as simple as possible to ensure transparency and adaptability. The idea

[1] Internally the code internally rescales the field velocity perturbation $\delta\dot{\phi}$ for performance reasons, as well as internally rescaling the wavenumbers involved, but the rescaling is reversed before outputting results – this is discussed further in appendix 1.

[2] Pyflation also makes sure of C code for speed through the use of Cython.

of the PyTransport package as a whole is that after a potential is provided by the user the C++ code is automatically edited and complied into a Python module by supporting Python functions (called from the *PyTransportDist/PyTransport/PyTransSetup.py* file which is described in full in appendix 2), meaning a lot of work is done for the user. The end result is a Python module consisting of a set of Python functions for a specific inflationary potential, called the PyTrans*** module. The functions of this module provide key routines for inflationary cosmology (including calculating the evolution of the two and three point correlations). The asterisks, ***, indicate we can label the module with a tag telling us what potential it corresponds to, and we can therefore install multiple modules if we want to work with many potentials simultaneously. The key functions available to these modules are defined in the file *PyTransportDist/PyTransport/PyTrans/PyTrans.cpp* (which is a C++ file defining the Python module), these functions are detailed in appendix 3. The scripts that edit the C++ code and compile the module are discussed further below in the setup section, and by default they place the compiled module in the local folder *PyTransportDist/PyTransport/PyTrans/lib/python/* to avoid access issues if, for example, you do not have root privileges. Other useful Python functions that perform common tasks, such as producing a power spectrum by looping over calls to the compiled module, can be found in *PyTransportDist/PyTransport/PyTransScripts.py*, and we describe them below, and in full in detail in appendix 4. Python treats functions written in Python inside a file, such as *PyTransScripts.py* and *PyTransSetup.py*, in the same way as a compiled module. So there are effectively **three modules within PyTransport**, one to setup a compiled module for the potential we want to study (**PyTransSetup**), the compiled module itself (**PyTrans*****) (or multiple complied modules labeled with different tags) and a module with various functions automating common tasks that use the functions of the compiled module (**PyTransScripts**). Also in the *PyTransportDist/* folder is an example folder *PyTransportDist/Examples* containing the examples discussed below. There are no dependancies external to the folders provided except for a working Python installation (with appropriate packages downloaded), and a C++ compiler – this is deliberate to make the code as easy as possible to use. An MPI installation such as openMPI is also needed if the module is required to be used across multiple cores.

We note that all the C++ code is written by the transport team except for an included Runge-Kutta-Fehlberg (rkf45) integrator routine written by John Burkardt and distributed under a GNU LGPL license detailed here[3]. We choose this lightweight integrator over other options, such as using integrators included with the BOOST library, in order that it could easily be included with the distribution with no external dependancies being introduced. In our (limited) testing it functions well for all the models we have looked at. There are no dependancies external to the folders provided except for a working Python installation (with appropriate packages downloaded), and a C++ compiler – this is deliberate to make using the code as easy as possible to use. An MPI installation such as openMPI is also needed if the module is required to be used across multiple cores.

IV. SETUP

A. Prerequisites

So what is needed? The idea is as little as possible beyond Python:

Python: A working Python installation is needed, in development we used Python 2.7 which we recommend, but have subsequently attempted to ensure compatibility with versions of Python 3. For convenience we recommend downloading a complete Python distribution, for example Enthought Canopy or Continuum Anaconda, which come with all the core packages used by the code as well as interactive development environments. Python packages currently used by PyTransport or by provided examples include Numpy, Matplotlib, SciPy, SymPy, Distutils, Math and Sys, as standard, and Mpi4Py and Mayavi are used for MPI and 3D bispsectra plots respectively. Of these only Mpi4Py and Mayavi may need to be downloaded separately from the distributions mentioned. One way to install a package such as Mpi4Py is to type "pip install Mpi4Py" in the terminal. If using Canopy, Anaconda or similar, they come with their own package managers which are an even easier way to install packages. There are many easily found resources on the internet to help with such installations if there is a system specific snag. Note that you should not attempt to install Mpi4Py without installing MPI first (which we deal with next). We also note that although apparently possible we have not easily been able to install Mayavi with Python 3.5, and recommend searching for online resources to help with this.

MPI: As computing the bispectrum can be computationally expensive, distributed computing can be helpful (even if only across the multiple cores of modern PCs). In some of the scripts in the *PyTransScripts* module, we use the

[3] https : //people.sc.fsu.edu/ jburkardt/f$_s$rc/rkf45/rkf45.html

Mpi4Py module to implement this. Mpi4Py needs a working MPI installation such as openMPI installed on your computer. Note that Mpi4py or openMPI are not needed for PyTransport in general, and if you do not have these installed you simply cannot run the scripts that use MPI, but can run the code in serial instead. A nice guide to installing openMPI is at this link[4].

C++ complier: Python needs to be able to find a C++ complier in order to compile the PyTrans module(s). This is bundled with most Linux distributions. If not present on a Mac system, downloading Xcode from the app store is the easiest way to install one (or Xcode command line tools can be downloaded separately, as explained here[5]).

B. Getting going

Once you have Python running and a C++ compiler, to get started take the *PyTransport/* folder from the *PyTransportDist/* folder and place it anywhere convenient in your computer's file system. It is essential that you don't change the structure of the sub-directories within *PyTransport/* , but you can place this folder wherever you want. That's more or less all you have to do. You can do this by copying the entire folder *PyTransportDist/* (which also contains examples and this guide) to a convenient location, but equally well you could run examples from anywhere else on your computer. In each example, we will see one needs to add the path of the *PyTransport/* folder to the paths which Python includes when looking for code, so that Python can find the setup file *PyTransSetup.py* (or this could be done permanently). Now you can get started, no other installation is required which is not handled for you by provided Python scripts.

Lets say you want to analyse the inflationary model defined by the potential

$$V = \frac{1}{2}m_\phi^2\phi^2 + \frac{1}{2}m_\chi^2\chi^2 \,. \tag{4.1}$$

The first step is to create (by compiling the C++ code) a Python module for this potential. This is achieved by writing the potential in a Python file in SymPy (symbolic Python notation) and calling the appropriate functions from the PyTransSetup module. First define two SymPy arrays one for the fields and the other for parameters which define the model (the parameters you might wish to change the value of). These have length nF and nP respectively. Then define the symbolic potential V using these arrays. The potential must be written into the C++ code by calling the function *potential(V,nF,nP)* which is in the file *PyTransSetup.py*. If working with a version of Python 2, the next step is to call the function *compileName("Quad")* (where "Quad" can be replaced by any name the user likes (the *** from above)). If using Python 3, users must use the function *compileName3("Quad")* to achieve the same thing. Multiple modules for different potentials with different names can therefore be created and used simultaneously. Below is a screen shot of the procedure just described with copious comments which should make the procedure clear:

```
##################################### Setup file for the double quadratic potential ##################################################

import sympy as sym    # we import the sympy package
import math            # we import the math package (not used here, but has useful constants such math.pi which might be needed in other cases)
import sys             # import the sys module used below

######################################################################################################################################

# if using an integrated environment we recommend restarting the python console after running this script to make sure updates are found

location = "/Users/david/Desktop/PyTransportDist/PyTransport/" # this should be the location of the PyTrans folder
sys.path.append(location)  # we add this location to the python path

import PyTransSetup  # the above commands allows python to find the PyTransSetup module and import it

######################################################################################################################################

nF=2  # number of fields needed to define the double quadratic potential
nP=2  # number of parameters needed to define the double quadtartic potential
f=sym.symarray('f',nF)   # an array representing the nF fields present for this model
p=sym.symarray('p',nP)   # an array representing the nP parameters needed to define this model (that we might wish to change) if we don't
                         # wish to change them they could be typed explicitly in the potential below

V = 1./2. * p[0]**2.0 *f [0]**2.0  +  1./2. * p[1]**2.0 * f[1]**2.0   # this is the potential written in sympy notation

PyTransSetup.tol(1e-8,1e-8)   # set tols for the numerical integration of the 3 point function (if not run this will remain as it the
                              #last time set)
PyTransSetup.potential(V,nF,nP) # writes this potential and its derivatives into C++ file potential.h when run

PyTransSetup.compileName3("DQuad") # this compiles a python module using the C++ code, including the edited potential.h file, called PyTransDQuad
                                   # and places it in the location folder, ready for use
```

[4] https://wiki.helsinki.fi/display/HUGG/Open+MPI+install+on+Mac+OS+X
[5] http://railsapps.github.io/xcode-command-line-tools.html

This example is contained in the *Examples/DoubleQuad/* folder which accompanies the code, with this script in the file *DQuadSetup.py* file. In this script we have used three functions from the PyTransSetup module, the two mentioned above, and *tols(rel, abs)*, which sets the relative and absolute tolerances of the ode integrator (note we need to reinstall the compiled module if we wish to alter these tolerances). Appendix 1 contains a summary of all the functions available in the setup module.

