LATTICEEASY

http://www.science.smith.edu/departments/Physics/fstaff/gfelder/latticeeasy

Gary Felder
gfelder@email.smith.edu
Clark Science Center, Smith College, Northampton, MA 01060

Igor Tkachev
Igor.Tkachev@cern.ch
Institute for Theoretical Physics, ETH,Hönggerberg, CH-8093, Zurich, Switzerland

January 21, 2008

Contents

Chapter 1

Overview

LATTICEEASY is a C++ program for doing lattice simulations of the evolution of interacting scalar fields in an expanding universe. The program is designed so that you can easily do runs with different parameters, and more importantly can easily introduce new models to evaluate. Starting with LATTICEEASY 2.0 you can do these simulations in one, two, or three dimensions simply by resetting a single variable. There is also a parallel processing version of LATTICEEASY called CLUSTEREASY.

The program is available on the Web at http://www.science.smith.edu/departments/Physics/fstaff/gfelder/latticeeasy. It is freely available to anyone who wants to use it or modify it. For details see chapter 8; basically it says that you can do whatever you want with the program as long as you continue to give us credit and leave our contact information with it so that people can contact us about it.

If you have any questions or comments about LATTICEEASY please email us at gfelder@email.smith.edu. We would love to hear how the program is working for you, and we would be happy to help with any questions you might have, bug reports, suggestions for future improvements, or anything else you might want to tell us about the program.

This document is divided into four main sections. Chapter 4 describes the use of LATTICEEASY. It tells you how to compile and run the program, how to set appropriate parameters for a given model, and finally how to create new models for the program to run. Chapter 5 describes the output of the program. It lists each of the output functions in the program and describes the files they create and how to interpret their results. Chapter 6 derives and explains the equations used by the program. Much of this section isn't necessary for using the program, but is necessary for understanding what the program does and why. One of the most important parts of this section deals with the variable rescalings used by the program. In order to simplify the equations the program uses rescaled values for fields and spacetime variable, and these rescalings are derived and explained in chapter 6. These rescalings are used throughout chapters 4 and 5, so you may find that some parts of these sections make more sense after you have read chapter 6. Nonetheless, these first two sections contain all the information you need to get started using the program. Chapter 7 describes the parallel processing version of LATTICEEASY.

There are a few minor sections in addition to these major ones. Chapter 2 describes our notation and conventions. Chapter 3 lists the files in LATTICEEASY and briefly describes what each of them does. Chapter 8 gives the terms of use for the program. This "credit" section is also reproduced in the comments at the top of `latticeeasy.cpp`, the main file of the program.

Chapter 2

Notation and Conventions

Bare values are given in Planck units where $M_p \approx 1.22 \times 10^{19} GeV$. The FRW metric is taken to be $g_{\mu\nu} = diag\{1, -a^2, -a^2, -a^2\}$ meaning bare time is measured in physical units and bare distances are measured in comoving coordinates. Variables that have been rescaled in the program are denoted by the subscript pr. Repeated indices are summed unless otherwise indicated. Overdots indicate differentiation with respect to the bare time t while primes indicate differentiation with respect to the program time t_{pr}.

Generic properties of fields are indicated by using f for the field variable. Subscripts such as f_i will be used only where needed for clarity. F_k and f_k refer to continuous and discrete Fourier transforms of f respectively. These are discussed at greater length in section 6.3.

The scale factor and time are initially set to 1 and 0 respectively for convenience.

Frequently the documentation refers to files included in the LATTICEEASY distribution, or to variables and/or functions in the program. These file names are written in a typewriter font, as in `latticeeasy.cpp`. Output files created by the program are printed in the same way. The functions and variables in the program are also in typewriter font, but they have also been slanted, as in *`nflds`* and *`potential_energy()`*. (An exception to this is when program variables appear in equations, in which case they are displayed in the usual equation format for readability.) Functions will always be indicated with parentheses, but the arguments to the function will generally not be included.

Chapter 3

The LATTICEEASY Files

LATTICEEASY consists of the following files

latticeeasy.cpp Contains the *`main()`* function for latticeeasy as well as the declarations of global variables (all of which are declared as extern in `latticeeasy.h`).

initialize.cpp Contains the function *`initialize()`*, which performs all initialization tasks. These include setting initial conditions for field fluctuations, setting the derivative of the scale factor to its initial value, and initializing various run parameters.

evolution.cpp Contains the functions for incrementing the fields, the field derivatives, and the scale factor and its derivative. This file also contains the function for calculating the gradient energy of the fields.

output.cpp Contains all the functions for calculating derived quantities and saving them to files. These quantities include field means and variances, the scale factor and its derivatives, components of energy density, spectra, and histograms. This file also contains a function for periodically saving an image of the lattice.

ffteasy.cpp Contains code for performing Fourier transforms. FFTEASY is itself a stand-alone set of routines written by Gary Felder and available at http://www.science.smith.edu/departments/Physics/fstaff/gfelder/ffteasy.

latticeeasy.h Contains definitions and declarations used throughout the program.

parameters.h Contains all the adjustable parameters for doing runs. These include variables such as grid size and time step, as well as variables determining what output to save and how often to do it.

model.h Contains the information specific to a particular model. A library of these files is included in the directory "models." Any one of them can be copied to the program directory and renamed `model.h` to run that model.

makefile A makefile for compiling the program. By default it is set to use g++ but this can easily be modified.

Chapter 4

Using LATTICEEASY

The basic structure of the program is as follows: There is a file called `model.h` that contains all the information specific to a particular model. For example, the model file that we include by default with the LATTICEEASY distribution encodes a model with two scalar fields ϕ and χ with the potential

$$V = \frac{1}{4}\lambda\phi^4 + \frac{1}{2}g^2\phi^2\chi^2. \qquad \text{(TWOFLDLAMBDA model)} \tag{4.1}$$

To create a new model for the program to run you need to create a new model file. We explain below how to do this.

To run the program you will need to set the parameters for your run (grid size, time step, etc.), compile the program, and run. All of the adjustable parameters are in the file `parameters.h`. So the only two files that you as a user should ever need to modify are `model.h` and `parameters.h`. Of course you are free to examine the rest of the program and make any modifications to it you wish, as long as you leave the credits paragraph at the beginning of `latticeeasy.cpp` intact. The program is heavily commented so it should be possible for anyone with a good amount of programming experience to understand what the different parts are doing.

The following sections describe in turn how to run the program for a particular model and how to incorporate new models into the program.

4.1 Running the Program With an Existing Model

To set the program up to run with a particular model you copy the header file for that model to the directory with the other source files and name it `model.h`. Near the top of each model file there is a commented out list of adjustable parameters for that model. You could just uncomment these and run the model like that, but we find it is easier to have all the adjustable parameters in one place, i.e. the file `parameters.h`, so we recommend leaving these lines commented out in `model.h` and copying them (uncommented) to `parameters.h`. There is a spot in `parameters.h` marked for this purpose.

In order to do a run with LATTICEEASY you set the parameters in the file `parameters.h`, compile the program, and run it. This sounds simple until you realize that there are over forty parameters to be set, not including parameters specific to the particular model you are running. Relax, though, most of these are very straightforward. In fact most of these adjustable parameters simply tell the program what kinds of output you want it to create.

We describe how to run the program in the following sections. Section 4.1.1 talks about compiling the program. Section 4.1.2 discusses running the program, and in particular discusses how to continue previous runs to later times. Section 4.1.3 describes all the parameters in `parameters.h`. Section 4.1.4 has suggestions on choosing reasonable values for quantities like the grid size and the time step. Finally, section 4.1.5 explains how to do runs with double precision real numbers.

4.1.1 Compiling

A makefile is provided with the program and it should work for any UNIX system with g++, the GNU C++ compiler. If you are using a different compiler you can directly compile the files, or simply edit the makefile as needed. For example you can change the variable "COMPILER" at the top of the makefile to something other than "g++." We recommend that you leave optimization on. The makefile is currently set to create an executable called "latticeeasy," which can then be run in the same directory as the source code.

Warning: There is a bug in some old versions of the GNU compiler that can sometimes cause the program to crash. If you are using g++ and the program exits with a segmentation fault, try checking your version of the compiler by running g++ -v. If the result you get includes "egcs-1.0.3 release" then you are using the buggy version. Unfortunately this is the version of g++ that shipped with RedHat Linux 5. The best fix is to update your compiler. The latest versions of g++, including the ones that come with RedHat 6, don't have this problem. If you are interested in more information about this bug you can email Gary Felder at gfelder@email.smith.edu.

4.1.2 Running the Program and Continuing Old Runs

Running the program simply involves running the executable that you created when you compiled it. There are no command line parameters, so you can just type "./latticeeasy" and the program will run and create all of its output files in the current directory.

Sometimes, however, you might want to continue a run that you've already done. Suppose, for example, that you ran the program to $t = 100$ and then decided you wanted to see what happens at later times. This is possible if you have set ***`scheckpoint`*** to save a grid image for the first run. (See below.) Simply edit `parameters.h` so that $tf = 200$ (or whatever you want it to be), set ***`continue_run`*** to be one or two, and rerun. If $continue\_run = 1$ then the program will append the new data to the output files from the previous run. If $continue\_run = 2$ then new files will be created with a different extension. The default extension for output files is "_0.dat," but if you choose to create new output files for a run continuation they will have the extension "_1.dat" for the first continuation, "_2.dat" for the next one, and so on.

If you set the program to continue a run and there is no grid image file in the current directory it will simply begin a new run. If the program detects a grid image file in the directory with a stored time at or after the value of ***`tf`*** it will simply give you a warning message and exit.

Note that if you do a run to $t = 100$ and then continue it to $t = 200$ the resulting output won't necessarily be exactly the same as if you had run it directly to $t = 200$. Assuming you left the parameter ***`noutput_times`*** at 1000 for all of the runs the program would generate means and variances at roughly 1000 times from 0 to 100 and then another 1000 from 100 to 200, so doing the run in two stages will produce twice as many output times as doing it in one step. You can change this behavior by adjusting the number of output times per run. The actual field evolution should be the same either way.

4.1.3 The Adjustable Parameters

There are three types of adjustable parameters in `parameters.h`. The first are general parameters like the grid size, the second are model-specific parameters that have been copied in from a model file, and the third are parameters controlling the output of the program. We can't document the model-specific parameters because they are different for each model, but we describe here the other two groups.

The general parameters are

NDIMS The number of dimensions. This can be set to 1, 2, or 3.

N The number of grid points per edge. The total number of points on the lattice is N^{NDIMS}.

nflds The number of fields. Some models (e.g. the TWOFLDLAMBDA model described below) require a specific number of fields. In this case it is up to the model to check that the correct number of fields has been set when running that model. See section 4.2.2.

L The size of the box in rescaled distance units. The volume of the box is L^{NDIMS}.

dt The size of the time step.

tf The final time up to which the program should run.

seed The seed for random number generation. This can be changed to produce two runs with different random initial distributions. (The random numbers are only used for initial conditions.)

initfield[] An array of initial homogeneous field values in rescaled units. If there are fewer elements in the array than there are fields then all unspecified elements are set to zero. See section 6.3.1.

initderivs[] An array specifying initial homogeneous field derivatives. Here too unspecified values are set to zero. See section 6.3.1.

expansion The kind of expansion to use. This can be set to zero for no expansion (Minkowski space), one for a fixed power-law expansion, or two for a self-consistent calculation of the expansion coupled to the field evolution.

expansion_power This parameter only has an effect if $expansion = 1$. In this case `expansion_power` is the exponent of time in the fixed power-law expansion. Note that this exponent is given in physical, not rescaled units. For example it should be .5 or .67 for radiation or matter domination respectively. See section 6.2.3.

continue_run This determines whether the program should start a new run (0) or look for a grid image file to continue an old run. If it does continue an old run this variable also determines whether new data will be appended to the old data files (1) or written in new files (2).

The following are the parameters for controlling the output of the program. Note that all parameters that control a binary option (e.g. whether or not to calculate spectra) use 1 for yes and 0 for no. The output functions and the files they create are described in detail in section 5.

noutput_flds The number of fields for which you want output (means, variance, spectra, etc.) to be generated. Set this to zero and the program will automatically generate output for all fields.

alt_extension By default the program creates output files with the extension "_0.dat" or "_<run_number>.dat" where "run_number" indicates whether or not the run is a continuation of a previous one. If you set this string to be anything other than empty then the program will use it as the extension for the output files instead.

noutput_times The number of times during the run to call the `save()` function that calculates and outputs derived quantities. Means and variances of fields as well as the scale factor and its derivatives are calculated every time `save()` is called. Other quantities like spectra are calculated less frequently (see below).

print_interval At evenly spaced intervals of clock time the program writes the current program time to an output file and, optionally, to the screen. This parameter determines the interval in seconds between these updates.

screen_updates Whether or not to write time updates to the screen.

checkpoint_interval How often (in program time units) to perform infrequent calculations and to save an image of the lattice. The infrequent calculations include spectra, components of energy, histograms, and slices.

store_lattice_times[] An array of values giving times (in program units) at which to create new files for lattice images. By default the program creates a grid image file and stores the current version of the lattice in it every time it checkpoints. If there are no entries in this array then at the end of the run the only grid image will thus be of the lattice at the last time calculated. If there are entries in this array, however, then a separate file will be created with an image of the grid at each specified time, as well as at the end of the run. See section 5.9.

smeansvars Whether or not to save means and variances of the fields. See section 5.2.

sexpansion Whether or not to save the scale factor and its derivatives. This option is ignored unless $expansion = 2$. See section 5.3.

smodel Whether or not to call a model-specific output function. See section 5.10.

t_start_output This sets a time at which to start recording infrequent output. Actually the start time can be set for each output routine separately, but by default they are all set to begin at this time so that they can easily be changed all at once.

scheckpoint and tcheckpoint Whether to save an image of the lattice (and when to start doing so).

sspectra and tspectra Whether or not to save spectra of the fields (and when to start). See section 5.4.

senergy and tenergy Whether or not to save the components of the energy density and a check of overall energy conservation (and when to start). See section 5.5.

shistograms and thistograms Whether or not to generate histograms of the field values for each field (and when to start). See section 5.6.

- **nbins** The number of bins to use in the histograms
- **histogram_min and histogram_max** Upper and lower bounds for the histograms. If these are set equal to each other then the histograms automatically cover the range of current field values.

shistograms2d and thistograms2d Whether or not to generate two-dimensional histograms of pairs of fields (and when to start). See section 5.7.

- **nbins2d** Default number of bins to use in each field direction.
- **nbins0** Number of bins to use in the first field direction.
- **nbins1** Number of bins to use in the second field direction.
- **histogram2d_min and histogram2d_max** Upper and lower bounds for the histograms. If these are set equal to each other then the histograms automatically cover the range of current field values.
- **hist2dflds[]** A list of the pairs of fields for which two-dimensional histograms should be generated. This list should always contain an even number of elements.

sslices and tslices Whether or not to save the field values on a slice through the lattice (and when to start).

- **slicedim** The number of dimensions for the slice. If $slicedim \geq NDIMS$ then the whole lattice is output. Note that for a three dimensional lattice this could generate very large ASCII files.
- **slicelength** The number of points (in each dimension) to include in the slice.
- **sliceskip** The number of points between adjacent points in the slice. Set $slicelength = N$ and $sliceskip = 1$ to include the full slice.

4.1.4 Choosing Run Parameters

The values you set for the parameters described above will of course depend on the physical features of your model, the information you are interested in, and the amount of computing power available to you. For example, increasing the number of grid points `N` while holding the box size `L` constant will result in better coverage of short-wavelength excitations, but it will also increase the time and memory requirements of your run. Remember that for a three dimensional run doubling `N` increases the duration and size in memory of your run by a factor of 8! Because of the Fourier transform routines used `N` must always be a power of two.