The complied Python module can now be used. To do so we need to point Python to the path of the new module (and the scripts module if we wish to call the provided Python scripts). This can be done automatically by calling the function in the setup module *pathSet()*. We recommend using the *PyTransQuad* module in a separate file from the one used to set it up, and if working in an integrated development environment to restart the Python kernel (this is to ensure the most recent version is always imported). Below is a screen shot of the start of a file in which we use the module we set up in the previous paragraph. In the screen shot we first calculate the value of the potential and the first derivative of the potential for a particular choice of field values and parameters. Then we use these to set up an array containing field values and the associated fields velocity (using the slow roll equation):

```
################### Simple example of using PyTransDQuad installed using the setup file which accompanies this one ########################

from matplotlib import pyplot as plt   # import package for plotting
from pylab import *  # contains some useful stuff for plotting
import time  # imports a package that allows us to see how long processes take
import math  # imports math package
import numpy as np # imports numpu package as np for short
import sys  # imports sys package for sue below
#################################################################################################################################################

#This file contains simple examples of using the PyTransDQuad
#It assumes the DQuadSetup file has been run to install a double quadratic version of PyTrans
#It is recommended you restart the kernel before running this file to insure any updates to PyTransDQuad are imported

location = "/Users/david/Desktop/PyTransportDistTest/PyTransport/" # this should be the location of the PyTransport folder folder
sys.path.append(location) # sets up python path to give access to PyTransSetup

import PyTransSetup
PyTransSetup.pathSet()  # this adds the other paths that PyTrans uses to the python path

import PyTransDQuad as PyT  # import module as PyT (PyTransDQuad is quite long to type each time and it saves time to use a shorter name
                            # using a generic name PyT means the file can be more easily reused for a different example (once field values
                            # etc are altered)
import PyTransScripts as PyS  # import the scripts module as PyS for convenience

#################################################################################################################################################



# Example

########################### set some field values and field derivatives in cosmic time ###################################################

fields = np.array([12.0, 12.0]) # we set up a numpy array which contains the values of the fields

nP=PyT.nP()   # the .np function gets the number of parameters needed for the potential -- this can be used as a useful crosscheck
pvalue = np.zeros(nP)
pvalue[0]=10.0**(-5.0); pvalue[1]=9.0*10.0**(-5) # we set up numpy array which contains values of the parameters

nF=PyT.nF() # use number of fields routine to get number of fields (can be used as a useful cross check)

V = PyT.V(fields,pvalue) # calculate potential from some initial conditions
dV=PyT.dV(fields,pvalue) # calculate derivatives of potential

initial = np.concatenate((fields, -dV/np.sqrt(3* V))) # sets an array containing field values and there derivative in cosmic time
                                                      # (set using the slow roll equation)

#################################################################################################################################################
```

This screen shot is of the start of the file *Examples/DoubleQuad/SimpleExample.py*. Of course we will usually want to use the module for more sophisticated tasks. Appendix 3 contains a summary of all the functions available within the PyTrans*** module. We will see the use of a number of the more sophisticated functions in the Examples section.

V. EXAMPLES

A. Double quadratic

First lets continue with the double quadratic example using more of the functions available from the compiled module. In the screen shot below we use the background evolution function to calculate a fidicual background trajectory in field space using the array we set up in the last part of the example as initial conditions. Here and for all the functions and output of the PyTransport package e-folds (N) are used as the time variable. The function used to calculate the background evolution is the *PyTrans.backEvolve* function. The output is plotted shown in Fig. 1.

Then we run the two point evolution function to calculate the evolution of Σ and the power spectrum of ζ for a k mode which crossed the horizon 15 e-folds into this fiducial run using the *PyTrans.sigEvolve* function. We plot the correlations and cross correlations of the fields in Fig. 2. We repeat for a neighbouring k to give us a crude estimate of the spectral index, n_s. Finally, we use the *PyTrans.alphaEvolve* function to calculate the evolution of the field space three-point function and the bispectrum of ζ for a set of three ks. We plot the three-point correlations and cross correlations of the fields in Fig. 2 and also the evolution of the reduced bispectrum, $f_{\rm NL}$. In the plots it can clearly be seen that the heavier field drops out of the dynamics at around 40 e-folds. At this point the system becomes adiabatic and ζ and its statistics become constant. A screen shot of the code which does all this from the *SimpleExample.py* file is below:

```
################################## run and plot the background fiducial run ###########################################################
Nstart = 0.0
Nend = 60.0
t=np.linspace(Nstart, Nend, 1000)

back = PyT.backEvolve(t, initial, pvalue)


fig1=plt.figure(1)
plt.plot(back[:,0], back[:,1], 'g')
plt.plot(back[:,0], back[:,2], 'r')
title(r'Background evolution',fontsize=15); grid(True); plt.legend(fontsize=15); ylabel(r'Fields', fontsize=20); xlabel(r'$N$', fontsize=15)
grid(True); plt.legend(fontsize=15); plt.savefig("DQ1.png")
#######################################################################################################################################



#######################################################################################################################################
# set a pivot scale which exits after certain time using the background run -- a spline
# is used to find field and field velocity values after Nexit number of e-folds, this gives H, and
# then k=aH gives the k pivot scale
# in this example we treat this scale as k_t

NExit = 15.0
k = PyS.kexitN(NExit, back, pvalue, PyT)



################################## example 2pt run ####################################################################################

NB = 6.0
Nstart, backExitMinus = PyS.ICsBE(NB, k, back, pvalue, PyT) #find conditions for 6 e-folds before horizon crossing of k mode


tsig=np.linspace(Nstart,Nend, 1000)  # array of times (e-folds) at which output is returned -- initial time should correspond to initial field
                                     # and velocity values which will be fed in to the functions which evolve correlations


# run the sigma routine to calc and plot the evolution of power spectrum value for this k -- can be
# repeated to build up the spectrum, here we run twice to get an crude estimate for ns
twoPt = PyT.sigEvolve(tsig, k, backExitMinus,pvalue, 1) # puts information about the two point fuction in twoPt array
zz1=twoPt[:,1] # the second column is the 2pt of zeta
sigma = twoPt[:,1+1+2*nF:] # the last 2nF* 2nF columns correspond to the evolution of the sigma matrix
zz1a=zz1[-1] # the value of the power spectrum for this k value at the end of the run

twoPt=PyT.sigEvolve(tsig, k+.1*k, backExitMinus,pvalue, 1)
zz2=twoPt[:,1]
zz2a=zz2[-1]
n_s = (np.log(zz2a)-np.log(zz1a))/(np.log(k+.1*k)-np.log(k))+4.0


fig2=plt.figure(2)
for ii in range(0,2):
    for jj in range(0,2):
        plt.plot(twoPt[:,0], np.abs(sigma[:,ii + 2*nF*jj]))
title(r'$\Sigma$ evolution',fontsize=15); grid(True); plt.legend(fontsize=15); ylabel(r'Aboslute 2pt field correlations', fontsize=20);
xlabel(r'$N$', fontsize=15); grid(True); yscale('log'); plt.legend(fontsize=15); plt.savefig("DQ2.png")


fig3=plt.figure(3)
plt.plot(tsig, zz1[:])
title(r'$P_\zeta$ evolution',fontsize=15); grid(True); plt.legend(fontsize=15); ylabel(r'$P_\zeta(k)$', fontsize=20);
xlabel(r'$N$', fontsize=15); grid(True); yscale('log'); plt.legend(fontsize=15); plt.savefig("DQ3.png")
```

```
################################### example bispectrum run ###########################################################################

# set three scales in FLS manner (using alpha, beta notation)
alpha=0.9
beta =.01

k1 = k/2 - beta*k/2. ; k2 = k/4*(1+alpha+beta) ; k3 = k/4*(1-alpha+beta)


# find initial conditions for 6 e-folds before the smallest k (which exits the horizon first) crosses the horizon
kM = np.min(np.array([k1,k2,k3]))
Nstart, backExitMinus = PyS.ICsBM(NB, kM, back, pvalue, PyT)


# run the three point evolution for this triangle
talp=np.linspace(Nstart,Nend, 1000)
threePt = PyT.alphaEvolve(talp,k1,k2,k3, backExitMinus,pvalue,1) # all data from three point run goes into threePt array
alpha= threePt[:,1+4+2*nF+6*2*nF*2*nF:]        # this now contains the 3pt of the fields and field derivative pertruabtions
zzz= threePt[:,1:5] # this contains the evolution of two point of zeta for each k mode involved and the 3pt of zeta

fig4 = plt.figure(4)
for ii in range(0,2):
    for jj in range(0,2):
        for kk in range(0,2):
            plt.plot(talp, np.abs(alpha[:,ii + 2*nF*jj + 2*nF*2*nF*kk]))
title(r'$\alpha$ evolution',fontsize=15); grid(True); plt.legend(fontsize=15); ylabel(r'Absolute 3pt field correlations', fontsize=20);
xlabel(r'$N$', fontsize=15); grid(True); yscale('log'); plt.legend(fontsize=15); plt.savefig("DQ4.png")

fig5=plt.figure(5)
fnl = 5.0/6.0*zzz[:,3]/(zzz[:,1]*zzz[:,2]  + zzz[:,0]*zzz[:,1] + zzz[:,0]*zzz[:,2])
plt.plot(talp, fnl,'r')
title(r'$f_{NL}$ evolution',fontsize=15); grid(True); plt.legend(fontsize=15); ylabel(r'$f_{NL}$', fontsize=20);
xlabel(r'$N$', fontsize=15); grid(True); plt.legend(fontsize=15); plt.savefig("DQ5.png")
```

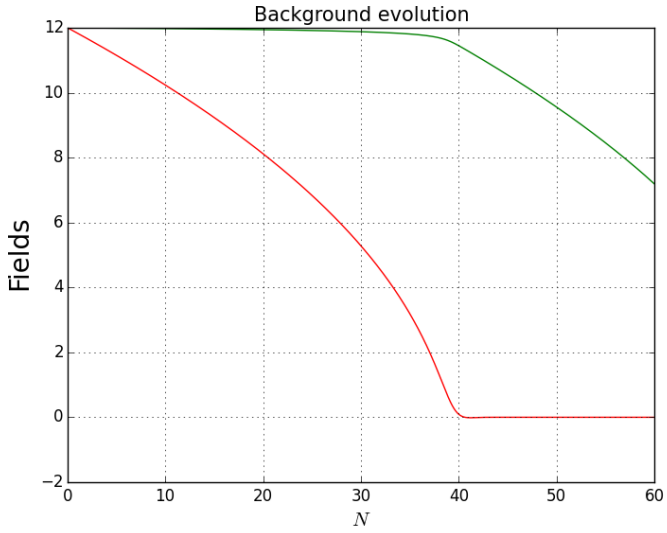


FIG. 1: Background evolution for double quadratic potential

There are few things to note from this script. First we note that it is important that we fix our final time sensibly. If we were to fix them after the end of inflation, when the lighter field oscillates indefinitely about its minimum, then the code would become very slow when it evaluates the correlations (since the the field correlations also oscillate a lot in response). Next we note the use of a function within *PyTransScripts.py* which finds the k value which corresponds to an exit time of 15 e-folds after the start of the fiducial run, the *PyTransScripts.kexitN* function. This function uses the background trajectory and Python spline routines to find this k. We also note that it is essential that we run the two and three point correlation evolution from a time the k mode of interest is deep inside the horizon. In the script, we calculate this time as well as the field and field velocity values at this time (which are then fed into the two and three point evolution routines) using another function from the scripts module, the *PyTransScripts.ICsBE* function. To use this function we need to specify the number of e-folds we require before horizon crossing, and here we specify 6.0. The function returns a time roughly 6 e-folds before the horizon crossing time and a numpy array containing the fields' values and velocities at that time. It is important to point out that this function is only approximate in the sense that it simply finds the first value of N in the background array (back) which is before the specified number of e-folds before horizon crossing, and returns this value and the corresponding field and field velocities at this time. It therefore requires the background fiducial evolution which is fed into it to be finely sampled (10 points

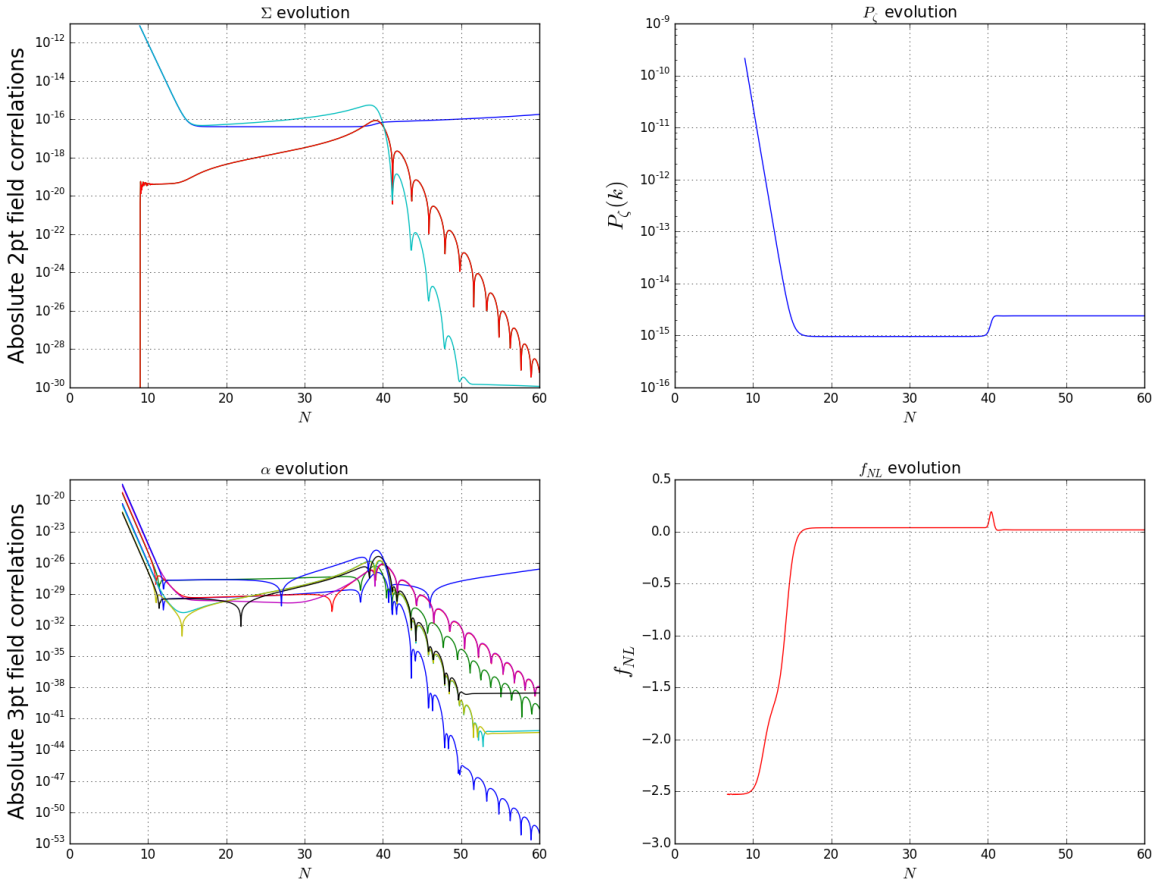


FIG. 2: Evolution of correlations for double quadratic potential from 6 e-folds inside the horizon for a Fourier mode which crosses the horizon at $N = 15$

per e-fold is a rough guide) to be accurate. Finally, we note that within *PyTransScripts.py* we also include the related functions *kexitPhi* which finds the k value which crosses the horizon at a particular field value, and and *ICsBM* which finds initial conditions a fixed time before the "massless condition" which is discussed further in section V B. These functions and all others in this module are detailed in appendix 4.