The first and most important guide in choosing parameters should be your understanding of the physics of your model. Try to figure out the typical wavelengths of fluctuations that you expect to be excited and make sure those wavelengths are comfortably in between the grid spacing L/N and box size `L` of your run. Make sure `dt` is smaller than the smallest important characteristic period of your problem.

Once you've done that, the rest becomes a process of trial and error. If you set the time step too large then you will probably get exponentially growing solutions. Try a run with a moderately large time step and see what happens, and then try incrementally decreasing it until you get a sense of how small you have to make it. Of course you don't want to take the largest value of `dt` that doesn't lead to disaster! Rather you want to keep decreasing the time step until you are satisfied that the results of your run are not changing significantly in response to those decreases. You should do this initial trial and error on a very small grid like $N = 16$ or even $N = 8$. Of course you will want to check again that the time step you pick is still working well with a larger grid, but in our experience the appropriate value of the time step is generally about the same for different size grids, so you can save yourself a lot of time by doing most of the trial and error with very quick runs.

Note that if you make the time step too large you will violate the Courant stability condition $dt < \frac{dx}{\sqrt{NDIMS}}$ [3], in which case the program will simply print a warning and exit.

A good way to test for appropriate values of `N` and `L` is to look at spectra. Hopefully the spectra of your fields will vanish in the ultraviolet once the interactions become strong. If not that may be a sign that you are missing significant physics because of your ultraviolet cutoff and you should either decrease `L` or increase `N`. One problem with this test is figuring out which spectrum contains the interesting physics for your problem. Requiring that $k^2|f_k|^2$ vanish at large k is a much more stringent requirement than requiring that $|f_k|^2$ vanish at large k. Here you will have to be guided by the relevant physics for your model.

Remember when choosing `N` and `L` that your box in position space is equivalent to an array in frequency space (via a Fourier transform). Specifically, the array covers values of k (frequency) from 0 to $\pi\frac{N}{L}$ with a spacing in k space of $\frac{\pi}{L}$. So increasing `N` while holding `L` fixed increases your range in k space while increasing `L` and `N` together increases your resolution in k space. These facts suggest another powerful, but time-consuming test. You can do a series of runs with different values of `N` and `L` and compare all the kinds of output for the run. For example, if you do three runs with $N_1 = 64$, $L_1 = 1$, $N_2 = 64$, $L_2 = 2$, $N_3 = 128$, $L_3 = 2$ then run 3 has the same ultraviolet cutoff as run 1 and the same infrared resolution as run 2. If all the results of runs 1 and 3 look the same then you can conclude that adding infrared resolution didn't change anything and run 1 probably had adequate resolution in Fourier space. Conversely if runs 2 and 3 give nearly identical results you can conclude that run 2 probably had adequate coverage of ultraviolet modes. By doing a series of tests like this you can figure out what ultraviolet range and infrared resolution is physically important for your problem and make sure you are sampling it adequately. Of course this method isn't infallible. In the example above, if the relevant physics is happening at wavelengths of 10^{-6} then none of your runs will see it and you won't know about it. But if you already expect on physical grounds that you are in the right ballpark then this method can be a good test.

4.1.5 Double Precision

By default LATTICEEASY uses single precision. While precision is technically compiler-dependent, most compilers use four bytes for a single precision real number, which typically corresponds to about seven or eight digits of precision. Our experience is that this is adequate for most physical problems, but sometimes you need more precision. This can happen, for example, if you are initializing one or more fields with a large homogeneous value and very small initial fluctuations. With single precision $1 + 10^{-10}$ will simply evaluate to 1, so to keep track of such small fluctuations you will need to use double precision.

To do a run with double precision simply insert

```
#define float double
```

into the files `latticeeasy.h` and `ffteasy.cpp`, just below the list of headers. This will cause the compiler to interpret the word "float" as if it were the word "double" throughout the program.

Using double precision generally has little or no cost in speed. The disadvantage is that it requires twice as much memory as a single precision run.

4.2 Writing a New Model

Ultimately the main reason you would want to use LATTICEEASY is presumably to do lattice simulations of your own favorite models. Writing a new model for the program requires the following steps, each of which is described in more detail in a later section:

1. Write down the equations for your model. You need to specify the field content and the potential, figure out what variable rescalings to use, and then work out a few quantities like $\frac{dV}{df}$ and $\frac{d^2V}{df^2}$. These instructions tell you step by step how to write these equations in the form expected by the program.

2. Write a *model file* for your model. The easiest way to do this is to copy an existing model file and modify it as needed. We describe below exactly what parameters and functions the model file should contain and how to specify them correctly. Once you've written the file simply copy it into the program directory with the name `model.h`.

Once you have the hang of it the entire process from deciding to run a particular model to having your simulations up and going should take no more than a few hours.

The following sections go through each of these steps in detail. For illustration purposes an example will be developed as we go. We chose as our example chaotic inflation with a $\lambda\phi^4$ inflaton potential and one other field coupled to the inflaton. The potential for this model is

$$V = \frac{1}{4}\lambda\phi^4 + \frac{1}{2}g^2\phi^2\chi^2. \qquad \text{(TWOFLDLAMBDA model)} \tag{4.2}$$

The TWOFLDLAMBDA model is the `model.h` file in the program directory in the distribution of LATTICEEASY. To avoid confusion all formulas in this section that are specific to this model will be marked (like the preceding equation).

4.2.1 Writing Down the Equations for a New Model

Variable Rescalings

The first thing you will need to decide about your model is how to rescale the variables. In general they are rescaled as

$$f_{pr} \equiv Aa^r f;\ \vec{x}_{pr} \equiv B\vec{x};\ dt_{pr} \equiv Ba^s dt. \tag{4.3}$$

You may set the variables A, B, and r to whatever you wish. The variable s must be set to $2r-3$ for the program to work correctly, so this setting is done automatically by the program. If you have in mind a particular set of rescalings that you think most simplifies your model then you can simply define these three variables appropriately. In general, however, there are a set of default values that should work well for any model where the dominant term is polynomial in the fields (or can be approximated as such). These default rescalings are derived and explained in section 6.1.2. Here we simply quote the results. Assuming the dominant term in your potential is of the form

$$V = \frac{cpl}{\beta}\phi^\beta \tag{4.4}$$

where ϕ is one of the fields in your problem, the default values for the rescaling variables are

$$A = \frac{1}{\phi_0};\ B = \sqrt{cpl}\,\phi_0^{-1+\beta/2};\ r = \frac{6}{2+\beta};\ s = 3\frac{2-\beta}{2+\beta} \tag{4.5}$$

where ϕ_0 is the initial value of the field ϕ. It is often most convenient to choose this value such that the derivatives of the homogeneous fields vanish initially, but this is not necessary.

For the TWOFLDLAMBDA model the dominant term is $\frac{1}{4}\lambda\phi^4$ so $\beta = 4$ and $cpl = \lambda$. This means that

$$A = \frac{1}{\phi_0};\ B = \sqrt{\lambda}\phi_0;\ r = 1;\ s = -1. \qquad \text{(TWOFLDLAMBDA model)} \tag{4.6}$$

From equation (4.5) the time derivative of ϕ in program variables is

$$\phi'_{pr} = \frac{1}{B}a^{-s}\dot{\phi}_{pr} = \frac{A}{B}\left(a^{r-s}\dot{\phi} + ra^{r-s-1}\dot{a}\phi\right) = \frac{a^2}{\lambda\phi_0^2}\left(\dot{\phi} + \frac{\dot{a}}{a}\phi\right). \qquad \text{(TWOFLDLAMBDA model)} \tag{4.7}$$

Before running this model on the lattice we solved the ODE for the evolution of a homogeneous field ϕ with $V(\phi) = \frac{1}{4}\lambda\phi^4$ near the end of inflation and determined that the expression above equals zero at $\phi \approx .342$. (Recall that all numbers are given here in Planck units.) So we set $\phi_0 = .342$ and for initial conditions set the homogeneous component of ϕ' to zero.

The Program Potential

The starting point for all the equations you need to enter in the program is the potential for your model, which in turn defines your field content. So the first expression you need to derive in program variables is the potential itself. The program potential is defined as

$$V_{pr} \equiv \frac{A^2}{B^2}a^{-2s+2r}V \tag{4.8}$$

where V is the physical potential of your model, expressed in terms of program fields. This is easiest to illustrate with an example, so continuing with the TWOFLDLAMBDA model we have

$$f_{pr} = \frac{a}{\phi_0}f \qquad \text{(TWOFLDLAMBDA model)} \tag{4.9}$$

so

$$V = \frac{1}{4}\lambda\phi^4 + \frac{1}{2}g^2\phi^2\chi^2 = \left(\frac{\phi_0}{a}\right)^4 \left(\frac{1}{4}\lambda\phi_{pr}^4 + \frac{1}{2}g^2\phi_{pr}^2\chi_{pr}^2\right) \qquad \text{(TWOFLDLAMBDA model)} \tag{4.10}$$

and

$$V_{pr} = \frac{a^4}{\lambda\phi_0^4}V = \frac{1}{4}\phi_{pr}^4 + \frac{1}{2}\frac{g^2}{\lambda}\phi_{pr}^2\chi_{pr}^2. \qquad \text{(TWOFLDLAMBDA model)} \tag{4.11}$$

Because the coupling constants g and λ appear in the program potential only in the combination $\frac{g^2}{\lambda}$ we choose for convenience to define a new variable $gl \equiv \frac{g^2}{\lambda}$.

The Potential Derivative $\frac{\partial V}{\partial f}$

The equations of motion for the fields all involve the term $\frac{\partial V}{\partial f}$. When these equations are derived in program variables in section 6.1 this term simply becomes $\frac{\partial V_{pr}}{\partial f_{pr}}$. So once you have written down the program potential all you have to do is take its derivative. For the TWOFLDLAMBDA model this gives

$$\frac{\partial V_{pr}}{\partial \phi_{pr}} = \left(\phi_{pr}^2 + gl\chi_{pr}^2\right)\phi_{pr} \qquad \text{(TWOFLDLAMBDA model)} \tag{4.12}$$

$$\frac{\partial V_{pr}}{\partial \chi_{pr}} = gl\phi_{pr}^2\chi_{pr}. \qquad \text{(TWOFLDLAMBDA model)} \tag{4.13}$$

Effective Masses

Finally, for setting the initial field fluctuations and later for calculating spectra of number density and energy density the program needs formulas for the effective masses of the fields. In program variables, these are given by (see section 6.3.2)

$$m_{pr}^2 = a^{2s+2}\frac{\partial^2 V_{pr}}{\partial f_{pr}^2}, \tag{4.14}$$

so you will need to calculate these second derivatives. For the TWOFLDLAMBDA model

$$\frac{\partial^2 V_{pr}}{\partial \phi_p r^2} = 3\phi_{pr}^2 + gl\chi_{pr}^2 \qquad \text{(TWOFLDLAMBDA model)} \tag{4.15}$$

$$\frac{\partial^2 V_{pr}}{\partial \chi_p r^2} = gl\phi_{pr}^2. \qquad \text{(TWOFLDLAMBDA model)} \tag{4.16}$$

4.2.2 Writing a Model File

Once you've written down the equations for your model as described in the previous section, writing the file to encode these equations for the program is quite simple. We strongly recommend that you start by copying an existing model file and then simply modify it as needed. You should always put a description of your model in the comments at the top of the model file as this makes it easy to refer to later as well as to show other people what you have done.

Probably the easiest way to read the rest of this section is to have a copy of the model file for the TWOFLD-LAMBDA model next to you as you read so that you can refer to it and see what each section we describe looks like.

Before going through the different parts of the model file, we should point out that at some points in the file you will see certain macros such as *FIELD*, *LOOP*, and *INDEXLIST* that are defined in `latticeeasy.h` in order to simplify the process of writing code that works in different numbers of dimensions. For the most part these have already been encoded and you will have to make little use of them in modifying the model files, however there may be some occasions when you would like to do something a little more involved and would want to use these macros. Section 4.2.3 describes all of the macros defined for this purpose.

After the initial comments the first section of the model file is a list of model-specific adjustable parameters. In the TWOFLDLAMBDA model, for example, this section consists of the parameters λ and gl (defined above). You could just put your model parameters at the top of your model file and leave them there, but if you ever expect to modify these parameters we recommend instead that you write them in a commented out section at the top of the model file. Then copy them (uncommented) to the file `parameters.h`. This way your model file contains all the information about your model, but when you are actually doing runs with it all of your adjustable parameters will be in one place.

The next section of the file contains the rescaling parameters A, B, and r described above. In the program they are called *`rescale_A`*, *`rescale_B`*, and *`rescale_r`*. If you want to set them to your own favorite values you can simply do so directly. Otherwise you can set the values ϕ_0, β, and *cpl* and let A, B, and r be set in terms of them. Of course if you are copying a model file where these defaults were reset you will have to put them back in. You can copy the definitions of A, B, and r in terms of ϕ_0, β, and *cpl* from the TWOFLDLAMBDA model file or simply write them in by hand; the equations are provided in comments just above the declarations of these parameters.

The next section of the file declares the array *`model_vars`*. This is a list of dynamic variables specific to your model. For most of our models this array is not used, but it is included to allow flexibility.

The last section before the functions is a list of macros for field variables. These are strictly optional. They are used to make the programs more readable. Their use should be clear from looking at any of our example files. Feel free to write appropriate ones for your model or not as you prefer.

The rest of this section describes the functions that must be defined in the model file.

modelinfo(file *info)

This function accepts a pointer to an open file stream and writes information about the model to that file. The function is called at the beginning of the run by the *`output_parameters()`* function that creates a file with general information about the run. You should put in this function the name of your model, a brief description (e.g. the potential), and values of model-specific parameters.

modelinitialize()

This function is mainly included for the sake of flexibility. It is called once just before the initial field values are set and once afterwards. The input parameter *`which_call`* tells the function which of those times it is being called; it is set to 1 or 2 for the different calls. You can use this function for any initializations required by your model or to check that appropriate constants have been defined (e.g. if your model requires a certain number of fields). If you want to use an initialization different from the usual quantum modes for the fields you can do so here and then set the global variable *`no_initialization`* to one, which tells the program not to initialize quantum modes. Alternatively you can let the program do its usual initialization and then when this function is called a second time (i.e. when *`which_call`*=2) you can add to or otherwise alter the initial conditions.

potential_energy(int term, float *field_values)

This function returns a term from V_{pr}, averaged over the lattice. You can choose to break the potential into separate terms in any way you wish. You can, for example, simply call the whole thing one term. The only difference this will make is that if you output the components of energy (see section 5.5) each term will be output separately. The number of terms must be given as the value of the constant *`num_potential_terms`*. For example, in the TWOFLDLAMBDA model this constant is set to 2. If the argument *`term`* is 0 the function returns $\langle \frac{1}{4}\phi_{pr}^4 \rangle$ and if *`term`* is 1 it returns $\langle \frac{1}{2}\phi_{pr}^2\chi_{pr}^2 \rangle$ where angle brackets denote lattice averages.

The second argument to the potential energy function is a pointer to a list of average field values. Normally during the run this pointer will be passed in as NULL, in which case the function does the grid averages described above. When setting the initial conditions, however, the program will call this routine before the field values on the lattice have been set. In this case it will pass in an array of values representing the initial homogeneous values of the fields. If the pointer that is passed in is not NULL then the function will calculate the potential terms using these values rather than by averaging over lattice values. The implementation of these options should be clear in the TWOFLDLAMBDA model file.

dvdf(int fld, INDEXLIST)

This function returns the value $\frac{\partial V_{pr}}{\partial f_{pr}}$ for a field fld at a grid point specified by the indices in *INDEXLIST*. (See below.) If you look at this function for the TWOFLDLAMBDA model you will see that equations (4.12) and (4.13) have simply been copied in.

effective_mass(float mass_sq[], float *field_values)

As explained in section 6.3.2 the effective mass used for initial conditions and for calculating spectra is given in program units by

$$m_{pr}^2 = a^{2s+2} \frac{\partial^2 V_{pr}}{\partial f_{pr}^2}. \tag{4.17}$$

This function calculates the grid average of $\frac{\partial^2 V_{pr}}{\partial f_{pr}^2}$ for each field and stores this square mass in the array *mass_sq*. Once again, you can look at the example file and easily see the equations from section 4.2.1. As with the potential energy function the effective mass function accepts a pointer to an array of field values. If this pointer is not NULL then these values should be used instead of the field values on the lattice.

In most of our model files the first part of the *effective_mass()* function consists of the calculation of the average squared values of the fields, either in terms of the lattice values or the array of field values passed to the function. Typically the derivatives $\frac{\partial^2 V_{pr}}{\partial f_{pr}^2}$ depend on these squared values. If this is the case for your model then you can leave this part of the function unchanged. The next part sets the square masses in terms of these values; these equations are the part of the function unique to each model. Finally the prefactor a^{2s+2} is added in. This last part should be the same for all models.

model_output(int flush, char *ext_)

This function can be used to generate outputs specific to your model. We have found the generic output functions that accompany LATTICEEASY to be sufficient for most purposes, so most of our model files leave this function empty. However, it provides flexibility in creating new models. This function is also used if you want to set a rescaling for the other output functions to use. See section 5.10 for more details.

4.2.3 Macros for Referencing Field Variables in Different Numbers of Dimensions

As we mentioned above, LATTICEEASY is designed so that you can simply set the parameters of your run, including the number of dimensions, without having to worry about the detailed way these options are implemented. One way in which this is done is through the use of certain macros for referring to fields and/or indices in different number of dimensions. Typically the only place you would need to use any of these macros in creating a new model file would be when you create your own names for the fields, using the predefined macro *FIELD*. For example, `twofldlambda.h` contains the line
#define PHI FIELD(0)
which tells the program to take "PHI" to refer to the 0'th field value at a particular point.

In some circumstances, however, you might want to do something more complicated with your model file. For example, if you are creating output functions specific to your model you may need to use loops and reference field values at particular points. For that reason we list here all of the macros that are predefined for this purpose. Their use should be clear from the examples you can find in the program, and in particular in `twofldlambda.h`.

LOOP This is a loop over the entire array using the indices *i*, *j*, and *k*. For example in two dimensions this macro expands to "for($i = 0$;$i < N$;$i++$) for($j = 0$;$j < N$;$j++$)". Note that if you use *LOOP* in a function you must also declare the ints *i*, *j*, and *k*.

FIELD(fld) Gives the value of a field variable at a particular point referred to by the indices *i*, *j*, and *k*. The parameter *fld* specifies which field is being used. This macro is particularly useful in conjunction with the *LOOP* macro.

FIELDD(fld) This is identical to *FIELD* except it gives the field derivative.

INDEXLIST This is a list of arguments that can be used for a function. It expands to "int i, ..." for one dimension, "int i, int j, ..." for two and "int i, int j, int k" for three. Thus a function with *`INDEXLIST`* in its argument list can take as arguments the correct number of indices for that number of dimensions, or it can take up to three indices regardless of the number of dimensions. For example the function *`dvdf()`* uses this argument and then uses the *`FIELD`* macro to refer to the field values at that point. This scheme implicitly ignores any passed indices beyond those needed for the number of dimensions being used.

FIELDPOINT(fld,i,j,k) This macro is defined in case you need to refer to the value of a field at a specific point rather than just using the generic indices *`i`*, *`j`*, and *`k`*. In one dimension *`i`* and *`j`* are ignored and in two dimensions *`i`* is ignored.

DECLARE_INDICES This is a list of index declarations to be put at the beginning of functions that use the *`LOOP`* macro. It declares the integers i, j, and/or k needed for the correct number of dimensions. Note that if you use any of these variable names in other ways in the function you should not use this macro but just declare them yourself.

Chapter 5

Output

This section explains the output of LATTICEEASY. Having calculated the evolution of a set of fields the program has many functions for calculating and saving derived quantities about those fields. These quantities include spatial means and variances, spectra, histograms, and more. The way this works in the program is the following. Every set number of time steps (described below) the program calls the function *`save()`*, which in turn calls a series of functions designed to calculate and save certain quantities. Which of these output functions gets called is determined by a set of parameters set in the file `parameters.h`. For example the parameter *`smeansvars`* determines whether means and variances should be calculated for a particular run.

The functions for means and variances and for recording values of the scale factor and its derivatives are called each time *`save()`* is called. The other functions, for saving spectra, energy density, histograms, slices through the grid, and images of the grid, are calculated less often. The parameter *`checkpoint_interval`* sets how often (in program time units) to call these functions. This same parameter also controls how often the output file buffers are flushed. So for example if $checkpoint\_interval = 10$ and $scheckpoint = 1$ and the program is interrupted at $t = 367$ then the program should have saved a complete grid image at $t = 360$ and all output files should have been written at that time, so the program may easily be resumed from that point without any loss of data. The function *`model_output()`* is called every time *`save()`* is called, but it is passed a parameter indicating whether infrequent calculations are being performed. This way if you design model specific output functions you can decide whether to calculate them frequently or infrequently.

There is another parameter called *`noutput_flds`* that controls how many fields should be included in the output. Aside from the energy density (which must include all fields in order to check energy conservation) and two dimensional histograms (which use an explicit list of fields given in `parameters.h`) the output functions only perform calculations for the first *`noutput_flds`* fields. This is useful if, for example, you are doing a run with several fields that you expect to act nearly identically. If *`noutput_flds`* is set to zero then all fields will be output.

How often *`save()`* is called is set by the parameter *`noutput_times`*, which specifies how many times during the run the *`save()`* function should be called. We have found 1000 to be a convenient number that generally gives reasonably high resolution output but doesn't create awkwardly large output files. For runs for which high resolution is required $10,000$ outputs (or more) might be appropriate.

There is another output function that is not called by *`save()`*, but directly by the main program. This function, *`output_parameters()`*, is called only at the beginning and end of a run. It outputs a description of the model being used, the values of the run parameters, the date and time at the beginning and end of the run, and the total elapsed clock time during the run.

The names of the output files are all given extensions that depend on a couple parameters. The default extension is "_0.dat". For example the file that stores the field means will be called `means_0.dat`. If, however, you are continuing a previous run (see section 4.1.2) and you elect to create new output files rather than appending data to the old ones then the new files will be given the extension "_1.dat". If you continue that run the new extension will be "_2.dat" and so on. Note, however, that if you set the string *`alt_extension`* in `parameters.h` to be anything other than empty then it will be used as the extension for all output files no matter what.

The function *`output_initialize()`* is called at the beginning of the run to set the default extension and the number of output fields. (The number of output fields must be reset to the total number of fields if $noutput\_flds = 0$.)

The field means, variances, histograms, and slices are all set to multiply the field values being output by a rescaling value. By default this value is simply one, meaning the values being output are the program field values. This option is included, however, so that you can calculate a rescaling that you wish to use in your model. This calculations could be done once in the run or separately at each output time. For example, you could use this to reverse the program rescalings and instead output field values in physical units. If you do set the rescaling once you should do so in *`model_initialize()`*, whereas if you want to reset it repeatedly as the run progresses you should do it in the *`model_output()`* function described below.

Each of the functions that generates output is described in more detail below. Each section below gives the name of an output function, the name(s) of the files it creates, the values of its outputs, and any other necessary information. With one exception, all of the functions described here are contained in the file `output.cpp`. That one exception is the function *`model_output()`*, which is a function in the model file that allows you to implement outputs specific to a particular model. In these sections we will refer to the output files with the extension "_ext," which will typically be "_0.dat" as described above.

5.1 Run Parameters

The function *`output_parameters()`* creates a file called `info_ext` that contains information about the parameters used in the run. The information is divided into three parts.

First, there is model specific information. For this *`output_parameters()`* simply calls a function called *`modelinfo()`* in the file `model.h`. The purpose of this function is to allow you to store whatever run-specific information you want for any particular model. We recommend as a minimum using this function to output the name of your model and a brief description (e.g. the equation for the potential) as well as the values of any coupling constants or other parameters of your model.

Second, there are general run parameters. These are the number of grid points, the number of dimensions, the number of fields, the size of the box (in distance units), the size of the time step, and the type of expansion used (full, power-law, or none).

Finally, this output file contains information about when the run occurred and how long it took. At the beginning and end of the run the date and time are recorded, and at the end the difference is calculated to give the total elapsed clock time. This information is useful for comparing timing on different machines or for different parameter values, or for simply reminding yourself at a later time how long to expect a run to take.

5.2 Field Means and Variances

The function *`meansvars()`* calculates the means and variances of all output fields. The means are simply the sum of the field at each gridpoint divided by the total number of gridpoints. The variance is calculated as

$$Variance(f) = \langle f^2 \rangle - \langle f \rangle^2. \tag{5.1}$$

The means and variances are stored in files called `means_ext` and `variance_ext`. Each of these fields contains one column for the time and one column each for the output fields.

The *`meansvars()`* function is also where the program checks for runaway field behavior. If the evolution becomes unstable (usually due to using too large a time step) the fields can start growing exponentially. In this case it will generally not take many time steps before the field values exceed the maximum storage possible on the computer, and the values are then simply stored as non-numerical. The program checks for such an occurence by verifying that the average is numerical. If this test fails then the program will print a warning message, call *`output_parameters()`* to record the elapsed time for the run, flush the output buffer for means and variances, and abort. Usually the solution in such a case is to reduce the time step and rerun.

Note that if you choose not to output means and variances then this error checking won't occur. In practice, though, it is almost invariably useful to have these quantities for at least one of the output fields, so we recommend leaving this output field on for all runs.

5.3 The Scale Factor and Its Derivatives

The function *`scale()`* outputs the scale factor a, the Hubble constant $H = \frac{\dot{a}}{a}$, and the second derivative $\ddot{a}$. These quantities are all output in physical units. Using the rescalings in equation (6.2) these can be written in terms of program variables as

$$\frac{\dot{a}}{a} = Ba^{s-1}a' \tag{5.2}$$

$$\ddot{a} = B^2\left(a^{2s}a'' + sa^{2s-1}a'^2\right). \tag{5.3}$$

The file `scale_ext` contains one column for the time and three for these three quantities, in the order just given. Note that even if ***sscale*** is set to one this function is only called if self-consistent expansion is used. If there is no expansion then there is no scale factor evolution and for power-law expansion this evolution is simply given by a fixed equation.

5.4 Spectra

The function ***spectra()*** calculates several spectra for the output fields. The function begins by performing a Fourier transform on the field and derivative grids, thus generating lattices of Fourier values f_k and f'_k. At each gridpoint in Fourier space the function then calculates $|f_k|^2$, $|f'_k|^2$, n_k, and ρ_k. The formulas for the occupation number n_k and the energy spectrum ρ_k are given below.

The ***spectra()*** function doesn't output these values for each grid point, but rather groups them into bins. The size of the bins is determined by the lattice spacing dk. This means for example that in three dimensions the $0th$ bin ($0 \leq k < dk$) includes only the point $k = 0$ while the next bin ($dk \leq k < 2dk$) includes the 9 neighboring points. In one dimension there should be one point per bin. The four quantities listed above are each averaged over each bin.

The output of this function is in several files. The first, `spectratimes_ext`, simply lists the times at which spectra were recorded. There is also one file per output field that contains the spectra. These files are called `spectra<field_number>_ext`. They contain seven columns listing, for each bin, k, the number of points in the bin, ω_k^2, $|f_k|^2$, $|f'_k|^2$, n_k, and ρ_k. (The quantities ω_k, n_k, and ρ_k are defined below). Note that the four spectra (the last four columns) are calculated at each point on the lattice and then averaged for the bin, whereas k and ω_k^2 are given for the bottom of the bin.

At each time for which spectra are recorded the ***spectra()*** function will write one row for each bin into each of the spectra files. When examining the output it's easy to tell where one time ends and another begins because the frequencies (first column) will monotonically increase at each time and then reset to zero when a new spectrum begins.

The number of points in each bin is given to make it possible to take sums of different quantities over the lattice. For example Parseval's theorem for a discrete Fourier transform states

$$\sum_{positions} |f(x)|^2 = \frac{1}{N^{NDIMS}} \sum_{frequencies} |f_k|^2. \tag{5.4}$$

Other quantities such as total number density are likewise calculated as sums over all lattice points. By adding $n_k * numberofpoints$ for all bins you can find the sum of n_k over the whole lattice.

Definitions of Number and Energy Spectra

LATTICEEASY is primarily designed for calculations of processes occurring in three dimensional space. When lower numbers of dimensions are used the modes are designed to approximately reproduce the behavior of a one or two dimensional slice through a three dimensional space. Thus in our discussion of the number and energy spectra we will consider the three dimensional case. The definitions of n_k and ρ_k used for one and two dimensional runs will be given in section 5.4.

The occupation number n_k is defined to be an adiabatic invariant of the field evolution, whose integral $n \sim \int d^3k n_k$ corresponds in the large amplitude limit to classical number density. The energy density ρ_k gives the spectrum of energy in different Fourier modes. In order to give a sensible definition of these quantities in an expanding universe it is necessary first to switch to conformal coordinates (defined below) in which the field equations take the form of an undamped oscillator. When the frequency ω_k of this oscillator is changing adiabatically then a nearly constant occupation number n_k can be calculated. So in the following sections we convert the field equation to conformal coordinates, define the quantities n_k and ρ_k, and then convert *back* to physical coordinates in order to apply the program rescalings and thus derive formulas for n_k and ρ_k in program units.

Before doing any of that, however, we start by discussing the properties of Fourier transforms of classical fields. It turns out that in order to define sensible intrinsic quantities like occupation number the Fourier transforms need to be rescaled in order to take account of the finite size and spacing of the lattice. These rescalings are derived below. Some of this discussion duplicates parts of section 6.3.2 but is presented here for completeness.

The vectors $\vec{k}$ and $\vec{x}$ are shown without vector notation except where needed for clarity. An ordinary V refers to the potential while $\mathcal{V}$ is used to mean volume. Dots indicate differentiation with respect to t while primes denote differentiation with respect to rescaled time, either t_c (conformal time) or t_{pr} (program time). The usage should be clear from context. Angle brackets indicate spatial averages over the lattice box.

Rescaled Fourier Transforms

The ordinary Fourier transform of a field f is defined as

$$F_k \equiv \int d^3x f(x) e^{-ikx}. \tag{5.5}$$

The problem with using this definition for a classical scalar field is that if the Fourier components F_k are given fixed values the resultant field values $f(x)$ become dependent on the overall size of the box within which the theory is defined. For example

$$\int d^3x f(x)^2 = \frac{1}{(2\pi)^3} \int d^3k |F_k|^2 \tag{5.6}$$

which implies

$$\left\langle f(x)^2 \right\rangle = \frac{1}{(2\pi)^3 \mathcal{V}} \int d^3k |F_k|^2. \tag{5.7}$$

The size of the box does not affect the integral, except by turning it into a discrete sum. So to keep $f(x)$ (and by extension all intensive quantities) independent of the box size we define a modified Fourier transform

$$\tilde{F}_k \equiv \frac{1}{\sqrt{\mathcal{V}}} F_k = \frac{1}{L^{3/2}} F_k \tag{5.8}$$

where L is the size of the box. This modified transform takes on the same value regardless of the box size, while the actual Fourier transform must be rescaled. Note that the units of F_k are $[M]^{-2}$ while those of $\tilde{F}_k$ are $[M]^{-1/2}$.

The Fourier transform used by the program is neither of these, however, but rather the discrete Fourier transform f_k, related to the usual, continuous, one by

$$f_k = \frac{1}{dx^3} F_k = \frac{L^{3/2}}{dx^3} \tilde{F}_k. \tag{5.9}$$

All physical quantities should be defined in terms of $\tilde{F}_k$ and the Fourier transform f_k used by the program should be adjusted accordingly. For example, the initial vacuum state $\tilde{F}_k^2 = \frac{1}{2\omega_k}$ becomes $f_k^2 = \frac{L^3}{2dx^6\omega_k}$. See section 6.3.2 for more details.