Calculating the value of n_s in the manner presented here is clearly a bad way of doing things, since it involves using only two points in the power spectrum to calculate a derivative, and the step between them is arbitrarily chosen. An alternative is to calculate the power spectrum around a given exit time using a number of points, and to fit a spline to it and differentiate. We now generate the power spectrum for this model with the following script which fits a spline to the entire spectrum, differentiates and produces n_s at every value of k over roughly 30 e-folds, the results are plotted in Fig. 3 (the start of the file is identical to that above):

```
################################## run the background fiducial run ###################################################################
Nstart = 0.0
Nend = 70.0
t=np.linspace(Nstart, Nend, 1000)

back = PyT.backEvolve(t, initial, pvalue)
######################################################################################################################################


######################################################################################################################################
# set up an array of k values to calculate power spectrum as a function of ks
kOut = np.array([])
for ii in range(0,500):
    NExit = 10.0 + ii*0.08
    k = PyS.kexitN(NExit, back, pvalue, PyT)
    kOut = np.append(kOut,k) # this builds an array of ks associated with different NExit times from 10 to 50
Pz, times = PyS.pSpectra(kOut,back,pvalue,4.0,PyT) # this cacalcute P_z for this range of ks, using 5.0 e-folds of subhorizon evolution
#=============================================================================
# zz, zzz, timesB = PyS.eqSpectra(kOut, back, pvalue, 4.0, PyT)
#=============================================================================


fig1=plt.figure(1)
plt.plot(np.log(kOut/kOut[0]), Pz/Pz[0] *kOut**3/kOut[0]**3, linewidth = 2 )
title('Power spectrum',fontsize=15); grid(True); plt.legend(fontsize=15); ylabel(r'$\log({\cal P}/{\cal P}_{\rm pivot})$', fontsize=20);
xlabel(r'$\log(k/k_{\rm pivot})$', fontsize=15);yscale('log'); plt.xlim(min(np.log(kOut/kOut[0])),max(np.log(kOut/kOut[0])));plt.savefig("Pz.png")

#fnl=5.0/6.0*zzz/(3.0*zz**2.0)
#plt.plot(np.log(kOut/kOut[0]), fnl )
##fig3=plt.figure(3)
#title(r'$f_{NL}$',fontsize=15); grid(True); plt.legend(fontsize=15); ylabel(r'$f_{NL}$', fontsize=20);
#xlabel(r'$\log(k/k_{\rm pivot})$', fontsize=15); plt.xlim(min(np.log(kOut/kOut[0])),max(np.log(kOut/kOut[0]))); plt.savefig("fnl.png")

######################################################################################################################################


######################################################################################################################################
# finally lets find n_s
from scipy.interpolate import UnivariateSpline
derivativeSp = UnivariateSpline(np.log(kOut/kOut[0]), np.log(Pz),k=4, s=1e-15).derivative()
ns=derivativeSp(np.log(kOut/kOut[0]))+4.0

fig2=plt.figure(2)
plt.plot(np.log(kOut/kOut[0]), ns , linewidth = 2)
title('Spectral index',fontsize=15); grid(True); plt.legend(fontsize=15); ylabel(r'$n_s$', fontsize=20);
xlabel(r'$\log(k/k_{\rm pivot})$', fontsize=15); plt.xlim(min(np.log(kOut/kOut[0])),max(np.log(kOut/kOut[0]))); plt.savefig("ns.png")
```

In this script we used the *PyTranScripts.pSpectra* function to generate the power spectrum over a range of ks. This function essentially just loops over calls to the compiled function which evolves the two-point function.

Next we wish to calculate the bispectrum. Here we first we calculate the bispectrum in the equilateral triangle configuration as a function of k value which. Then we generate (and plot using a separate plots file) a slice through the bispectrum for a given k_t as a function of the α, β variables defined such that $k_1 = k_t/2 - \beta k_2/2$, $k_2 = k_t/4*(1+\alpha+\beta)$ and $k3 = k_t/4*(1-\alpha+\beta)$. We use two separate scripts for each of these tasks (and the plots file) which are pasted below, both use MPI to speed things up. They should be called using the command "/usr/local/bin/mpiexec', '-n', '10', 'python', 'MpiEqBi.py' " (for the equilateral run, where the first part should be replaced with your location of mpiexec if different, and 10 replaced by the number of processes one desires to call):

```
################### generate equilateral bispectrum using PyTransDQuad and MPI ###################################################################
import math  # imports math package
import numpy as np # imports numpu package as np for short
import sys  # imports sys package for sue below

from mpi4py import MPI

location = "/Users/mulryne/Dropbox/PyTransportDist/PyTransport/" # this should be the location of the PyTrans folder
sys.path.append(location) # sets up python path to give access to PyTransSetup

import PyTransSetup
PyTransSetup.pathSet()  # this sets the other paths that PyTrans uses

import PyTransDQuad as PyT  # import module
import PyTransScripts as PyS

comm = MPI.COMM_WORLD

######################### initial field values ###########################################################################################################
fields = np.array([12.0, 12.0]) # we set up a numpy array which contains the values of the fields

nP=PyT.nP()   # the .nP() function gets the number of parameters needed for the potential -- this can be used as a useful crosscheck
pvalue = np.zeros(nP)
pvalue[0]=10.0**(-5.0); pvalue[1]=9.0*10.0**(-5) # we set up numpy array which contains values of the parameters

nF=PyT.nF() # use number of fields routine to get number of fields (can be used as a useful cross check)

V = PyT.V(fields,pvalue) # calculate potential from some initial conditions
dV=PyT.dV(fields,pvalue) # calculate derivatives of potential (changes dV to derivatives)

initial = np.concatenate((fields, -dV/np.sqrt(3* V))) # sets an array containing field values and there derivative in cosmic time
                                                      # (set using the slow roll equation)
###################################################################################################################################################


################################### run the background fiducial run ###############################################################################
Nstart = 0.0
Nend = 70.0
t=np.linspace(Nstart, Nend, 1000)
back = PyT.backEvolve(t, initial, pvalue)
###################################################################################################################################################

rank=comm.Get_rank()
points = 500

NExit1 = 10.0
NExit2 = 10.0+0.08*points
k1 = PyS.kexitN(NExit1, back, pvalue, PyT)
k2 = PyS.kexitN(NExit2, back, pvalue, PyT)
kOut= np.logspace(log10(k1), log10(k2), points)

Pztot, Bztot, times = PyS.eqSpecMpi(kOut, back, pvalue, 5.0, PyT)

print "\n\n process", rank, "done \n\n"

if rank ==0:
    fnlOut = 5.0/6*Bztot/(3.0*Pztot**2.0)
    fig2 = plt.figure(2)
    plt.plot(np.log(kOut/kOut[0]), fnlOut, linewidth=2)
    title('Reduced bispectrum in equilateral configuration',fontsize=15) ;grid(True); plt.legend(fontsize=15); ylabel('$fNL$', fontsize=20)
    xlabel(r'$\log(k/k_{\rm pivot})$', fontsize=15); grid(True); plt.legend(fontsize=15); plt.xlim(min(np.log(kOut/kOut[0])),max(np.log(kOut/kOut[0])));
    plt.savefig("BiEq.png")

    np.savetxt('data/EqBi.dat', (kOut,fnlOut,Bztot,Pztot))   # x,y,z equal sized 1D arrays
```

```
################### generate alpha beta  bispectrum using PyTransDQuad and MPI ###################################################################
from matplotlib import pyplot as plt   # import package for plotting
from mpl_toolkits.mplot3d import Axes3D
from pylab import *  # contains some useful stuff for plotting
import time  # imports a package that allows us to see how long processes take
import math  # imports math package
import numpy as np # imports numpu package as np for short
import sys  # imports sys package for sue below

from mpi4py import MPI

location = "/Users/mulryne/Dropbox/PyTransportDist/PyTransport/" # this should be the location of the PyTrans folder
sys.path.append(location) # sets up python path to give access to PyTransSetup

import PyTransSetup
PyTransSetup.pathSet()  # this sets the other paths that PyTrans uses

import PyTransDQuad as PyT  # import module
import PyTransScripts as PyS

comm = MPI.COMM_WORLD

######################### initial field values ###########################################################################################
fields = np.array([12.0, 12.0]) # we set up a numpy array which contains the values of the fields

nP=PyT.nP()   # the .nP() function gets the number of parameters needed for the potential -- this can be used as a useful crosscheck
pvalue = np.zeros(nP)
pvalue[0]=10.0**(-5.0); pvalue[1]=9.0*10.0**(-5) # we set up numpy array which contains values of the parameters

nF=PyT.nF() # use number of fields routine to get number of fields (can be used as a useful cross check)

V = PyT.V(fields,pvalue) # calculate potential from some initial conditions
dV=PyT.dV(fields,pvalue) # calculate derivatives of potential (changes dV to derivatives)

initial = np.concatenate((fields, -dV/np.sqrt(3* V))) # sets an array containing field values and there derivative in cosmic time
                                                      # (set using the slow roll equation)
##################################################################################################################################################


################################## run the background fiducial run ###############################################################################
Nstart = 0.0
Nend = 50.0
t=np.linspace(Nstart, Nend, 1000)
back = PyT.backEvolve(t, initial, pvalue)
##################################################################################################################################################

side = 100
nsnaps = 0
Nbefore=4.5
rank=comm.Get_rank()

NExit = 15.0
kt = PyS.kexitN(NExit, back, pvalue, PyT)

alpha =  np.linspace(-1,1,side)
beta=np.linspace(0,1,side/2)

Bztot, Pz1tot, Pz2tot, Pz3tot,  times, snaps = PyS.alpBetSpecMpi(kt,alpha, beta, back, pvalue, Nbefore, nsnaps, PyT)

if rank == 0:
    bet, alp = np.meshgrid(beta, alpha)
    np.save('data/alp',alp);np.save('data/bet',bet); np.save('data/alBetBi',Bztot)
    np.save('data/alBetPz1',Pz1tot); np.save('data/alBetPz2.npy',Pz2tot); np.save('data/alBetPz3',Pz3tot)
print "\n\n process", rank, "done \n\n"