Conformal Coordinates

The equation of motion for a scalar field in an expanding universe is

$$\ddot{f} + 3\frac{\dot{a}}{a}\dot{f} - \frac{1}{a^2}\nabla^2 f + \frac{\partial V}{\partial f} = 0. \tag{5.10}$$

To define an adiabatic invariant occupation number for this field we need to switch to conformal variables in which this equation becomes a more standard oscillator equation. These variables are

$$f_c \equiv af;\ dt_c \equiv \frac{1}{a}dt. \tag{5.11}$$

Using conformal time and noting that

$$\dot{f} = \frac{1}{a}f';\ \ddot{f} = \frac{1}{a^2}f'' - \frac{a'}{a^3}f' \tag{5.12}$$

the equation of motion becomes

$$f'' + 2\frac{a'}{a}f' - \nabla^2 f + a^2\frac{\partial V}{\partial f} = 0. \tag{5.13}$$

Then switching to conformal field values note that

$$f' = \frac{1}{a}f_c' - \frac{a'}{a^2}f_c;\ f'' = \frac{1}{a}f_c'' - 2\frac{a'}{a^2}f_c' + 2\frac{a'^2}{a^3}f_c - \frac{a''}{a^2}f_c \tag{5.14}$$

and thus

$$f_c'' - \frac{a''}{a}f_c - \nabla^2 f_c + a^4\frac{\partial V}{\partial f_c} = 0. \tag{5.15}$$

This equation can be approximated in Fourier space by

$$\tilde{F}_{k,c}'' + \omega_k \tilde{F}_{k,c} = 0 \tag{5.16}$$

where $\tilde{F}_{k,c} = a\tilde{F}_k$ and

$$\omega_k^2 \equiv k^2 + a^4\left\langle\frac{\partial^2 V}{\partial f_c^2}\right\rangle - \frac{a''}{a} = k^2 + a^2\left\langle\frac{\partial^2 V}{\partial f^2}\right\rangle - \frac{a''}{a}. \tag{5.17}$$

Occupation Number and Energy Spectrum

In terms of the conformal variables of the previous section it makes sense to define occupation number as

$$n_k \equiv \frac{1}{2}\left(\omega_k|\tilde{F}_{k,c}|^2 + \frac{1}{\omega_k}|\tilde{F}_{k,c}'|^2\right). \tag{5.18}$$

Note that this quantity is adiabatically invariant, meaning it is conserved in the limit $\frac{\omega_k'}{\omega_k^2} \ll 1$. Note also that because it is defined in terms of $\tilde{F}$ instead of F_k, n_k is unitless.

The energy density ρ_k is defined as

$$\rho_k \equiv \omega_k n_k = \frac{1}{2}\left(\omega_k^2|\tilde{F}_{k,c}|^2 + |\tilde{F}_{k,c}'|^2\right). \tag{5.19}$$

To convert these definitions back to physical coordinates note that

$$a' = a\dot{a};\ a'' = a^2\ddot{a} + a\dot{a}^2;\ \tilde{F}_{k,c}' = a\tilde{F}_k' + a'\tilde{F}_k = a^2\dot{\tilde{F}}_k + a\dot{a}\tilde{F}_k \tag{5.20}$$

so

$$n_k = \frac{1}{2}\left(a^2\omega_k|\tilde{F}_k|^2 + \frac{a^4}{\omega_k}|\dot{\tilde{F}}_k + \frac{\dot{a}}{a}\tilde{F}_k|^2\right) \tag{5.21}$$

$$\rho_k = \frac{1}{2}\left(a^2\omega_k^2|\tilde{F}_k|^2 + a^4|\dot{\tilde{F}}_k + \frac{\dot{a}}{a}\tilde{F}_k|^2\right) \tag{5.22}$$

$$\omega_k^2 = k^2 + a^2\left\langle\frac{\partial^2 V}{\partial f^2}\right\rangle - a\ddot{a} - \dot{a}^2. \tag{5.23}$$

Finally, in terms of the discrete Fourier transform f_k

$$n_k = \frac{a^2 dx^6}{2L^3}\left[\omega_k|f_k|^2 + \frac{a^2}{\omega_k}|\dot{f}_k + \frac{\dot{a}}{a}f_k|^2\right]. \tag{5.24}$$

$$\rho_k = \frac{a^2 dx^6}{2L^3}\left[\omega_k^2|f_k|^2 + a^2|\dot{f}_k + \frac{\dot{a}}{a}f_k|^2\right]. \tag{5.25}$$

Program Variables

In general the program uses neither physical nor conformal coordinates but a run-specific set of rescalings defined as

$$f_{pr} \equiv A a^r f;\ dt_{pr} \equiv B a^s dt;\ x_{pr} \equiv Bx. \tag{5.26}$$

(See section 6.1.) Noting that k scales like $1/x$ and

$$\dot{a} = B a^s a';\ \ddot{a} = B^2 \left(a^{2s} a'' + s a^{2s-1} a'^2\right) \tag{5.27}$$

$$f_k = \frac{a^{-r}}{A} f_{k,pr};\ \dot{f}_k = B a^s f_k' = \frac{B}{A}\left(a^{s-r} f_{k,pr}' - r a^{s-r-1} a' f_{k,pr}\right), \tag{5.28}$$

the equations for n_k, ρ_k, and ω_k become

$$n_k = \frac{a^{2-2r}}{A^2 B^3} \frac{dx_{pr}^6}{2L_{pr}^3}\left[\omega_k |f_{k,pr}|^2 + \frac{a^{2+2s}B^2}{\omega_k} |f_{k,pr}' + (1-r)\frac{a'}{a} f_{k,pr}|^2\right] \tag{5.29}$$

$$\rho_k = \frac{a^{2-2r}}{A^2 B^3} \frac{dx_{pr}^6}{2L_{pr}^3}\left[\omega_k^2 |f_{k,pr}|^2 + a^{2+2s}B^2 |f_{k,pr}' + (1-r)\frac{a'}{a} f_{k,pr}|^2\right] \tag{5.30}$$

$$\omega_k^2 = B^2\left(k_{pr}^2 + a^{2+2r}\frac{A^2}{B^2}\left\langle \frac{\partial^2 V}{\partial f_{pr}^2}\right\rangle - a^{2s+1}a'' - (1+s)a^{2s}a'^2\right). \tag{5.31}$$

Using the additional definitions (sections 6.1.1 and 6.3.2)

$$V_{pr} \equiv \frac{A^2}{B^2} a^{-2s+2r} V;\ m_{pr}^2 \equiv a^{2s+2}\left\langle \frac{\partial^2 V_{pr}}{\partial f_{pr}^2}\right\rangle;\ \omega_{k,pr} \equiv \frac{\omega_k}{B}, \tag{5.32}$$

we bring the equations to their final forms

$$n_k = \frac{a^{2-2r}}{A^2 B^2} \frac{dx_{pr}^6}{2L_{pr}^3}\left[\omega_{k,pr} |f_{k,pr}|^2 + \frac{a^{2+2s}}{\omega_{k,pr}} |f_{k,pr}' + (1-r)\frac{a'}{a} f_{k,pr}|^2\right] \tag{5.33}$$

$$\rho_k = \frac{a^{2-2r}}{A^2 B} \frac{dx_{pr}^6}{2L_{pr}^3}\left[\omega_{k,pr}^2 |f_{k,pr}|^2 + a^{2+2s} |f_{k,pr}' + (1-r)\frac{a'}{a} f_{k,pr}|^2\right] \tag{5.34}$$

$$\omega_{k,pr}^2 = k_{pr}^2 + m_{pr}^2 - a^{2s+1}a'' - (1+s)a^{2s}a'^2. \tag{5.35}$$

Number and Energy Spectra in One and Two Dimensions

The ordinary power spectra $|f_k|^2$ and $|f_k'|^2$ are output in the same way regardless of the number of dimensions, but the spectra n_k and ρ_k have well defined normalizations, so in one and two dimensions they are normalized so as to try to approximate the values that they would have in three dimensions. Let f_{3k} be the Fourier transform of a three dimensional field $f(x)$, and let f_{1k} and f_{2k} be the Fourier transforms of the field taken on one and two dimensional slices respectively. Assuming that isotropy holds on average we show in section 6.3.5 that

$$|f_{1k}|^2 \approx \frac{dx^4}{2\pi L^2} k^2 |f_{3k}|^2 \tag{5.36}$$

and

$$|f_{2k}|^2 \approx \frac{dx^2}{\pi L} |k| |f_{3k}|^2. \tag{5.37}$$

Recall that the wave vector $\vec{k}$ is given by

$$\vec{k} = \frac{2\pi}{L}\vec{i} \tag{5.38}$$

where $\vec{i}$ is the position on the grid in Fourier space (i.e. a triplet of integers from $-N/2$ to $N/2$). Thus

$$|f_{1k}|^2 \approx \frac{2\pi dx^4}{L^4} i^2 |f_{3k}|^2 = \frac{2\pi}{N^4} i^2 |f_{3k}|^2, \tag{5.39}$$

$$|f_{2k}|^2 \approx \frac{2dx^2}{L^2}|i||f_{3k}|^2 = \frac{2}{N^2}|i||f_{3k}|^2. \tag{5.40}$$

The Fourier transform f_k that is actually calculated in one and two dimensional simulations will correspond to f_{1k} and f_{2k}. The definitions of n_k and ρ_k, however, are given in terms of f_{3k}. Thus equations [5.33] and [5.34] are multiplied by prefactors equal to $\frac{N^4}{2\pi i^2}$ and $\frac{N^2}{2|i|}$ in one and two dimensions respectively.

5.5 Energy Density

The function ***energy()*** calculates the components of the energy density. The energy density is calculated for all fields regardless of the value of ***noutput_flds***. The output is in two files, one for the components of energy density and one for monitoring energy conservation.

The components are output to a file called `energy_ext`. The number of columns in this file varies with the number of fields and with the number of terms in the potential. Specifically, the first column contains the time, the next ***nflds*** columns contain the kinetic energy $\frac{1}{2}\left(\frac{\partial f_{pr}}{\partial t_{pr}}\right)^2$, the next ***nflds*** columns contain the gradient energy $\frac{1}{2a^2}|\nabla f_{pr}|^2$, and the remaining columns contain the potential terms. The kinetic and gradient energies are calculated by ***energy()***, whereas the potential energy is calculated by a model-specific function called ***potential_energy()*** in `model.h`. The output is in program units, meaning

$$\rho_{pr} \equiv \frac{A^2}{B^2} a^{-2s+2r} \rho \tag{5.41}$$

where

$$\rho = \left\langle \frac{1}{2}\dot{f}^2 + \frac{1}{2a^2}|\nabla f|^2 + V \right\rangle \tag{5.42}$$

is the physical energy density and A, B, r, and s are the rescaling parameters defined in equation (6.2). Angle brackets denote averages over the grid. Plugging in the rescalings gives

$$\rho_{pr} = \left\langle \frac{1}{2}f_{pr}'^2 - r\frac{a'}{a}f_{pr}f_{pr}' + \frac{1}{2}r^2\left(\frac{a'}{a}\right)^2 f_{pr}^2 + \frac{1}{2}a^{-2(s+1)}|\nabla_{pr} f_{pr}|^2 + V_{pr} \right\rangle. \tag{5.43}$$

The second file generated by ***energy()*** is called `conservation_ext`. The first column contains the time and the second column contains a quantity used to monitor energy conservation. For the case of no expansion (Minkowski space) this quantity is simply the ratio of total energy density to energy density at the beginning of the run. Of course this quantity should remain close to 1 throughout the run. In an expanding universe the situation is a little more complicated because energy density is not conserved, but rather decreases in a way determined by the expansion rate and the equation of state of the fields. This redshifting of energy is described by the continuity equation

$$\dot{\rho} + 3\frac{\dot{a}}{a}(\rho + 3p) = 0. \tag{5.44}$$

Equation (5.44) can be derived from the Friedmann equations, so if these are being solved exactly then the continuity equation will be obeyed as well. So a simple way to check energy conservation in an expanding universe is to use one of the Friedmann equations, equations (6.19) and (6.20) to evolve the scale factor and then check if the other is being satisfied as well. In practice the program uses a combination of these two equations so either one can be used as a check of energy conservation. We use equation (6.20), i.e.

$$\left(\frac{\dot{a}}{a}\right)^2 = \frac{8\pi}{3}\rho. \tag{5.45}$$

So for an expanding universe the second column of the conservation file contains the ratio of $\left(\frac{\dot{a}}{a}\right)^2$ to $\frac{8\pi}{3}\rho$. To express this in terms of program variables we use equations (5.2) and (5.41) to give

$$H^2/\left(\frac{8\pi}{3}\rho\right) = \frac{3}{8\pi}A^2 a^{2r-2} a'^2 \frac{1}{\rho_{pr}}. \tag{5.46}$$

As in the case of no expansion this quantity should remain close to 1 throughout the run. We have generally found that the deviation of this ratio from one is comparable to the lack of energy conservation for runs done with the same models without expansion. We have also checked that using the second Friedmann equation, equation (6.19), gives essentially the same results as this method.

In the case of fixed power-law expansion the conservation file is not created.

5.6 Histograms

The function *`histograms()`* creates histograms of the output fields. The number of bins used is set in `parameters.h`. These bins are equally spaced between the minimum and maximum field values for the histogram. These values can be set to a fixed value using the variables *`histogram_min`* and *`histogram_max`* in `parameters.h`. This option can be useful, for example, for generating output that is easy to compare from one time to the next. If these two values are set equal to each other, however, then *`histograms()`* calculates the appropriate range for each field by finding the minimum and maximum value of the field on the grid at each time.

The output is simply the fraction of gridpoints for each bin for which the field is in the appropriate range.

The output is in the following files. The file `histogramtimes_ext` contains a column with the times at which histograms were recorded plus two additional columns for each field. These columns record the minimum field value for that field and the spacing (in units of field values) between successive bins. These may thus be used to label the field values in each bin of a given histogram. These are the quantities that are rescaled if the model file sets a rescaling for all the output. See section 5.10. The histograms themselves, i.e. the counts for each bin, are in the files `histogram<field_number>_ext`. These files contain the bin frequencies in a single column. Histograms at different times are separated by blank lines. If you are reading the data into a program for plotting and you don't remember how many bins were used in each histogram you can recover this either by looking at how many lines occur between each blank line or by dividing the total number of lines in the `histogram` files by the number of times in the `histogramtimes` file.

5.7 Two Field Histograms

The function *`histograms_2d()`* records two dimensional histograms of pairs of fields. The file `parameters.h` contains an array that lists the pairs for which these histograms are to be calculated, as well as two parameters giving the number of bins to use in each direction. The basic structure of this function is similar to *`histograms()`*. The minimum and maximum value used in each histogram are given by the parameters *`histogram2d_min`* and *`histogram2d_max`* or, if these are equal, set dynamically to the minimum and maximum value of each field at that time. Next the bin values are fixed in a two dimensional array between those limiting values. Then the number of grid points in each square of that array is counted. The file `histogram2dtimes_ext` is similar to `histogramtimes` from the *`histogram()`* function except that for each histogram the file now lists the minimum value of the first field, the bin spacing of the first field, and then the minimum value and bin spacing of the second field. The files `histogram2d<field1_number>_<field2_number>_ext` lists the bin frequencies in a single column. They should be read in the usual C array order. In other words if the histogram has 10 bins in the second field direction then the first 10 frequencies represent increasing values of the second field with the first field at its minimum, the next 10 represent increasing values of the second field with the first field at its next highest value, and so forth.