```

The results are in Figs. 4 and 5 respectively. Note that in all these scripts we use more functions available from the PyTransScripts module whose function should be self evident and which are described in full in appendix 4. If one didn't wish to use MPI the only change needed would be to call the function *alpBetSpec* rather than *alpBetSpecMpi* from the scripts module, and to remove MPI related lines and reference to rank etc. When using MPI we recommend calling more processes that the system has cores. This is because we have not implemented sophisticated load sharing, and since some ranges will be faster to evaluate than others, if the number of processes is larger than cores, the cores that have capacity first will end up running more processes, sharing the load in a simple way. In this example we also save the data at the end for future use. This can be done simply for any numpy array in various ways and then read back into Python easily. For reasons of simplicity and flexibility we leave data management up to the user, but note that Python is a powerful tool for this purpose.

If we want to generate the full bispectrum we would simply loop over the *alpBetSpec* or *alpBetSpecMpi* function for many k_t.

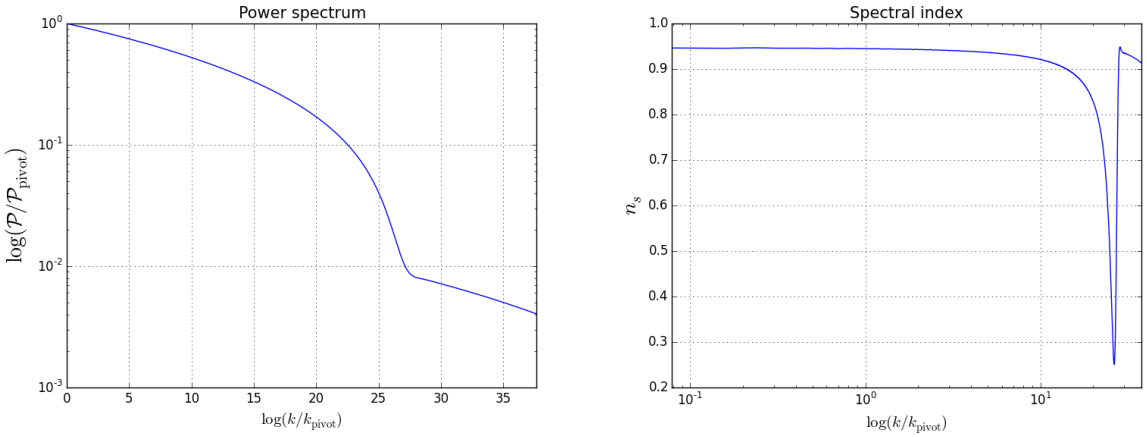


FIG. 3: The power spectrum and n_s in the double quadratic model.

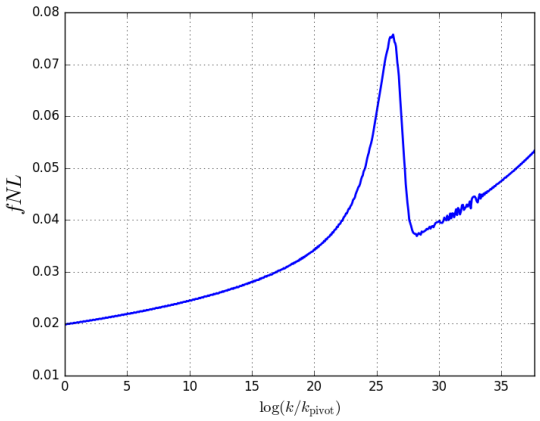


FIG. 4: The reduced bispectrum in equilateral configurations for the double quadratic potential.

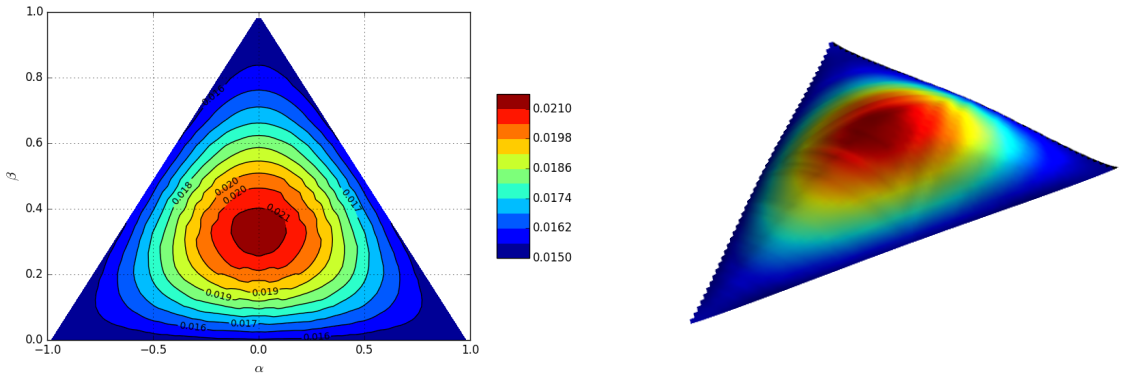


FIG. 5: A slice through the reduced bispectrum for a mode exiting after 15 e-folds for the double quadratic potential run discussed in the text.

B. Heavy field examples

In the previous example both fields which played a role in the dynamics were light (at least at the start of the evolution). Interesting dynamics can also occur when the field orthogonal to the direction of travel in field space is heavy, if the field trajectory curves. In this kind of example it is imperative that the initial conditions for the evolution of the two and three point functions are set when the k^2/a^2 term in the equation of motion for the scalar field perturbations dominates over the mass squared of the heavy field. This is a requirement for our initial conditions to be accurate as discussed in Ref [1]. There is a script in *PyTransScripts.py* which will achieve this, the *ICsBM* function. This finds initial conditions a user specified number of e-folds before the massless condition where $k^2/a^2 = M^2$ (where M is the largest eigenvalue of the mass matrix). There is also the function *ICs* which evaluates initial conditions using *ICsBE* (the before horizon exit function) and *ICsBM*, and takes the earliest one. The power spectrum and bipsectrum routines use this latter function.

One example is the potential:

$$V = \frac{1}{2}m_\phi^2\phi^2 + \frac{1}{2}M^2\cos^2\left(\frac{\Delta\theta}{2}\right)\left[\chi - (\phi-\phi_0)\tan\Xi\right]^2 \tag{5.1}$$

where

$$\Xi = \frac{\Delta\Theta}{\phi}\arctan[s(\phi-\phi_0)] \tag{5.2}$$

which is from Ref. [26] and is in the *Examples/LH/* folder with some simple scripts to those discussed above for the double quadratic potential for users to play with. This example is also discussed at length in Ref. [1].

C. Further examples

Also in the *Examples/* folder is another light field example with more interesting dynamics than the double quadratic example which we refer to as the axion quartic model. This example is again accompanied with scripts and plots for users to explore and was also discussed in Ref. [1]. It is in the *QuadAx* folder, and has the potential

$$V = \frac{1}{4}\lambda\phi^4 + \Lambda^4\left(1-\cos\left(\frac{2\pi\chi}{f}\right)\right). \tag{5.3}$$

Finally in the *Examples/* folder and discussed in Ref. [1] is a single field example with a step in the potential:

$$V = \frac{1}{2}m^2\phi^2\left(1 + c\tanh\left(\frac{\phi-\phi_0}{d}\right)\right) \tag{5.4}$$

which is in the folder *SingleField/* and was discussed in Refs. [9, 10], as well as in Ref. [1] .

VI. THINGS THAT CAN GO WRONG

While using the PyTransport package some issues have presented themselves which it might be useful for new users to know about. Many more have been ironed out, but it is neither practical nor desirable to make the code fully immune to misuse. Below we detail a few common problems we have faced.

A. Potential computation fails

This is the most severe bug/problem we have found, but also the least common.

The function *PyTransScripts.potential(V)* takes a potential V written in SymPy format and calculates and then simplifies its derivatives using the function *sympy.simplify*. As discussed at this reference[6], however, this is not always

[6] http://docs.sympy.org/latest/tutorial/simplification.html

the best way to simplify an expression and can take some time to complete the simplification for complicated functions. For example the simplification process takes a relatively long time (tens of seconds) for the heavy field model above.

We found a more serious problem occurred when looking at the example:

$$V = V_0 \left(10 - \sqrt{2\epsilon_s} \arctan\left(\frac{\phi}{\chi}\right) + \frac{1}{2} m_\chi^2 \left(\sqrt{(\chi^2 + \phi^2)} - 2 \right)^2 \right) \tag{6.1}$$

which represents a semi-circular valley in field space. For this example, there appears to be a *simpy.simplify* bug which made it crash with the potential written in the form given above, when the powers where written as doubles – i.e. as 2.0 – rather than simply as 2. The problem appears when taking cross derivatives. Uncommon errors such as this are something users might need to watch out for.

B. Make sure latest module version is imported

A more common problem is that if we wish to update a complied PyTrans*** module, for example after altering the tolerances, then after recompiling the module we need to ensure the new module is imported. The only reliable way to do this seems to be to restart or open a new Python session and then use the import command. If working in the Canopy editor, for example, this can be achieved by selecting "restart kernel" from the "run" menu.

C. Selecting the absolute and relative tolerances

The evolution of the three point function can be a numerically intensive task, requiring high numerical accuracy. The question arises how low (the lower the higher the accuracy) do we need to set numerical tolerances. This question can't be answered absolutely, and must be dealt with on a model by model basis. Models with finer features in the potential, or in which the excitation of the two and three point function occurs on sub-horizon scales will require higher tolerances. Models which produce a small signature may also need higher accuracy to resolve the true answer from noise than models which produce a large bispectrum.

Convergence is the key criterion in selecting tolerances. If one calculates the evolution of the three-point function of ζ for a example run with a given potential, reduce the tolerances and run again without the answer changing in any significant manner, then the tolerances are likely to be sufficient. As a rule of thumb 10^{-8} for the absolute tolerance and 10^{-8} for the relative tolerance is usually sufficient. However, some examples do require lower values. As the values are lowered, the code takes longer to run and eventually will fail (described below). Therefore, there is significant benefit for picking a required accuracy which is sufficient for the task, but not one which is too stringent. Those with experience of solving ODEs numerically will be familiar with this problem, which is of course more general than the specific integration at hand.

D. Integration stalling or failing

The consequences of picking a tolerance that is too demanding can be that the integrator will not finish. But this can also be a consequence of setting silly initial conditions or parameter choices. It is always a good idea once you begin with a new model of inflation to build up gradually. First integrate the background. Even here it is possible for the code to take a long time. For example, if the final time is after the end of inflation the code will try and track all the oscillations of the field. As these increase exponentially with e-fold it can be very time consuming. Once the background is giving sensible output, move onto integrating the correlations, initially just for a single k value of the power spectrum or single triangle of the bispectrum. If everything looks good then run over many values to calculate the power or bi-spectrum. If you feel you are waiting too long try just asking the code to evolve a short e-folding time (.1 say). Typically a single triangle of the bispectrum evaluated before the end of inflation will take from between a fraction of a second to half a minute to run with $4 - 6$ efolds of sub-horizon evolution, depending on the required accuracy. Heavy field models however typically have many more sub-horizon efolds given that the massless condition is already met only deep inside the horizon. If the run time seems to be much longer than you expect, double check you are working with the correct potential for your initial conditions and parameter choices.

E. Integration failing

If the code can't reach the accuracy demanded by the user the rkf45 routine will stop running and issue an error message that the required accuracy couldn't be reached.

F. Not enough efolds before horizon exit

A requirement that needs to be met in order to get accurate power spectra and bispectrum is that all the k values involved in a given correlation must be sufficiently deep inside the horizon initially for the initial conditions to be accurate. Since the functions which find the initial conditions for the two and three point evolutions take a background trajectory as their input, this trajectory must have enough e-folds prior to the exit of the k values of interest so that the correct initial conditions can be found.

Moreover, we must choose how many e-folds these k values stay sub-horizon for. As described above we can either measure backwards from horizon crossing itself or from the massless condition (only suitable for models with heavy fields) or pick the earliest of the two conditions. Normally 4-5 e-fold from these points is about right. But as with the setting of tolerances, the only way to tell for sure is to demand convergence. Too little run in time will lead to spurious oscillations in the spectra.

G. Not enough efolds after horizon exit

For single field models or effective single field models, such as models additional heavy fields, the correlations of ζ become conserved after horizon crossing. Likewise if there are multiple additional light fields which then decay, leading to adiabatic evolution, ζ also becomes conserved. In the former case conservation only occurs a few e-folds after horizon crossing, and in the latter the process also takes a few e-folds to complete. We must ensure, therefore, that the statistics are evaluated at a time when ζ has become conserved if that is what is intended, and so our evaluation time must be sufficiently far after horizon crossing or field decay.

H. Computer crashes and data loss

The PyTrans*** compiled module returns data in memory to Python. Moreover, the functions provided to calculate power spectra and bispectra which loop over the complied module also return arrays of data in memory to the calling process. This is true even for the MPI functions. If anything goes wrong before these functions complete the data is lost. For example if the computer crashes, or if using distributed computing any one of process crashes. These modules can be easily edited by users to instead write data to disk periodically, particularly advisable for distributed computing. We have not included this functionality to avoid overly complicated functions, and in any case, we anticipate that different users will have different needs, but it is an issue to look out for.

VII. SUMMARY

We have presented the PyTransport package for the calculations of inflationary spectra. This package compliments and extends currently available tools, and is also complimentary to two related packages mTransport [24] and CppTransport [1, 2] all described at this website[7].

Through use of a detailed example we have shown how PyTransport can be used in practice. We have also summarised the structure of the code, with some more details provided in the appendices.

Acknowledgments

DJM is supported by a Royal Society University research fellowship. This work would not have been possible without the related collaboration with David Seery, Mafalda Dias and Jonathan Frazer. We also thank John Ronayne

[7] https://transportmethod.com

for vital feedback and ongoing work on future releases.

Appendix 1: Base code description

We now give a brief description of the C++ code which is the core of PyTransport.

The folder *PyTransportDist/PyTransport/CppTrans/* contains the base C++ code which the complied Python module uses. There are five header files in this directory. The biggest is the *model.h* file. This contains the model class, which defines the properties of an inflationary model. It contains member functions which allow the calculation of important properties of the inflationary model, from simple objects such as the value of the Hubble rate, to more complex ones such as the A, B, C tensors which define the third order action, as discussed in Ref. [1]. It also calculates the "N" tensors, mentioned in the main text, that determine how field fluctuations are related to ζ. The equations of motion for the transport system are written in terms of "u" tensors (see section 5 of Ref. [1]), and there is also a member function of the model class which calculates these tensors. As noted in the main text we rescale the field velocity perturbation for performance reasons using the *scale* member function of the model class. This rescaling appears in the member functions which calculate u_2 and u_3 tensors and the N tensors, altering them slightly from the from given in Ref. [1]. The output of these functions is also divided by the Hubble rate, changing the time variable from cosmic time (which the equations are written in in Ref. [1]) to N, e-fold time. The scaling we choose essentially means the initial conditions for correlations of the velocity perturbations are of the same order of magnitude as correlations of field perturbations, helping the performance of the integrator.

The *moments.h* file contains classes which define properties of the two and three point functions of field perturbations. This class is not extensively used by the Python code, but the constructor of these classes allows an instance of the two/three-point function to be created containing the values these objects take deep inside the horizon – ie the initial conditions needed by the code. Once again these initial conditions are rescaled using the *scale* member function of the model class so that the initial conditions match the evolved correlations.

The *potential.h* file contains a class which defines the properties of the potential, and importantly has member functions which provide the derivatives of V. There is no barrier to changing this file by hand for a new potential. But the directory also contains *potentialProto.h*, which is a file that contains the skeleton of the potential.h file, and is used to automatically generate a potential.h file by the *PyTransSetup.potential()* function, detailed in appendix 1.

The the *evolve.h* file contains functions which define the evolution equations for three systems in the form needed by the rk45 ODE stepper we use: the background only system, the two point function coupled to background, and three point coupled to two point coupled to background. The system of equations of the latter two systems are given in Eqs. (5.5) and (5.16) of Ref. [1]). These equations use the "u" tensors which are also described in section 5 of Ref. [1] and which are calculated by the *model.h* class.