5.8 Slices

The function *`slices()`* creates an ASCII file containing the field values at each point in a slice through the lattice. The number *`slicedim`* in `parameters.h` sets the number of dimensions for the slice. There are three other parameters controlling the output of this function, *`slicelength`*, *`sliceskip`*, and *`sliceaverage`*. The first determines how many points (in each dimension) will be included while *`sliceskip`* determines how many points will be skipped in between adjacent points on the slice. For example, if $slicedim = 2$, $slicelength = 2$, and $sliceskip = 3$ then the function will output the lattice points $(0, 0)$, $(0, 3)$, $(3, 0)$, and $(3, 3)$. (If the lattice is three dimensional then the first index will be fixed at zero.) If *`sliceaverage`* is set to one, however, then all field values in between the array points being sampled will be averaged. In the example above, for instance, the first output value would be the average of the field values at the nine points $(0-2, 0-2)$.

The file `slicetimes` simply records the times at which slices were output and the files `slice<field>_ext` record the field values for each output field. The values are recorded in usual C array order, as in the 2x2 example just given.

5.9 Checkpointing - Grid Images

The function *`checkpoint()`* outputs an exact image of the grid, as well as several other variables needed to restart the program from a particular time. Specifically the grid image contains (in order) the run number (0 for a first run, 1 for a continuation of that, etc.), the time t, the scale factor a and its derivative a', the array of fields, and the array of field derivatives. These quantities are stored in binary so as to keep the file size minimal. The program is designed to be able to read in one of these image files to use as initial conditions for a run continuation. These values are stored in leapfrog fashion with the field and scale factor derivatives representing values one half time step behind the time, field, and scale factor variables. See section 4.1.2 for details on resuming runs.

Ordinarily the file with the grid image is called `grid.img`. It is saved every time the program checkpoints. We recommend setting the program to checkpoint fairly often so that if a run is interrupted not much time is lost. Every time the grid image is stored it overwrites the previous image, so at the end of the run the only image will be of the last time calculated. Sometimes, however, you may want to store intermediate images. For example you might want to store the value of the field just before a critical event like a phase transition so that later if you need to you can rerun and calculate histograms more frequently in that period. Whatever the reason, there is a simple way to do this. In the file `parameters.h` there is an array called *`store_lattice_times`*. If the first entry in this array is anything other than zero then its entries are taken as times to store intermediate grid images. In this case the program will write to the first file until the first storage time has passed and then close that file and open a new one. If the last storage time is smaller than the final time of the run then one last file will be created after that last time and will be used until the end of the run.

The files will be named `grid<number>.img` where number is the integer part of the storage time. For example, if you do a run from $t = 0$ to $t = 200$ and the array *`store_lattice_times`* contains 50.0, 125.0, and 175.0 then when the run ends you will have the following grid image files: `grid50.img`, `grid125.img`, `grid175.img`, and `grid200.img`. If you did this same run with no storage times indicated the sole image file would simply be named `grid.img`. Note that if a run is interrupted then the times listed in the files may be incorrect. In the example above, if the computer crashed at $t = 103$ then the last image file would be called `grid125.img` even though it might at that point contain an image recorded at $t = 100$. Also, if you set the program to only checkpoint every $t = 10$ then the files `grid125.img` and `grid175.img` would contain images at $t = 120$ and $t = 170$ respectively.

When continuing a previous run LATTICEEASY looks for a grid image named `grid.img`, so if you want to continue from one of these stored files you may have to rename it.

5.10 Model Specific Output

The function *`model_output()`* allows you to generate outputs specific to your model. This could be used, for example, to look for topological defects, to output different combinations of the fields, or for whatever else you wanted. The function takes two inputs, the integer *`flush`* and the string *`ext_`*. The former tells the function whether or not infrequent calculations such as checkpointing and spectra are being done (1 and 0 for yes and no respectively). This way you can choose for your output tasks to be done each time a save is performed or only when infrequent calculations are done. In the latter case you can simply have *`model_output()`* return without doing anything except when $flush = 1$. The second variable is the extension being used for output files, so that if you wish you can create other output files with the same extension.

This function can also be used to set the global variable *`rescaling`*, which will be multiplied by all field values output by *`meansvars()`*, *`histograms()`*, *`histograms2d()`*, and *`slices()`*. If you just want to set this value once for the entire run you should probably do that in *`model_initialize()`*, but if you do it in *`model_output()`* then you can rescale dynamically as the run continues.

Chapter 6

Equations

This section derives and explains the equations solved in LATTICEEASY. Although occasional reference will be made to program files or variables, the focus here is on the equations themselves; their implementation in the program is discussed in the other sections of the documentation.

All of the equations are presented in two forms. First they are derived in their usual physical forms. The metric, units, and other conventions used are explained in section 2. In the program itself, however, the field and spacetime variables are rescaled in ways that make the equations simpler to solve. The form of these rescalings is discussed in section 6.1, and from there on all equations are given both in terms of physical and program variables.

Section 6.1 gives the field equations and uses them to motivate the variable rescalings used in the program. Section 6.2 derives the evolution equation for the scale factor, which the program solves for self-consistently using the Einstein equations. There is also an option in the program to use a fixed power-law expansion and the relevant equations are derived and discussed in section 6.2 as well. Section 6.3 discusses the setting of initial conditions, including initial values for the fields, field derivatives, and Hubble constant. Finally, section 6.4 discusses the "staggered leapfrog" method used by the program for solving differential equations.

6.1 Field Equations and Coordinate Rescalings

6.1.1 Field Equations

The equation of motion for a scalar field in an expanding universe is

$$\ddot{f} + 3\frac{\dot{a}}{a}\dot{f} - \frac{1}{a^2}\nabla^2 f + \frac{\partial V}{\partial f} = 0. \tag{6.1}$$

To begin with we will assume a general form for the variable rescalings

$$f_{pr} \equiv A a^r f;\ \vec{x}_{pr} \equiv B\vec{x};\ dt_{pr} \equiv B a^s dt. \tag{6.2}$$

The position coordinates shouldn't be modified by any powers of the scale factor so that the same comoving wavelengths stay in the box throughout the run. In principle the coefficient B could be different for $\vec{x}$ and t but it is simpler and more useful to keep them the same, thus giving an initial sound speed of one. Below we will derive expressions for these rescaling coordinates in terms of parameters in the potential of the model being solved. To express the field equation in program variables note that

$$\dot{a} = B a^s a';\ \dot{f} = B a^s f' = \frac{B}{A}\left(a^{s-r} f'_{pr} - r a^{s-r-1} a' f_{pr}\right) \tag{6.3}$$

$$\ddot{f} = \frac{B^2}{A}\left(a^{2s-r} f''_{pr} + (s-2r) a^{2s-r-1} a' f'_{pr} - r(s-r-1) a^{2s-r-2} a'^2 f_{pr} - r a^{2s-r-1} a'' f_{pr}\right). \tag{6.4}$$

Finally define a new potential

$$V_{pr} \equiv \frac{A^2}{B^2} a^{-2s+2r} V. \tag{6.5}$$

(The potential rescaling is chosen to be consistent with an energy rescaling that gives $\rho_{pr} \supset 1/2 f_{pr}'^2$.) Then putting in all rescalings the equation of motion becomes

$$f_{pr}'' + (s - 2r + 3)\frac{a'}{a} f_{pr}' - a^{-2s-2} \nabla_{pr}^2 f_{pr} - \left(r(s - r + 2) \left(\frac{a'}{a}\right)^2 + r\frac{a''}{a} \right) f_{pr} + \frac{\partial V_{pr}}{\partial f_{pr}} = 0. \tag{6.6}$$

6.1.2 Selecting the Rescaling Variables

The program allows the user to set the rescaling variables A, B, r, and s to whatever is most useful for a particular model (up to one caveat discussed below). However there are certain guidelines that we suggest following in setting these variables. These guidelines are written into the program as defaults in the file `model.h`. (See the section on how to implement new models for more details on how the program sets these variables.) These default settings are based on the following criteria:

1. We want to eliminate the first derivative term from the equation of motion. This will make the equations simpler to solve.
2. We want to set the scale of the field variables to be of order unity, at least initially. This will make numerical calculations simpler as well as making the output more readable.
3. We want the coefficient of the dominant potential term to be of order unity. This means the program time variable will automatically measure the natural time scale of the problem.
4. We want the coefficient of the dominant potential term to include no powers of the scale factor. If this weren't true the time scale of the calculations would be changing as the program progressed and a fixed time step would be untenable.

The first two of these conditions immediately give us two equations

$$s - 2r + 3 = 0 \tag{6.7}$$

and

$$A = \frac{1}{\phi_0} \tag{6.8}$$

where ϕ_0 is the initial value of the inflaton (or whatever field dominates initially). The latter two conditions require that we assume some form for the dominant potential term. We assume that term to be polynomial; if you wish to consider a model where the dominant term is not polynomial you should expand it in a Taylor series and will hopefully be able to identify an effective dominant polynomial term. So take this dominant term to be of the form

$$V = \frac{cpl}{\beta} \phi^\beta \tag{6.9}$$

Then the corresponding term that appears in the equation of motion for ϕ will be

$$\frac{dV_{pr}}{d\phi_{pr}} = \frac{d}{d\phi_{pr}} \left(\frac{cpl}{\beta} A^{2-\beta} B^{-2} a^{-2s+r(2-\beta)} \phi_{pr}^\beta \right) = cpl A^{2-\beta} B^{-2} a^{-2s+r(2-\beta)} \phi^{\beta-1} \tag{6.10}$$

so the last two criteria above give

$$cpl A^{2-\beta} B^{-2} = 1 \tag{6.11}$$

$$-2s + r(2 - \beta) = 0. \tag{6.12}$$

Putting all these equations together

$$A = \frac{1}{\phi_0};\ B = \sqrt{cpl}\phi_0^{-1+\beta/2};\ r = \frac{6}{2+\beta};\ s = 3\frac{2-\beta}{2+\beta}. \tag{6.13}$$

If you choose to use these defaults you simply have to set β, cpl, and ϕ_0 for your model and the other variables will be set automatically according to equation (6.13). Otherwise you can directly change the definitions of A, B, r, and s. All of these definitions are in the file `model.h`. If you do explicitly change the four rescaling parameters you should set the initial field values and derivatives as needed for your rescaling scheme. (These values are all set in the file `parameters.h`.) However, the relationship $s - 2r + 3 = 0$ must be maintained because the evolution equations don't include a first derivative term and will thus be invalid if this relationship is violated. The variable definitions in `model.h` explicitly set $s = 2r - 3$ so you can independently set A, B, and r to whatever you wish.

Using these default values the field equation becomes

$$f''_{pr} - a^{-4(4-\beta)/(2+\beta)}\nabla^2_{pr} f_{pr} - \left(6\frac{4-\beta}{(2+\beta)^2}\left(\frac{a'}{a}\right)^2 + \frac{6}{2+\beta}\frac{a''}{a}\right) f_{pr} + \frac{dV_{pr}}{df_{pr}} = 0 \tag{6.14}$$

where

$$V_{pr} = \frac{1}{cpl\phi_0^\beta} a^{6\beta/(2+\beta)} V. \tag{6.15}$$

Note that this expression for the potential will pick up additional rescaling factors when V is expressed in terms of the program fields f_{pr}. For example a quartic coupling of the form $1/2g^2\phi^2\chi^2$ will give rise to

$$V_{pr} = \frac{g^2}{2}\frac{\phi_o^{4-\beta}}{cpl} a^{6(-4+\beta)/(2+\beta)} \phi_{pr}^2 \chi_{pr}^2 \tag{6.16}$$

$$\frac{dV_{pr}}{d\phi_{pr}} = \frac{g^2\phi_o^{4-\beta}}{cpl} a^{6(-4+\beta)/(2+\beta)} \chi_{pr}^2 \phi_{pr} \tag{6.17}$$

$$\frac{dV_{pr}}{d\chi_{pr}} = \frac{g^2\phi_o^{4-\beta}}{cpl} a^{6(-4+\beta)/(2+\beta)} \phi_{pr}^2 \chi_{pr}. \tag{6.18}$$

6.2 Scale Factor Evolution

The Einstein equations for an FRW universe determine the evolution of the scale factor a. In section 6.2.1 we derive the particular combination of these equations used by the program and then convert this equation to program variables. There is a difficulty that arises in solving this equation using the staggered leapfrog algorithm employed by LATTICEEASY, though, because the equation contains first derivative terms. This problem is explained in detail in section 6.4. Section 6.2.2 presents a revised formula that solves the problem. Finally, section 6.2.3 derives the equations used for fixed power-law expansion.

6.2.1 The Scale Factor Equation

The equation for the scale factor a is derived from the Friedmann equations

$$\ddot{a} = -\frac{4\pi a}{3}(\rho + 3p) \tag{6.19}$$

$$\left(\frac{\dot{a}}{a}\right)^2 = \frac{8\pi}{3}\rho \tag{6.20}$$

For a set of scalar fields f_i in an FRW universe

$$\rho = T + G + V;\ p = T - \frac{1}{3}G - V \tag{6.21}$$

where T, G, and V are kinetic (time derivative), gradient, and potential energy respectively with

$$T = \frac{1}{2}\dot{f}_i^2;\ G = \frac{1}{2a^2}|\nabla f_i|^2. \tag{6.22}$$

Equations (6.19) and (6.20) and the field evolution equations form an overdetermined system. In principle either scale factor equation could be used but in practice it is easiest to combine them so as to eliminate the time derivative term T because in the staggered leapfrog algorithm f and $\dot{f}$ are known at different times. Eliminating T we get

$$T = \frac{3}{8\pi}\left(\frac{\dot{a}}{a}\right)^2 - G - V \tag{6.23}$$

$$\ddot{a} = -\frac{4\pi a}{3}(4T - 2V) = -2\frac{\dot{a}^2}{a} + 8\pi a\left(\frac{2}{3}G + V\right) = -2\frac{\dot{a}^2}{a} + \frac{8\pi}{a}\left(\frac{1}{3}|\nabla f_i|^2 + a^2 V\right). \tag{6.24}$$

To convert to program variables note that

$$\dot{a} = B a^s a';\ \ddot{a} = B^2\left(a^{2s}a'' + s a^{2s-1}a'^2\right), \tag{6.25}$$

so the scale factor equation becomes

$$a'' = (-s-2)\frac{a'^2}{a} + \frac{8\pi}{A^2}a^{-2s-2r-1}\left(\frac{1}{3}|\nabla_{pr} f_{i,pr}|^2 + a^{2s+2}V_{pr}\right). \tag{6.26}$$

6.2.2 Correcting for Staggered Leapfrog

In practice the program uses a staggered leapfrog algorithm so in solving for $a''(t)$ the value of a' is known at $t - d/2$ where d is the time step. See section 6.4 for more details. The solution to this problem is to use the two equations

$$a'_+ \approx a'_- + d a'';\ a' \approx \frac{1}{2}\left(a'_+ + a'_-\right) \tag{6.27}$$

where a'_+ and a'_- refer to the values of a' at $t + d/2$ and $t - d/2$ respectively and all other variables are evaluated at time t. Take the evolution equation to be

$$a'' = -C_1\frac{a'^2}{a} + C_2. \tag{6.28}$$

Plugging this form into equation (6.27) and eliminating a' gives

$$a'_+ \approx a'_- + d\left(-\frac{C_1}{4a}\left(a'_+ + a'_-\right)^2 + C_2\right) = -\frac{dC_1}{4a}a'^2_+ - \frac{dC_1}{2a}a'_+a'_- - \frac{dC_1}{4a}a'^2_- + dC_2 + a'_- \tag{6.29}$$

$$\begin{aligned} a'_+ &\approx -\frac{2a}{dC_1}\left(\frac{dC_1}{2a}a'_- + 1 \pm \sqrt{\frac{d^2C_1^2}{4a^2}a'^2_- + \frac{dC_1}{a}a'_- + 1 - \frac{d^2C_1^2}{4a^2}a'^2_- + \frac{d^2C_1C_2}{a} + \frac{dC_1}{a}a'_-}\right) && (6.30)\\ &= -a'_- - \frac{2a}{dC_1} \pm \frac{2a}{dC_1}\sqrt{1 + \frac{2dC_1}{a}a'_- + \frac{d^2C_1C_2}{a}}. && (6.31)\end{aligned}$$