Finally there is a folder called rk45 stepper which contains the code for the ODE solver routine we use.

Appendix 2: Setup functions

The functions which are part of the PyTransSetup modules.

- **PyTransSetup.directory()** takes no arguments and automatically edits the C++ file *PyTrans/PyTrans.cpp* so it knows about the absolute location in the file system of the C++ files *potential.h*, *model.h* and *moments.h*. A user shouldn't need to call this function themselves, it is called by the compile functions below.
- **PyTransSetup.pathSet()** takes no arguments and adds the location of *PyTrans* compiled module to the Python path. This function was called in the examples above.
- **PyTransSetup.tol(rtol, atol)** takes a relative tolorance and absolute tolorance provided by the user and edits the C++ code such that these are the ones compiled into the compiled Python module. It does this my making edits to the *PyTrans/PyTrans.cpp* file. If not run before compiling the values last used remain in the C++ code. This routine is used in the examples above.
- **PyTransSetup.potential(V,nF,nP)** takes a expression in SymPy format (V), which is a function of two SymPy arrays p and f, and integers nF and nP which are of length of the arrays p and f. The function then writes this potential and its derivates into the *PyTransport/CppTrans/potential.h* file (using the *potentialProto.h* file). A line in the *potential.h* file is updated every time this function is run to record the time of the last run.

- **PyTransSetup.compileName(Name)** takes the argument "Name" which must be a string, and complies the C++ code into a Python module called "PyTransName" using the file *PyTransport/PyTrans/moduleSetup.py* which it automatically edits. The module is placed in the folder *PyTransport/PyTrans/lib/Python/*. Note that this function also automatically makes edits to the *PyTrans/PyTrans.cpp* file. This function is intended to be used on Python 2 systems.

- **PyTransSetup.compileName3(Name)** performs the same tasks as the function above, but is intended to be used on Python 3 systems.

- **PyTransSetup.deleteModule(Name)** takes the argument name and deletes the compiled module *PyTransName*.

Appendix 3: Compiled module

The functions which are part of the complied PyTrans module are detailed below.

These functions essentially use the C++ code in the *CppTrans/* folder to perform various tasks. The code for the functions can be found in the file *PyTrans/PyTrans.cpp* file. This code should be clear to those who are familiar with C++. The only elements which might unfamiliar are those which convert the C++ arrays to numpy arrays (at input and output), and the part of the file which defines how distutils is to turn the code into a Python module. A guide to creating compiled Python modules which we followed can be found at this tutorial page[8].

One other point to note is that the code which defines the functions which evolve correlations invokes a scaling of the wavenumbers involved (the kscale variable in the code), which helps the integrator perform well for very different input values of the wave numbers. The rescaling is then reversed for output values which are returned to the python code calling the function.

Note the *** (Name) label needs to be added to the functions below if the module has been given a name, as described in the main text.

- **PyTrans.nF()** takes no arguments and returns an integer which is the number of fields present in the model.

- **PyTrans.nP()** takes no arguments and returns an integer which is the number of parameters used to define the model.

- **PyTrans.V(fields, params)** takes a numpy array of length nF (the number of fields) containing a set of field values, and a numpy array of length nP (the number of parameters) containing parameter values. It returns a double which is the value of the potential for these field values and parameters.

- **PyTrans.dV(fields, params)** takes a numpy array of length nF (the number of fields) containing a set of field values, and a numpy array of length nP (the number of parameters) containing parameter values. It returns a numpy array of length nF, which contains the first derivative of the potential.

- **PyTrans.dVV(fields, params)** takes a numpy array of length nF (the number of fields) containing a set of field values, and a numpy array of length nP (the number of parameters) containing parameter values. It returns a two dimensional numpy array of size nF by nF, which contains the second derivatives of the potential.

- **PyTrans.H(fields-dotfields, params)** takes an numpy array of length 2 nF (twice the number of fields) containing a set of field values followed by the field's velocities in cosmic time (field derivative wrt cosmic time), and an array of length nP containing parameter values. It returns as a double the value of the Hubble rate.

- **PyTrans.backEvolve(Narray, fields-dotfields, params)** takes a numpy array of times in efolds (N) at which we want to know the background value of the fields and field velocities. This must start with the initial time in e-folds (initial N) we wish to evolve the system from, and finish with the final value of N. It also takes a numpy array of length 2 nF which contains the background field and velocity values at the initial time, as well as a numpy array of length nP containing parameter values. It returns a two dimensional numpy array. This array contains the fields, and field velocities at the times contained in Narray. The format is that the array has 1 + 2nF columns, with the zeroth column the times (Narray), the next columns are the field values and field velocity values at those times.

---

[8] http://www.tutorialspoint.com/python/python_further_extensions.htm

- **PyTrans.sigEvolve(Narray, k, fields-dotfields, params, full)** takes a numpy array of times in efolds (N) at which we want to know the value of the two point function of inflationary perturbations. This must start with the initial N we wish to evolve the system from, and finish with the final N. It also takes a Fourier mode value (k), and the initial conditions of the background cosmology (field and field velocity values) at the initial time as a numpy array of length 2 nF, the parameters of the system as a numpy array of length 2 nP, and an integer "full" set to 0 or 1 (if another value it defaults to 1). The initial time and the initial field and field velocity array are used to calculate the initial conditions for the evolution of the two-point function. The function returns a two dimensional numpy array the zeroth column of which contains the times (Narray). If full=0 the next column contains the power spectrum of ζ at these times, and this is the final column (there are therefore only 2 columns in total). If full = 1 then there is in addition 2 nF + 2 nF * 2 nF columns. The first 2 nF of these columns contain the fields and field velocities at the time steps requested. Then the final 2 nF * 2 nF contain the elements of matrix Σ_r^{ab} (the power spectrum and cross correlations of the fields and field velocities). There are therefore [1 + 1 +2 nF +2 nF *2 nF] columns in total. The convention is that the element Σ_r^{ab} corresponds to the [1 + 2nF + a + 2nF x (b-1)]th column of the array (recall the columns start at the zeroth column – a and b run from 1 to 2 nF).

- **PyTrans.alpEvolve(Narray, k1, k2, k3, fields-dotfields, params, full)** takes a numpy array of times in efolds (N) at which we want to know the value of the three point function of inflationary perturbations. This must start with the initial N we wish to evolve the system from, and finish with the final N. It also takes three Fourier mode values (k1, k2, k3), the initial conditions of the background cosmology (fields and field velocities) at the initial time as a numpy array of length 2 nF, the parameters of the system as a numpy array of length 2 nP, and an integer "full" set to 0 or 1 (if another value it defaults to 1). The function returns a two dimensional numpy array, the first column of which contains the times (Narray). If full=0 the next four columns contains the power spectrum of zeta for each of the three k values input, and the value of the bispectrum of zeta for a triangle with sides of length of these k values. A total of [1+ 4] columns. If full =1, there are an additional 2*nF + 6 *2 nF* 2 nF + 2 nF * 2 nF * 2 nF columns. The first 2nF of these columns contain the fields, and field velocities at the time steps requested (the background cosmology). The next 2nF * 2nF of these contain the real parts of $\Sigma^{ab}(k_1)$ in the same numbering convention as above. Then the real part of $\Sigma^{ab}(k_2)$ and then the real parts of $\Sigma^{ab}(k_3)$, the following 3 * 2nF * 2nF columns are the imaginary parts of the $\Sigma^{ab}(k_1)$, $\Sigma^{ab}(k_2)$ and $\Sigma^{ab}(k_3)$ matrices. So for example if one wanted access to the $\Sigma_i^{ab}(k_2)$ element that would be the [1 +4 + 2nF + 4 * 2nF * 2nF + a + 2nF * (b-1)]th column of the file. The final 2nF * 2nF * 2nF columns of this file correspond to the $B^{abc}(k_1, k_2, k_3)$ matrix, such that the corresponding columns would be the [1 + 2nF + 6 * 2nF * 2nF + a + 2nF * (b-1) + 2nF * 2nF * 2nF * (c-1)]th columns (recall the columns start at the zeroth column – a, b and c run from 1 to 2 nF).

Appendix 4: Python Scripts

The functions which are part of the PyTransScripts modules are detailed below. The code which provides these scripts in the *PyTransScripts.py* file should be clear to those familiar with Python. The scripts are simply an indication of what is possible, and it is intended that users will modify them for their own purposes, or write their own, as well as using the ones provided.

- **PyTransScripts.ICsBE(NBExit, k, back, params, PyT)** takes in the number of e-folds before horizon exit of a scale k at which initial conditions for the evolution of correlations are to be set, the scale k itself, a numpy array containing the background cosmology (returned from *PyTrans.backEvolve*), the parameters of the model as a numpy array, and the PyTrans module being used. It returns a double, and an numpy array. The former is the starting number of efolds which are at least NBExit before exit, as measured with respect to the background trajectory, back. The array contains the initial value of the fields and field velocities at this starting time. This script simply runs through the elements of back to find the first one before the exit time minus NBExit. As such it requires back to have enough entries for the result to be useful (roughly 10 per e-fold is fine), as discussed in the main text.

- **PyTransScripts.ICsBM(NBMassless,k, back, params, PyT)** works like PyTransScripts.ICsBM, but instead of calculating initial conditions before exit time, it evaluates the time when $k^2/a^2 = M^2$ where M is the largest eigenvalue of the mass matrix of the potential (we call this the massless condition), and returns conditions more than NBExit before that time. This is useful for example with a heavy field, for which we need to ensure the approximation we use to fix initial conditions is accurate (which requires $k^2/a^2 \gg M^2$).

- **PyTransScripts.ICsBM(NB,k, back, params, PyT)** takes the same arguments as the two previous functions and calls each in turn, then outputs the number of e-folds and the fields and field velocities at the earliest time of the two. This is so that the start time can be set to either NB before exit or NB before the massless condition, whichever is earlier.

- **PyTransScripts.kexitN(Nexit, back, params, PyT)** takes an exit time in e-folds, a background evolution which runs though this time (returned from *PyTrans.backEvolve*), a set of parameters and the PyTrans module being used. It returns as a double the k mode that exited at the time Nexit. This routine uses NumPy spline routines to find the value of k at the exit time.

- **PyTransScripts.kexitPhi(PhiExit, n, back, params, PyT)** takes an exit value of one of the fields, and a number indicating one of the fields (in range from 1 to nF), a background evolution which runs through this field value, a set of parameters and the PyTrans module being used. It returns the k mode that exited at the that field value. This routine uses NumPy spline routines to find the right k.

- **PyTransScripts.pSpectra(kA, back, params, NB, PyT)**: takes a numpy array specifying a range of k, a background evolution, a set of parameters, a number of e-folds (before massless or before exit whichever is earlier) and the PyTrans module being used. It returns two numpy arrays. The first has the corresponding values of P_ζ (to the input array of k) at the end of the evolution, and the second the times taken to perform the integration for each element.

- **PyTransScripts.pSpecMpi(kA, back, params, NB, PyT)**: does the same as the function above but spreads the calculation across as many process as are active using Mpi4Py. The script which contains this function should be called using the the command "mpiexec -n N python Script.py", where N is the number of processes to be opened. The length of kA must be divisible by N. We recommend calling at least twice as many processes as cores are available so that cores that run processes which finish first don't simply remain idle. The function returns the two numpy arrays to the process with rank 0. Empty arrays are returned to the other processes.

- **PyTransScripts.eqSpectra(kA, back, params, NB, PyT)**: takes in the same information as the function above, and returns three arrays. The first has the corresponding values of P_ζ, the second the corresponding values of B_ζ in the equilateral configuration ($k1 = k2 = k3 = k$) at the end of the evolution, and the final the times taken to perform the integration for each element.

- **PyTransScripts.eqSpecMpi(kA, back, params, NB, PyT)**: does the same as the function above but spreads the calculation across as many process as are active using Mpi4Py. The script which contains this function should be called using the the command "mpiexec -n N python Script.py", where N is the number of processes to be opened. The length of kA must be divisible by N. We recommend calling at least twice as many processes as cores are available so that cores that run processes which finish first don't simply remain idle. The function returns the three numpy arrays to the process with rank 0. Empty arrays are returned to the other processes.

- **PyTransScripts.alpBetSpectra(kt,alpha, beta, back, params, NB, nsnaps, PyT)**: takes in a value of kt, and two numpy arrays defining a range of values of alpha and beta, as well as a background evolution, set of parameters and a number of e-folds before horizon exit or the massless condition is met. It also takes an integer nsnaps, and the PyTrans module being used. nsnaps tells the code at how many different times at which to provide output. The function returns six arrays. The first output array is a three dimensional numpy array containing B_ζ for k values corresponding to the values of alpha, beta and kt input, and at nsnaps different times between the start and finish time. The second array is also three dimensional and corresponds to $P_zeta(k_1)$ for these values and times, the next $P_\zeta(k_2)$ and the next $P_\zeta(k_3)$. If nsnaps is 0 or 1, the third dimension of these arrays is only 1 element long, and the values returned correspond to those at the end of the evolution. If it is greater than one the output is given at evenly spaced times until the end of the evolution. This allows us to see how a slice through bispectrum evolves with time if we wish. The fifth array is two dimensional and corresponds to the times taken to perform the integrations associated with every combination of alpha and beta. The last array is the times at which the nsnaps are taken.

- **PyTransScripts.alpBetMpi(kt,alpha, beta, back, params, NB, nsnaps, PyT)**: does the same as the function above but spreads the calculation across as many process as are active using Mpi4Py. The script which contains this function should be called using the the command "mpiexec -n N python Script.py", where N is the number of processes to be opened. The length of alpha must be divisible by N. We recommend calling at least twice as many processes as cores are available so that cores that run processes which finish first don't simply

remain idle. The function returns the five numpy arrays to the process with rank 0. Empty arrays are returned to the other processes.

---

[1] M. Dias, J. Frazer, D. Mulryne, and D. Seery (2016).
[2] D. Seery (2016).
[3] J. M. Maldacena, JHEP **0305**, 013 (2003), astro-ph/0210603.
[4] D. Seery and J. E. Lidsey, JCAP **0506**, 003 (2005), astro-ph/0503692.
[5] X. Chen, M.-x. Huang, S. Kachru, and G. Shiu, JCAP **0701**, 002 (2007), hep-th/0605045.
[6] J. Elliston, D. Seery, and R. Tavakol, JCAP **1211**, 060 (2012), 1208.6011.
[7] G. Rigopoulos and E. Shellard, Phys.Rev. **D68**, 123518 (2003), astro-ph/0306620.
[8] D. H. Lyth, K. A. Malik, and M. Sasaki, JCAP **0505**, 004 (2005), astro-ph/0411220.
[9] X. Chen, R. Easther, and E. A. Lim, JCAP **0706**, 023 (2007), astro-ph/0611645.
[10] X. Chen, R. Easther, and E. A. Lim, JCAP **0804**, 010 (2008), 0801.3295.
[11] D. K. Hazra, L. Sriramkumar, and J. Martin (2012), 1201.0926.
[12] H. Funakoshi and S. Renaux-Petel (2012), 1211.3086.
[13] J. Elliston, D. J. Mulryne, D. Seery, and R. Tavakol, JCAP **1111**, 005 (2011), 1106.2153.
[14] D. J. Mulryne, D. Seery, and D. Wesley, JCAP **1001**, 024 (2010), 0909.2256.
[15] D. J. Mulryne, D. Seery, and D. Wesley, JCAP **1104**, 030 (2011), 1008.3159.
[16] M. Dias and D. Seery, Phys.Rev. **D85**, 043519 (2012), 1111.6544.
[17] G. J. Anderson, D. J. Mulryne, and D. Seery, JCAP **1210**, 019 (2012), 1205.0024.
[18] D. Seery, D. J. Mulryne, J. Frazer, and R. H. Ribeiro, JCAP **1209**, 010 (2012), 1203.2635.
[19] D. J. Mulryne, JCAP **1309**, 010 (2013), 1302.3842.
[20] M. Dias, J. Elliston, J. Frazer, D. Mulryne, and D. Seery, JCAP **1502**, 040 (2015), 1410.3491.
[21] I. Huston and A. J. Christopherson, Phys.Rev. **D85**, 063507 (2012), 1111.6919.
[22] L. C. Price, J. Frazer, J. Xu, H. V. Peiris, and R. Easther, JCAP **03**, 005 (2015), 1410.0685.
[23] D. Salopek, J. Bond, and J. M. Bardeen, Phys.Rev. **D40**, 1753 (1989).
[24] M. Dias, J. Frazer, and D. Seery (2015), 1502.03125.
[25] J. S. Horner and C. R. Contaldi, JCAP **1409**, 001 (2014), 1311.3224.
[26] X. Gao, D. Langlois, and S. Mizuno, JCAP **1310**, 023 (2013), 1306.5680.