To determine whether to use the plus or minus sign in equation (6.31) consider the limit as $d \to 0$. In this limit

$$a'_+ \approx -a'_- - \frac{2a}{dC_1} \pm \frac{2a}{dC_1}\sqrt{1 + \frac{2dC_1}{a}a'_-} \approx -a'_- - \frac{2a}{dC_1} \pm \left(\frac{2a}{dC_1} + 2a'_-\right). \tag{6.32}$$

This suggests that the plus sign must be used in order to reduce to the limit $a'_+ \approx a'_-$. Hence

$$a'_+ \approx -a'_- - \frac{2a}{dC_1}\left(1 - \sqrt{1 + \frac{2dC_1}{a}a'_- + \frac{d^2C_1C_2}{a}}\right). \tag{6.33}$$

In the program it's useful to calculate a'', which is roughly $(a'_+ - a'_-)/d$, so

$$a'' \approx \frac{1}{d}\left[-2a'_- - \frac{2a}{dC_1}\left(1 - \sqrt{1 + \frac{2dC_1a'_-}{a} + \frac{d^2C_1C_2}{a}}\right)\right]. \tag{6.34}$$

Thus equation (6.26) becomes

$$a'' \approx \frac{1}{d}\left\{-2a'_- - \frac{2a}{dC_1}\left[1 - \sqrt{1 + \frac{2dC_1a'_-}{a} + d^2C_1\frac{8\pi}{A^2}a^{-C_3}\left(\frac{1}{3}|\nabla_{pr} f_{i,pr}|^2 + a^{C_4}V_{pr}\right)}\right]\right\} \tag{6.35}$$

where

$$C_1 = s + 2;\ C_3 = 2s + 2r + 2;\ C_4 = 2s + 2 \tag{6.36}$$

6.2.3 Power-Law Expansion

LATTICEEASY is designed to self-consistently solve for the evolution of scalar fields f and the scale factor a in an expanding universe. In some cases, however, you may wish to solve for the behavior of a set of fields in a universe dominated by other forms of energy, e.g. pure matter or radiation. In this case you can tell the program to impose a fixed power-law expansion and evolve the fields in this background. In this section we derive the equations for such an expansion in program variables. Note that we use the variables C_i to denote constants of the equations. The C_i in this section have no relation to the ones in the previous (or any other) section.

For a general constant equation of state the scale factor evolution is given by

$$a = C_1(t - t_0)^\gamma. \tag{6.37}$$

The program time is rescaled as

$$t_{pr} \equiv B \int a^s dt = C_1^s B \int (t - t_0)^{\gamma s} dt = \frac{C_1^s B}{\gamma s + 1}(t - t_0)^{\gamma s+1} + C_2, \tag{6.38}$$

which can be inverted to give

$$t - t_0 = \left(\frac{\gamma s + 1}{C_1^s B}(t_{pr} - C_2)\right)^{\frac{1}{\gamma s+1}} \tag{6.39}$$

and thus

$$a = (C_3 t_{pr} - C_4)^{\frac{\gamma}{\gamma s+1}}. \tag{6.40}$$

To solve for the parameters C_i we want to match the values of a and H at the beginning of the simulation, $t_{pr} = 0$. The scale factor itself has an arbitrary scaling and is set to 1 initially, while the Hubble constant has some well defined initial value H_0. The first constraint trivially gives $C_4 = -1$. The second constraint is most easily defined in terms of the program value of the Hubble constant,

$$H_{pr} \equiv \frac{a'}{a} = \frac{\gamma}{\gamma s + 1}\frac{C_3}{C_3 t_{pr} + 1}. \tag{6.41}$$

Let $H_{pr,0}$ be the value of H_{pr} when $t_{pr} = 0$

$$C_3 = \frac{\gamma s + 1}{\gamma} H_{pr,0}. \tag{6.42}$$

So

$$a = \left(\frac{1}{G} H_{pr,0} t_{pr} + 1\right)^G \tag{6.43}$$

where

$$G \equiv \frac{\gamma}{\gamma s + 1}. \tag{6.44}$$

The program value $H_{pr,0}$ is derived in section 6.3.6 and is automatically calculated by the program. The rescaling variable s should be defined for your model, so all you need for a power-law expansion is to specify the value of γ, which is declared in `parameters.h` with the variable name *`expansion_power`*. Note that if you know the equation of state $p = \alpha\rho$ that you want the corresponding power-law expansion will be given by

$$\gamma = \frac{2}{3(1 + \alpha)}. \tag{6.45}$$

(See for example [4].) If we let

$$f(t_{pr}) \equiv \frac{1}{G} H_{pr,0} t_{pr} + 1. \tag{6.46}$$

then the final form of the power-law expansion equations is

$$a = f^G \tag{6.47}$$

$$a' = H_{pr,0} f^{G-1} = \frac{H_{pr,0}}{f} a \tag{6.48}$$

$$a'' = \frac{G - 1}{G} H_{pr,0}^2 f^{G-2} = \frac{(G - 1) H_{pr,0}^2}{G f^2} a. \tag{6.49}$$

The parameters f and G are called *`sfbase`* and *`sfexponent`* respectively in the program.

6.3 Initial Conditions on the Lattice

There are three kinds of initial conditions that need to be set for the lattice calculations. The first consists of the homogeneous values of the fields and derivatives. The second consists of the fluctuations of the fields and field derivatives. Finally, in an expanding universe the derivative of the scale factor, or equivalently the Hubble constant, needs to be set. Section 6.3.1 describes the setting of the homogeneous field and derivative components. In section 6.3.2 we derive equations for the field fluctuations and in section 6.3.3 we derive similar expressions for the fluctuations of the derivatives. Section 6.3.4 explains a slight modification of these equations required to preserve isotropy on the lattice. All of the calculations up to this point are done for three dimensional fluctuations. Section 6.3.5 generalizes the results for one and two dimensional lattices. In section 6.3.6 we derive an expression for the initial value of the Hubble constant in physical and program units. Finally section 6.3.7 gives a justification for an approximation used in setting the fluctuations of the derivatives.

Note that while the initializations described here all occur by default, you can choose to override them and use your own initializations in your model file. See section 4.2.2 for more details.

6.3.1 Homogeneous Field and Derivative Values

In a typical run of reheating after chaotic inflation the rescaling constant ϕ_0 will be set to the initial value of the inflaton in Planck units. We have generally found it most convenient to choose the starting point of our simulations at the moment when the program time derivative f'_{pr} is equal to zero. Assuming no other fields have been excited, the inflaton will initially have value one and all other fields zero, and all field derivatives will initially be zero. However, the program allows you to set these initial values however you want. The values you give for these homogeneous modes should be in program variables, as determined by the rescaling parameter A. By default $A = 1/\phi_0$ so if you set ϕ_0 to the initial value of the inflaton and leave the default definition of A you can set $\phi_{pr} = 1$ initially and set any other nonzero initial field values in units of ϕ_0.

6.3.2 Initial Conditions for Field Fluctuations

Although the field equations are solved in configuration space with each lattice point representing a position in space, the initial conditions are set in momentum space and then Fourier transformed to give the initial values of the fields and their derivatives at each grid point. As mentioned above, all of the expressions in this and the following two sections will be derived for a three dimensional lattice. Section 6.3.5 will explain how the results are altered in other dimensions. The Fourier transform F_k in three dimensions is defined by

$$f(x) = \frac{1}{(2\pi)^3} \int d^3k F_k e^{ikx} \tag{6.50}$$

It is assumed that no significant particle production has occurred before the beginning of the program, so quantum vacuum fluctuations are used for setting the initial values of the modes. The probability distribution for the ground state of a real scalar field in a FRW universe is given by [1, 2]

$$P(F_k) \propto exp(-2a^2\omega_k|F_k|^2) \tag{6.51}$$

where

$$\omega_k^2 = k^2 + a^2m^2, \tag{6.52}$$

$$m^2 = \frac{\partial^2 V}{\partial f^2}. \tag{6.53}$$

Although f is a real field the Fourier transform is of course complex, so this probability distribution is over the complex plane. The phase of F_k is uniformly randomly distributed and the magnitude is distributed according to the Rayleigh distribution

$$P(|F_k|) \propto |F_k| exp(-2a^2\omega_k|F_k|^2). \tag{6.54}$$

Note that this distribution gives the mean-squared value

$$< |F_k|^2 >= \frac{1}{2a^2\omega_k}. \tag{6.55}$$

To derive the expressions used for setting field values on the lattice we must modify equation (6.54) to account for a finite, discrete space, then account for the rescalings of field and spacetime variables, and finally discuss how to implement the Rayleigh distribution. In the rest of this section we do each of these in turn.

There are two steps involved in normalizing these modes on a finite, discrete lattice. First this definition has to be adjusted to account for the finite size of the box. This is necessary in order to keep the field values in position space independent of the box size. To see this consider the spatial average $< f^2 >$.

$$f^2 = \frac{1}{(2\pi)^6} \int \int d^3k d^3k' F_k F_{k'} e^{i(k+k')x} \tag{6.56}$$

$$< f^2 >= \frac{1}{L^3 (2\pi)^6} \int \int \int d^3x d^3k d^3k' F_k F_{k'} e^{i(k+k')x} = \frac{1}{L^3} \int d^3k |F_k|^2 \tag{6.57}$$

where L^3 is the volume of the region of integration. So in order to keep $< f^2 >$ constant as L is changed the modes F_k must scale as $L^{3/2}$.

Accounting for the discretization of the lattice is even easier. From the definition of a discrete Fourier transform (denoted here as f_k) in three dimensions

$$f_k \equiv \frac{1}{dx^3} F_k. \tag{6.58}$$

Note that values such as $< f^2 >$ will be affected by changes in the lattice spacing, but this is reasonable since this spacing determines the ultraviolet cutoff of the theory. Without such a cutoff $< f^2 >$ would be divergent.

Putting these effects together gives us the following expression for the rms magnitudes, which we denote by W_k.

$$W_k \equiv \sqrt{< |f_k|^2 >} = \sqrt{\frac{L^3}{2a^2 \omega_k dx^6}} \tag{6.59}$$

At a point (i_1, i_2, i_3) on the Fourier transformed lattice the value of k is given by

$$|k| = \frac{2\pi}{L} \sqrt{i_1^2 + i_2^2 + i_3^2}. \tag{6.60}$$

Next we rescale to program variables. The L, dx, and k rescalings are determined by the rescaling of x in equation (6.2), i.e.

$$L_{pr} = BL; \; dx_{pr} = Bdx; \; k_{pr} = \frac{k}{B}. \tag{6.61}$$

We can define a rescaling $\omega_{k,pr} \equiv \sqrt{k_{pr}^2 + m_{pr}^2} = \omega_k / B$ where $m_{pr} \equiv \frac{am}{B}$. (The extra factor of a appears because the bare values k and m are measured in conformal and physical units respectively.) Then, taking into account the field rescaling $f_{pr} = A a^r f$

$$W_{k,pr}^2 =< |f_{k,pr}|^2 >= \frac{A^2 B^2 a^{2r-2} L_{pr}^3}{2\omega_{k,pr} dx_{pr}^6} \tag{6.62}$$

Meanwhile the rescaled mass is given by

$$m_{pr}^2 = \frac{a^2 m^2}{B^2} = a^{2s+2} \frac{d^2 V_{pr}}{df_{pr}^2} \tag{6.63}$$

Finally it remains to implement the Rayleigh distribution

$$P(|f_{k,pr}|) \propto |f_{k,pr}| exp(-|f_{k,pr}|^2 / W_{k,pr}^2) \tag{6.64}$$

Normalizing this distribution gives

$$P(|f_{k,pr}|) = \frac{2}{W_{k,pr}^2} |f_{k,pr}| e^{-|f_{k,pr}|^2 / W_{k,pr}^2}. \tag{6.65}$$

To generate this distribution from a uniform deviate (i.e. a random number generated with uniform probability between 0 and 1) first integrate it and then take the inverse (see [3]), which gives

$$|f_{k,pr}| = \sqrt{-W_{k,pr}^2 ln(X)} \tag{6.66}$$

where X is a uniform deviate.

There are two more points to note in setting the initial conditions for the fluctuations. The first is simply that the scale factor is set to 1 at the beginning of the calculations and may thus be dropped from the equations. The second is that the phases of all modes are random and uncorrelated, so they are each set randomly. The expression for the field modes is thus

$$f_{k,pr} = e^{i\theta}\sqrt{-W_{k,pr}^2 ln(X)} \tag{6.67}$$

where

$$W_{k,pr} = \frac{ABL_{pr}^{3/2}}{\sqrt{2\omega_{k,pr}}dx_{pr}^3} \tag{6.68}$$

and θ is set randomly between 0 and 2π. The frequency $\omega_{k,pr}$ for a given point (i_1, i_2, i_3) on the momentum space lattice is given by

$$\omega_{k,pr}^2 = \left(\frac{2\pi}{L_{pr}}\right)^2 \left(i_1^2 + i_2^2 + i_3^2\right) + \frac{d^2V_{pr}}{df_{pr}^2}. \tag{6.69}$$

6.3.3 Initial Conditions for Field Derivative Fluctuations

To calculate the field derivatives it is necessary to know the time dependence of the vacuum fluctuations being considered. The full time dependence comes from several sources. First, there is an oscillatory term $e^{\pm i\omega_k t}$. We discuss in section 6.3.4 below the use of the plus or minus sign in this term. Next, all the modes have an extra factor of $1/a$ relative to their Minkowski space values. This can be seen for example from equation (6.55). This extra factor ensures that the physically meaningful quantity $< f(x)^2 >$ depends on the physical rather than the comoving momenta of the modes in the box. There is an additional time dependence that arises from the fact that ω_k is time dependent, both because of the scale factor multiplying the mass term and because of possible time dependence of the effective mass itself. Finally there is the issue of converting to program variables, which can involve rescaling the field values and time coordinates in a time dependent way. Here we calculate the field derivatives taking all of these effects into account except the time dependence of ω_k. Section 6.3.7 explains the justification for this approximation.

Using this approximation the full time dependence of the mode $f_{k,pr}$ is given by the square root of the rms amplitude (equation (6.62)) times $e^{\pm i\omega t}$, i.e.

$$f_{k,pr} \propto a^{r-1}e^{\pm i\omega_k t}. \tag{6.70}$$

Denoting derivatives with respect to the program time t_{pr} with primes we have

$$f_{k,pr}' = B^{-1}a^{-s}\dot{f}_{k,pr} = B^{-1}a^{-s}\left(\pm i\omega_k + (r-1)\frac{\dot{a}}{a}\right)f_{k,pr} = \left(\pm ia^{-s}\omega_{k,pr} + (r-1)\frac{a'}{a}\right)f_{k,pr}. \tag{6.71}$$

Taking the initial scale factor to be one this becomes

$$f_{k,pr}' = \left(\pm i\omega_{k,pr} + (r-1)H_{pr}\right)f_{k,pr} \tag{6.72}$$

where $H_{pr} \equiv \frac{a'}{a}$. The calculation of the initial value of H_{pr} is described in section 6.3.6.

6.3.4 Standing Waves: Preserving Isotropy

Equation (6.70) tells us the frequency of oscillation of the mode f_k, but the question still remains whether we should use the plus or minus sign in the exponential. The answer is that we must use both. This fact arises from a simple property of Fourier transforms, namely that the Fourier transform of a real field f must obey the symmetry

$$f_{-k} = f_k^*. \tag{6.73}$$

(It doesn't matter if you are considering a complex field since you must still then set initial conditions for its real and imaginary parts, and their Fourier transforms will be constrained to obey this same symmetry relation.) We can ignore the expansion of the universe for a moment and imagine that for some mode f_k we have chosen to use the plus sign in the exponential, i.e.

$$\dot{f}_k = i\omega_k f_k. \tag{6.74}$$

However, since both f and $\dot{f}$ are real fields it must be true that

$$\dot{f}_{-k} = \dot{f}_k^* = -i\omega_k f_k^* = -i\omega_k f_{-k}. \tag{6.75}$$

In other words choosing the plus sign for a given momentum k necessarily means using the minus sign for the momentum $-k$. Recall that a mode f_k translates into a function $f(x)$ with spatial dependence e^{-ikx}. So if you use the plus sign in the exponential for some positive k and the minus sign for $-k$ you have effectively initialized the two oscillatory modes

$$f(x) = e^{-i(kx-\omega_k t)} + e^{i(kx-\omega_k t)}. \tag{6.76}$$

In other words you have created a right moving wave. Likewise choosing the minus sign in the exponential for a positive value of k corresponds to setting up a left moving wave. Of course there is no physically preferred direction on the lattice, so in reality your initial conditions should contain equal components of right and left moving fluctuations.

In practice the signs you use for the exponential time dependence of different modes has a negligible effect on the evolution once preheating begins. Even if every mode is initialized to be left-moving, the total momentum this imparts to the field is unnoticeable by the late stages of the evolution in every problem we have considered. Nonetheless it is presumably desirable to enforce Lorentz invariance, at least in an averaged sense. You could do this by randomly initializing each mode with either a plus or a minus sign. Instead, we choose to set up both left and right moving waves with equal amplitude at each value of k. In other words the initial conditions correspond to standing waves. Thus the final form of the initial fluctuations is

$$f_k = \frac{1}{\sqrt{2}}\left(f_{k,1} + f_{k,2}\right) \tag{6.77}$$

$$\dot{f}_k = \frac{1}{\sqrt{2}} i\omega_k \left(f_{k,1} - f_{k,2}\right) - H f_k. \tag{6.78}$$

where $f_{k,1}$ and $f_{k,2}$ are two modes with separate random phases but equal amplitudes determined by equation (6.66).

By now it may have struck you that we seem to be determining these initial conditions based on issues of convenience, symmetry, and so on. What about whatever is the physically correct form for vacuum fluctuations, as given by their quantum mechanical probability distributions? Shouldn't those distributions provide an answer to all of these questions as to the correct form of the equations? The answer is no. Although equation (6.65) gives the correct quantum distribution for the mode amplitudes, it is not correct to use this distribution and then use equation (6.72) to set the values of the field derivatives. The problem is that quantum mechanically f_k and $\dot{f}_k$ are noncommuting operators and can not be simultaneously set. Although this uncertainty presents a problem in principle it is unimportant in practice. Once parametric resonance begins the occupation numbers of the modes f_k become large and their quantum uncertainty becomes irrelevant. Moreover the rapid growth that occurs during this resonance effectively destroys all information about the initial values of the modes so that the final simulation results are insensitive to the details of how the initial conditions are set. In our experience runs that use the probability distribution of equation (6.65) give essentially the same results as ones that use the exact value of equation (6.68) for each mode, and likewise all qualitative results are unchanged by the use of left-moving waves, right-moving waves, or any combination of the two.

6.3.5 Initial Conditions in One and Two Dimensions

To properly solve field theory in a one or two dimensional universe it would be necessary to quantize your fields in this space and derive the expressions for the mode amplitudes and other quantities there. In practice, however, this is not the problem that LATTICEEASY is intended to solve. We are interested in the evolution of fields in a three dimensional space, and the option to do these simulations in lower dimensions is intended to be viewed as a computationally convenient way of solving that same problem. Thus our intention in setting the initial conditions for the modes in these lower dimensions is to duplicate as accurately as possible the field values on a slice through a three dimensional space.

In practice, we approximate such a result by initializing our modes in such as a way as to achieve the same initial value of $\langle f^2 \rangle$ as we would in a three dimensional simulation. To see how this works we denote the D dimensional Fourier transform of the field f as F_{Dk}. Parseval's theorem for a discrete Fourier transform in three dimensions says

$$\sum_{\vec{x}} f(\vec{x})^2 = \frac{1}{N^3} \sum_{\vec{k}} |f_{3k}|^2. \tag{6.79}$$

Averaging over the lattice in position space gives

$$\langle f(\vec{x})^2 \rangle = \frac{1}{N^6} \sum_{\vec{k}} |f_{3k}|^2. \tag{6.80}$$

Given that the field distribution in Fourier space is on average isotropic this sum can be approximated as a one dimensional sum

$$\langle f(\vec{x})^2 \rangle \approx \frac{2\pi}{N^6} \sum_{k} |i|^2 |f_{3k}|^2 \tag{6.81}$$

where $|i|$ is the distance from the origin of the lattice measured in units of the grid spacing. The 2π arises from the 4π that comes from three dimensional integration of an isotropic function, divided by 2 to account for the fact that the one dimensional sum is taken over both positive and negative k. Recall from section 6.3.2 that

$$|k| = \frac{2\pi}{L} |i|, \tag{6.82}$$

so

$$\langle f(\vec{x})^2 \rangle \approx \frac{L^2}{2\pi N^6} \sum_{k} |k| |f_{3k}|^2. \tag{6.83}$$

Comparing this result to Parseval's theorem in one dimension

$$\langle f(\vec{x})^2 \rangle = \frac{1}{N^2} \sum_{k} |f_{1k}|^2 \tag{6.84}$$

gives

$$|f_{1k}|^2 \approx \frac{L^2}{2\pi N^4} k^2 |f_{3k}|^2 = \frac{dx^4}{2\pi L^2} k^2 |f_{3k}|^2. \tag{6.85}$$

Similar logic gives

$$|f_{2k}|^2 \approx \frac{dx^2}{\pi L} |k| |f_{3k}|^2. \tag{6.86}$$

We initialize the modes in one or two dimensions by using the formulas derived above for three dimensions and then adjusting each mode by the factor derived here. Note that these approximations are somewhat inaccurate because the space is cubical rather than spherical. This could be corrected by introducing an extra constant factor to f_{1k} and f_{2k} but in practice we haven't found this difference to be relevant.

6.3.6 The Initial value of the Hubble Constant

The initial value of the Hubble constant is used for setting the field derivatives (equation (6.72)) and as an initial condition for the second order evolution equation for the scale factor. The derivative of a is determined by the equation

$$H^2 = \left(\frac{\dot{a}}{a}\right)^2 = \frac{8\pi}{3} \rho. \tag{6.87}$$

Initially a is set to 1 and

$$H_{pr}^2 = a'^2 = \frac{8\pi}{3B^2} \rho. \tag{6.88}$$

In setting initial values we assume all inhomogeneities are small and thus use only the homogeneous values of the fields $< f >$ and $< \dot{f} >$. Typically the initial field values will be one for the inflaton and zero for all other fields but they can be set to any values by the user. In general, the initial energy density is thus

$$\rho \approx \frac{1}{2} \sum_{fields} \dot{f}^2 + V. \tag{6.89}$$

Converting to program variables

$$\begin{aligned} \rho &= \frac{1}{2}\frac{B^2}{A^2} \sum_{fields} \left(a^{2s-2r} f_{pr}'^2 - 2ra^{2s-2r-1} a' f_{pr} f_{pr}' + r^2 a^{2s-2r-2} a'^2 f_{pr}^2\right) + \frac{B^2}{A^2} a^{2s-2r} V_{pr} \qquad (6.90) \\ &= \frac{B^2}{A^2}\left(\sum_{fields}\left(\frac{1}{2} f_{pr}'^2 - ra' f_{pr} f_{pr}' + \frac{1}{2} r^2 a'^2 f_{pr}^2\right) + V_{pr}\right) \end{aligned}$$

where the second step uses the fact that initially $a = 1$. Since initially $H_{pr} = a'$ we can plug equation (6.90) into equation (6.88) to get an equation we can solve for H_{pr}. Solving this quadratic equation gives

$$H_{pr,0} = \frac{1}{\frac{3A^2}{4\pi} - r^2 f_{pr}^2}\left(-r f_{pr} f_{pr}' + \sqrt{\frac{3A^2}{4\pi} f_{pr}'^2 + 2V_{pr}\left(\frac{3A^2}{4\pi} - r^2 f_{pr}^2\right)}\right) \tag{6.91}$$

where $H_{pr,0}$ refers to the initial value of H_{pr} and each term with field or field derivative values is understood to be summed over all fields.

6.3.7 The Adiabatic Approximation

We noted in section 6.3.3 that the time dependence of the modes comes from their explicit time dependence $f_k \propto e^{\pm i\omega_k t}$, from factors of the scale factor, and from the time dependence of ω_k itself. Using program variables for the fields the time dependence of the modes is given by

$$f_{k,pr} \propto \frac{1}{\sqrt{\omega_k}} a^{r-1} e^{\pm i\omega_k t}. \tag{6.92}$$

Thus the derivative is given by

$$f_{k,pr}' = B^{-1} a^{-s} \dot{f}_{k,pr} = B^{-1} a^{-s}\left[\pm i\omega_k \pm i\dot{\omega}_k t - \frac{1}{2}\frac{\dot{\omega}_k}{\omega_k} + (r-1)\frac{\dot{a}}{a}\right] f_{k,pr} = B^{-1}\omega_k\left[\pm i - \frac{1}{2}\frac{\dot{\omega}_k}{\omega_k^2} + (r-1)\frac{\dot{a}}{\omega_k}\right] f_{k,pr} \tag{6.93}$$

where the last step uses the fact that initially $t = 0$ and $a = 1$. Neglecting the time dependence of ω_k as we did earlier amounts to making the approximation

$$\dot{\omega}_k \ll \omega_k^2, \tag{6.94}$$

which is precisely the condition that ω_k is changing adiabatically. If this condition is not satisfied in the late stages of inflation then gravitational particle production will occur and it will no longer make sense to take the vacuum fluctuations of equation (6.51) as initial conditions.

There's another way to view this condition. Gravitational particle production will occur unless $\omega_k > H$. Since this condition is automatically satisfied for $k > H$ consider the opposite case $k \ll H$, for which $\omega_k \approx am$. Then neglecting the time dependence of m, $\dot{\omega}_k = \dot{a}m = Hm$ when $a = 1$, so the condition $\dot{\omega}_k \ll \omega_k^2$ is equivalent to the condition $m \gg H$. In fact $\dot{\omega}_k \ll \omega_k^2$ is the stronger (and more accurate) condition because it also specifies that m shouldn't be changing rapidly, which would lead to particle production irrespective of the value of H. However, all particle masses should vary slowly during inflation because they should only depend on constants and on the value of the inflaton, which must be changing slowly.

In the case of a field with $m < H$ during inflation the approximation that the field ends inflation in its ground state is no longer valid. In the limit $m \ll H$ the fluctuations of the field produced during inflation can be accurately described by Hankel functions [4]. However in this case the fields will be copiously produced during inflation, leading to severe cosmological problems [5]. For this reason we do not implement these Hankel function solutions in the lattice program. In order to avoid the moduli problem associated with light fields it's best to assume that some mechanism must have given all scalar fields large masses during inflation, in which case equation (6.51) is an accurate expression for the modes at the end of inflation.

6.4 The Staggered Leapfrog Method

LATTICEEASY uses a method called "staggered leapfrog" for solving differential equations. In order to solve a second order (in time) equation you need to store the value of the variable and its first time derivative at each step, and use these to calculate the value of the second time derivative. The idea of staggered leapfrog is to store the variables (i.e. the field values) and their derivatives at different times. Specifically, if the program is using a time step dt and the field values are known at a time t then the derivatives will initially be known at a time $t - dt/2$. Using the field values the program can then calculate the second derivative $\ddot{f}$ at time t and use this to advance $\dot{f}$ to $t + dt/2$. This value of $\dot{f}$ can in turn be used to advance f to $t + dt$, thus restarting the process. Schematically, this looks like

$$\begin{aligned} f(t) &= f(t-dt) + dt\dot{f}(t-dt/2) \\ \dot{f}(t+dt/2) &= \dot{f}(t-dt/2) + dt\ddot{f}[f(t)] \\ f(t+dt) &= f(t) + dt\dot{f}(t+dt/2) \\ &\ldots \end{aligned} \tag{6.95}$$

Because each step advances f or $\dot{f}$ in terms of its derivative at a time in the *middle* of the step this method has higher order accuracy and greater stability than a simple Euler method. However, the method relies on being able to calculate $\ddot{f}$ in terms of f at the time t, so both accuracy and stability are generally lost if $\ddot{f}$ depends on the first derivative $\dot{f}$. This is the reason we choose our rescalings so as to eliminate first derivative terms in the equations of motion. Note that the evolution equation for the scale factor does have a first derivative in it. Section 6.2.2 describes how this problem is solved in the program.

To set the initial conditions for the staggered leapfrog calculations the field values and derivatives must be desynchronized. The initial conditions are set at $t = 0$ and then the fields are advanced by an Euler step of size $dt/2$ to begin the leapfrog. Thereafter all calculations are done in full, staggered steps.

When the program saves output data, however, it is useful to synchronize the data again. The ***save()*** routine is always called when the field values are ahead of the field derivatives, so the function begins by moving the field values backwards by a half step, then calculating and saving all output quantities, and then moving the field values forward again. The one output function that does not use this technique is the checkpointing function that saves an image of the grid. This image is saved with the field values and derivatives desynchronized so that they can be read in again and used to continue the leapfrog calculation.

Chapter 7

CLUSTEREASY: The Parallel Computing Version of LATTICEEASY

CLUSTEREASY is a version of LATTICCEEASY designed to run in parallel on multiple computers. It is written using the the Message Passing Interface (MPI) libraries, which is the standard for parallel computing on distributed memory clusters. To use CLUSTEREASY you will need access to a computer cluster with MPI installed. [1]

If you are using a shared memory system (more than one processor with access to the same physical memory) you do not need to use CLUSTEREASY. Instead you can easily parallelize LATTICEEASY for such systems using the freely available OpenMP libraries. The only changes that you need to make to LATTICEEASY to use OpenMP is to include a new header file and add one line above each of the main evolution loops in `evolution.cpp`. See the documentation for OpenMP for more details.

The physical equations and computational algorithms used by CLUSTEREASY are the same as they are for LATTICEEASY, so we will assume in this section that you are already familiar with the use of LATTICEEASY. If you have any general questions about using LATTICEEASY you should start with the LATTICEEASY documentation, and then look here for issues specific to the parallel version.

The output of CLUSTEREASY is in the exact same format as it is for LATTICEEASY, so it can be read and plotted with the same plotting routines (e.g. the Mathematica notebooks provided with LATTICEEASY.) You can even use the grid images from a serial run to continue the run in parallel and vice-versa.

The rest of this chapter describes the aspects of CLUSTEREASY that are different from LATTICEEASY. Section 7.1 describes how to obtain, compile, and run CLUSTEREASY. Section 7.2 describes the algorithm used for parallelization in CLUSTEREASY. Section 7.3 explains how to modify LATTICEEASY model files to work with CLUSTERREASY. Section 7.4 describes in more detail the new variables and functions you will need to write (or modify) customized output and/or initialization functions for CLUSTEREASY models. Finally, section 7.5 describes some of the tests that we have performed on CLUSTEREASY and explains why the output of CLUSTEREASY may in some cases look different from the output of LATTICEEASY (and why this shouldn't worry you).

7.1 Obtaining, Compiling, and Running CLUSTEREASY

The file structure of CLUSTEREASY is the same as it is for LATTICEEASY, except for the addition of a file called `mpiutil.cpp` with functions for communicating between processors. All of the files have been modified in the parallel version, however, so you need to download the CLUSTEREASY files from the LATTICEEASY download page.

As noted above, CLUSTEREASY requires a cluster using MPI. On such a cluster there should typically be a script called mpiCC for compiling MPI C++ programs. The makefile that comes with CLUSTEREASY uses this command, so if your cluster is set up differently you may need to modify it accordingly.

You will also need a freely available set of Fourier Transform routines called FFTW. You can contact your system administrator if this is not already installed on your cluster. CLUSTEREASY calls FFTW for Fourier Transforms in 2D and 3D. FFTW doesn't have a 1D parallel FFT, so for 1D CLUSTEREASY uses a parallelized version of the

[1]You can actually install and run MPI on a single processor machine for testing and debugging purposes, but to get any actual advantage in speed or memory you need to run it on multiple computers.

FFTEASY routines that come with LATTICEEASY. Note that the parallel FFTEASY routine does not scale well, so large runs in 1D may become very slow.

Note that FFTW can be installed in three different ways: single-precision, double-precision, or both. When it is installed for both, all compiler flags for FFTW must include a letter "s" or "d" that specifies whether single or double precision is being used. By default CLUSTEREASY assumes that FFTW is installed for single and double precision. If it is installed for single precision only on your system you should remove the letter "s" from all the FFTW compiler flags in the makefile and FFTW header names in `latticeeasy.h`.

If FFTW is installed for both precisions and you wish to do runs with double precision, you need to make the following changes

1. Add the line `#define float double` to `latticeeasy.h` and `ffteasy.cpp`, just below the list of header files.
2. Change the letter "s" to "d" in the two FFTW header file names in `latticeeasy.h`.
3. Change the letter "s" to "d" in the FFTW compiler flags in `makefile`.

If you have mpiCC and FFTW set up properly on your system then you should be able to compile CLUSTEREASY simply by typing "make." Note that you do not specify the number of processors when you compile the program, but rather in the command line when you run it. Your cluster should have instructions for executing MPI programs, but on most systems the command is

```
mpirun -np <number of processors> latticeeasy
```

Note that more processors doesn't always mean faster performance. If you use too many processors the program will spend more time communicating between processors than evolving the fields. You may have to do some trial and error for your particular problems, but a good rule of thumb is that you probably won't get much benefit from using more processors than $N/4$, where N is the number of gridpoints along an edge. Also, you will get slightly better performance per processor if the number of processors is a factor of N so that the processors can divide the lattice up evenly.

7.2 Implementation Notes

CLUSTEREASY uses "slab decomposition," meaning the grid is divided along a single dimension (the first spatial dimension). For example, in a 2D run with $N = 8$ on two processors, each processor would cast a 4×8 grid for each field. At each processor the variable `n` stores the local size of the grid in the first dimension, so in this example each processor would store $n = 4$, $N = 8$. Note that `n` is not always the same for all processors, but it generally will be if the number of processors is a factor of `N`.

In practice, the grids are actually slightly larger than $n \times N$ because calculating spatial derivatives at a gridpoint requires knowing the neighboring values, so each processor actually has two additional columns for storing the values needed for these gradients. Continuing the example from the previous paragraph, each processor would store a 6×8 grid for each field. Within this grid the values $i = 0$ and $i = 5$ would be used for storing "buffer" values, and the actual evolution would be calculated in the range $1 \leq i \leq 4$, $0 \leq j \leq 7$.

This scheme is shown in Figure 7.1. At each time step each processor advances the field values in the shaded region, using the buffers to calculate spatial derivatives. Then the processors exchange edge data. At the bottom of the figure I've labeled the i value of each column in the overall grid. During the exchange processor 0 would send the new values at $i_{totalgrid} = 0$ and $i_{totalgrid} = 3$ to processor 1, which would send the values at $i_{totalgrid} = 4$ and $i_{totalgrid} = 7$ to processor 0.

The actual arrays allocated by the program are even larger than this, however, because of the extra storage required by FFTW. When you Fourier Transform the fields the Nyquist modes are stored in extra positions in the last dimension, so the last dimension is $N + 2$ instead of N. The total size per field of the array at each processor is thus typically $n + 2$ in 1D, $(n + 2) \times (N + 2)$ in 2D and $(n + 2) \times N \times (N + 2)$ in 3D. In 2D FFTW sometimes requires extra storage for intermediate calculations as well, in which case the array may be somewhat larger than this, but usually not much. This does not occur in 3D.

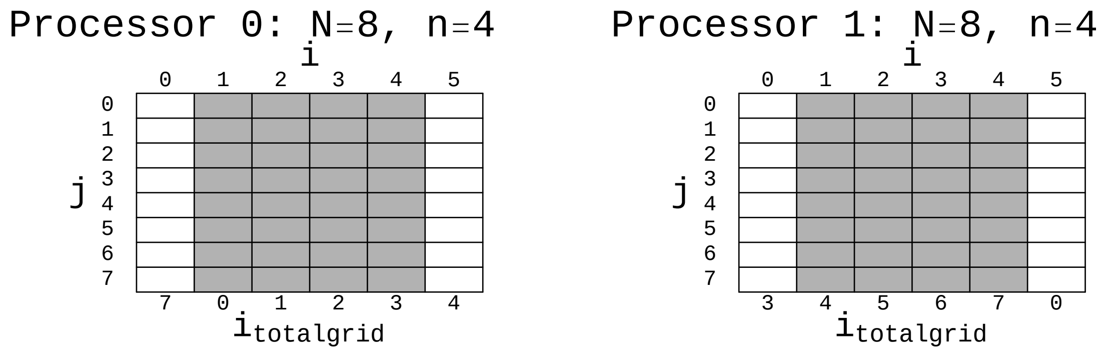

Figure 7.1: Data layout in CLUSTEREASY

7.3 Modifying Model Files for Use with CLUSTEREASY

Model files that you have written (or downloaded) for use with LATTICEEASY can be easily modified to work with CLUSTEREASY. You can probably most easily see the changes that need to be made by running diff on the model files that come with LATTICEEASY and CLUSTEREASY, but I also describe the changes below. They need to be made in two functions.

In the function `potential_energy()` you should add a variable called `result` and initialize it to zero. In the loop over the grid you should replace each instance of `potential` with `result`. Immediately after the loop and before the line where `potential` is divided by `gridsize`, you should insert the line

```
// Sum potentials from all processors to get the average potential for the array.
MPI_Allreduce(&result, &potential, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
```

This will cause each processor the calculate its contribution to the potential in the variable `result` and then combine them all in the variable `potential`.

Similarly, in the function `effective_mass()` you should add a variable called result. Inside the loop over fields (in the case where $field\_values = NULL$) you should replace every instance of `fldsqrd[fld]` with `result`. Immediately after the loop and before the line where you divide `result` by `gridsize` you should insert the line

```
MPI_Allreduce(&result, &(fldsqrd[fld]), 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
```

Again, if anything in this section is unclear simply look in the model file provided with CLUSTEREASY. You should realize, however, that we've only described the changes that need to be made in the generic parts of the model files. If there are model-specific output functions, or complicated calculations done in the function `model_initialize()`, these may need to be parallelized as well. A good rule of thumb is that any time you average or sum a quantity over the entire lattice you will need to modify the code to run in parallel. If you are doing any such modification you should read section 7.4 below, and you may also find it helpful to read section 7.2. If you have any questions about parallelizing your functions for CLUSTEREASY feel free to contact G.F. for help.

You should be able to use LATTICEEASY parameter files with no modification.

7.4 Customizing CLUSTEREASY

LATTICEEASY is written to be easily customizable. The most common ways to insert your own code into a particular model are through the `model_initialize()` and `model_output()` functions. This section lists the most important new features of CLUSTEREASY that you will need to know to write such functions.

First, however, we can give one general warning about output in MPI. It is generally not reliable to have more than one processor writing data to the same file. Even if you are careful to ensure that the processors write the data in the correct order, file caching in the operating system can still cause unexpected results. We found that the only reliable method of writing to a file was to copy the output data to the root processor and do all the writing from there. (You could also have each processor write to a separate file, but we wanted to produce output whose format didn't depend on the number of processors.)

7.4.1 CLUSTEREASY Variables

The new global variables introduced in CLUSTERREASY include

numprocs The number of processors. (This is determined by the mpirun command or the batch script you use to execute the program.)

my_rank The rank of each processor. It runs from 0 to *numprocs* − 1.

n The size of the local grid in the x direction at each processor. The sum of `n` over all processors should equal `N`.

my_start_position The starting point of each processor's grid in the x direction. This will always be zero for the root processor ($my\_rank = 0$). For the next processor it will be equal to the value of `n` on the root processor, and so on.

fstore This is an extra array used by the root processor to gather field data from other processors. The size of this array is the size of the largest single-field array held by any processor. Note that this array is only cast when checkpointing is turned on or when slices of the same dimension as the grid are being output.

fstore_size Size of the above array

buffers_up_to_date This variable indicates whether the buffers at each processor are up to date (1) or not (0). See section 7.4.3 below.

7.4.2 Looping Over the Grid

As described in section 7.2, each processor stores `n` values in the i direction and `N` values in each other direction. In addition, each processor stores one extra "buffer" value of i at each end of its grid for calculating spatial derivatives. To loop over all gridpoints in 3D, for example, you would write

$$for(i = 1; i \leq n; i++) \quad for(j = 0; j < N; j++) \quad for(k = 0; k < N; k++) \tag{7.1}$$

The macro `LOOP` automatically does this looping correctly. If you are calculating the average of a quantity across the entire lattice you will need to gather the results from all the processors after the loop is completed. The simplest way to do this is with the functions `MPI_Reduce()` or `MPI_Allreduce()`. These are explained in any standard reference on MPI, and you can find examples of them in CLUSTEREASY.

7.4.3 Updating Buffers

Any time you need to calculate field gradients you need to ensure that the field buffers are up to date. This is done through the variable `buffers_up_to_date`. If you write a function that changes the values of the fields you should set `buffers_up_to_date` to zero. This includes any function that Fourier transforms the fields because the Fourier Transform routines can overwrite the parts of the arrays used for the buffers.

If you write a function that requires the buffers - typically because it calculates spatial derivatives - you should check if they are up to date and call the function `update_buffers()` if they are not. Note, however, that if you are calling the LATTICEEASY function `gradient_energy()` you don't need to do this check because it's performed inside that function.

7.4.4 Fourier Transforms

In 1D the syntax for a Fourier Transform is essentially the same as it is in LATTICEEASY, except that you must include the parameters `my_rank` amd `numprocs`. You should also remember that the field values start at $i = 1$. For example, to do a forward Fourier Transform of a single field you would write

```
fftr1(&(f[fld][1]),N,1,my_rank,numprocs);
```

In 2D and 3D CLUSTEREASY uses the Fourier Transform package FFTW, so the syntax for a Fourier Transform is different. First, you must declare a variable of type rfftwnd_mpi_plan to store the "plan" that FFTW uses to calculate the Fourier Transform. Next you must create this plan with the command

`<my_plan_name> = rfftw3d_mpi_create_plan(MPI_COMM_WORLD, N, N, N, FFTW_REAL_TO_COMPLEX, FFTW_ESTIMATE);`
(In 2D you would write "2d" instead of "3d" and only include the argument *N* twice.) If you are planning to do a forward transform and an inverse transform you will need a separate plan for each, and for the inverse plan you should write "COMPLEX_TO_REAL" instead of "REAL_TO_COMPLEX." You do not need a separate plan for each field, or for the fields and derivatives. Finally, to take the Fourier Transform you execute the command
`rfftwnd_mpi(<my_plan_name>, 1, &(f[fld][1][0][0]), NULL, FFTW_NORMAL_ORDER);`
In 2D you would simply write *f[fld][1][0]* instead of *f[fld][1][0][0]*.

7.5 Accuracy Tests of CLUSTEREASY

CLUSTEREASY solves the same equations as LATTICEEASY and produces the same outputs in the same format. Nonetheless, if you run the same model and parameters with the two programs you will find that the results differ. There are several reasons why this occurs. First, the initial amplitudes and phases of the modes in LATTICEEASY are set using random numbers. In CLUSTEREASY each processor generates and uses a separate random number sequence for the initial conditions, so the initial amplitudes and phases of the modes will be different for different numbers of processors. Second, LATTICEEASY uses FFTEASY for Fourier Transforms while CLUSTEREASY uses FFTW. In principle the two routines should produce identical output, but in practice they have different (but comparable) roundoff errors. Third, the calculations of gradient energy and potential energy both involve averaging quantities over the lattice. When this is done in parallel the order of some operations in this averaging is done differently, which again leads to slightly different roundoff errors. Without expansion this fact only affects the energy output function, while in runs with expansion the field evolution is also slightly affected.

When all of these effects are put together, the final numbers that appear in the output files at the end of a run can look completely different in serial and parallel runs, or in two parallel runs with different numbers of processors. Nonetheless, all of these sets of data should be physically valid. Recall that the field evolution being simulated by LATTICEEASY is a representative sample from an ensemble of possible physical realizations of the fields, as encoded in the random initial conditions. In all of the tests we have run all physical quantities (energies, occupation numbers, etc.) come out the same in all runs, whereas details such as the locations of peaks in the field distribution generally do not come out the same. We have done extensive tests to verify that the differences between serial and parallel runs (or parallel runs with different numbers of processors) are equivalent to the differences that come from doing two identical runs with different random number seeds.

Finally, however, we should note the one actual difference in the output of LATTICEEASY and CLUSTEREASY. The option ***sliceaverage*** (described in the LATTICEEASY documentation above) is sometimes disabled in CLUSTEREASY. Specifically, if $slicedims < NDIMS$ then the slices are all computed on a single processor and ***sliceaverage*** is enabled as usual, but if $slicedims = NDIMS$ then the slices come from multiple processors and averaging won't be done.

Chapter 8

Terms of Use

LATTICEEASY consists of the C++ files `latticeeasy.cpp`, `initialize.cpp`, `evolution.cpp`, `output.cpp`, `latticeeasy.h`, and `parameters.h`. CLUSTEREASY includes the additional file `mpiutil.cpp`. (The distribution also includes the file `ffteasy.cpp` but this file is distributed separately and therefore not considered part of the LATTICEEASY distribution in what follows.) LATTICEEASY is free. We are not in any way, shape, or form expecting to make money off of these routines. We wrote them for the sake of doing good science and we're putting them out on the Internet in case other people might find them useful. Feel free to download them, incorporate them into your code, modify them, translate the comment lines into Swahili, or whatever else you want. What we do want is the following:

1. Leave this notice (i.e. this entire paragraph beginning with "LATTICEEASY consists of..." and ending with our email addresses) in with the code wherever you put it. Even if you're just using it in-house in your department, business, or wherever else we would like these credits to remain with it. This is partly so that people can...

2. Give us feedback. Did LATTICEEASY work great for you and help your work? Did you hate it? Did you find a way to improve it, or translate it into another programming language? Whatever the case might be, we would love to hear about it. Please let us know at the email addresses below.

3. Finally, insofar as we have the legal right to do so we forbid you to make money off of this code without our consent. In other words if you want to publish these functions in a book or bundle them into commercial software or anything like that contact us about it first. We'll probably say yes, but we would like to reserve that right.

For any comments or questions you can reach us at
gfelder@email.smith.edu
Igor.Tkachev@cern.ch

Enjoy LATTICEEASY!

Gary Felder and Igor Tkachev

Bibliography

[1] D. Polarski, and A. Starobinsky, Semiclassicality and Decoherence of Cosmological Perturbations, Class. Quant. Grav. **13**, 377 (1996), gr-qc/9504030.

[2] S. Khlebnikov, and I. Tkachev, Classical Decay of Inflaton, Phys. Rev. Lett. **77**, 219-222 (1996), hep-ph/9603378.

[3] Press, William, et al, *Numerical Recipes in C: The Art of Scientific Computing, Second Edition* (Cambridge University Press, Melbourne, Australia, 1992).

[4] A. D. Linde, *Particle Physics and Inflationary Cosmology* (Harwood, Chur, Switzerland, 1990).

[5] G. Felder, L. Kofman, and A. Linde, Gravitational Particle Production and the Moduli Problem, JHEP 0002:027 (2000), hep-ph/9909508.