

lite-cosmic-sim

Generated by Doxygen 1.10.0

1 lite-cosmic-sim	1
1.1 Overview	1
1.2 Sample usage	1
1.3 How to get and build the project	2
1.4 Documentation	3
1.5 Convenience utilities for visualizing output	3
1.6 Overview of implemented functionalities	3
1.7 Notes on using CUDA	3
2 Implementing your own equation	5
2.1 Adding the equation class	5
2.2 Adding the coupling parameter in workspace	6
2.3 Setting <code>workspace.kappa</code> from a parameter struct	6
2.4 Add a function to compute energy density	7
2.5 Using CUDA	7
3 Concept Index	9
3.1 Concepts	9
4 Hierarchical Index	11
4.1 Class Hierarchy	11
5 Class Index	13
5.1 Class List	13
6 File Index	15
6.1 File List	15
7 Concept Documentation	17
7.1 HasFa Concept Reference	17
7.1.1 Concept definition	17
7.2 HasFRWParameters Concept Reference	17
7.2.1 Concept definition	17
7.3 HasLambda Concept Reference	17
7.3.1 Concept definition	17
7.4 HasLatticeParams Concept Reference	17
7.4.1 Concept definition	17
7.5 HasMass Concept Reference	18
7.5.1 Concept definition	18
7.6 HasPsiApproximationParameters Concept Reference	18
7.6.1 Concept definition	18
7.7 LatticeEquationConcept Concept Reference	18
7.7.1 Concept definition	18
8 Class Documentation	19

8.1 ComovingCurvatureEquationInFRW Struct Reference	19
8.1.1 Detailed Description	20
8.1.2 Member Function Documentation	20
8.1.2.1 compute_energy_density()	20
8.2 ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density > Struct Template Reference	20
8.2.1 Detailed Description	21
8.2.2 Member Data Documentation	22
8.2.2.1 t_end	22
8.2.2.2 t_interval	22
8.2.2.3 t_start	22
8.3 CudaApproximateComovingCurvatureEquationInFRW Struct Reference	22
8.4 CudaComovingCurvatureEquationInFRW Struct Reference	23
8.5 CudaFixedCurvatureEquationInFRW Struct Reference	24
8.6 CudaKleinGordonEquationInFRW Struct Reference	24
8.7 CudaLambdaEquationInFRW Struct Reference	25
8.8 CudaSqrtPotentialEquationInFRW Struct Reference	25
8.9 cufftWrapper Struct Reference	26
8.9.1 Detailed Description	26
8.10 cufftWrapperBatchedD2Z Struct Reference	27
8.11 cufftWrapperD2Z Struct Reference	27
8.12 cufftWrapperNoBatching Struct Reference	27
8.12.1 Detailed Description	28
8.13 empty Struct Reference	28
8.13.1 Detailed Description	28
8.14 fftWrapperDispatcher< Vector > Struct Template Reference	29
8.14.1 Detailed Description	29
8.15 fftWrapperDispatcher< Eigen::VectorXd > Struct Reference	29
8.16 fftWrapperDispatcher< thrust::device_vector< double > > Struct Reference	29
8.17 fftwWrapper Struct Reference	30
8.17.1 Detailed Description	30
8.17.2 Constructor & Destructor Documentation	30
8.17.2.1 fftwWrapper()	30
8.17.3 Member Function Documentation	30
8.17.3.1 execute_batched_d2z()	30
8.17.3.2 execute_d2z()	31
8.17.3.3 execute_inplace_z2d()	31
8.17.3.4 execute_z2d() [1/2]	31
8.17.3.5 execute_z2d() [2/2]	32
8.18 KGParam Struct Reference	32
8.19 KleinGordonEquation Struct Reference	33
8.19.1 Detailed Description	33

8.19.2 Member Function Documentation	33
8.19.2.1 compute_energy_density()	33
8.19.2.2 operator>()	34
8.20 KleinGordonEquationInFRW Struct Reference	34
8.20.1 Detailed Description	35
8.20.2 Member Function Documentation	35
8.20.2.1 compute_energy_density()	35
8.21 midpoint< State, Value, Deriv, Time, Algebra, Operations, Resizer > Class Template Reference	35
8.22 MyParam Struct Reference	36
8.23 SampleParam Struct Reference	37
8.23.1 Detailed Description	37
8.23.2 Member Data Documentation	37
8.23.2.1 a1	37
8.23.2.2 H1	37
8.23.2.3 k_ast	37
8.23.2.4 L	37
8.23.2.5 lambda	38
8.23.2.6 m	38
8.23.2.7 N	38
8.23.2.8 t1	38
8.23.2.9 varphi_std_dev	38
8.24 StaticEOSCosmology Struct Reference	38
8.24.1 Detailed Description	39
8.24.2 Member Function Documentation	39
8.24.2.1 eta()	39
8.24.3 Member Data Documentation	39
8.24.3.1 a1	39
8.24.3.2 H1	40
8.24.3.3 p	40
8.24.3.4 t1	40
8.25 WKBSolutionForKleinGordonEquationInFRW Struct Reference	40
8.26 WorkspaceGeneric< Vector > Struct Template Reference	41
8.26.1 Detailed Description	41
9 File Documentation	43
9.1 src/cuda_wrapper.cuh File Reference	43
9.1.1 Detailed Description	44
9.2 cuda_wrapper.cuh	44
9.3 src/dispatcher.hpp File Reference	45
9.3.1 Detailed Description	46
9.4 dispatcher.hpp	46
9.5 src/eigen_wrapper.hpp File Reference	46

9.5.1 Detailed Description	47
9.6 eigen_wrapper.hpp	47
9.7 src/equations.hpp File Reference	47
9.7.1 Detailed Description	48
9.8 equations.hpp	48
9.9 src/equations_cuda.cuh File Reference	49
9.9.1 Detailed Description	49
9.10 equations_cuda.cuh	50
9.11 src/fdm3d.hpp File Reference	51
9.11.1 Detailed Description	52
9.11.2 Macro Definition Documentation	52
9.11.2.1 PADDED_IDX_OF	52
9.11.3 Function Documentation	52
9.11.3.1 compute_cutoff_fouriers()	52
9.11.3.2 compute_field_with_scaled_fourier_modes()	52
9.11.3.3 compute_inverse_laplacian()	53
9.11.3.4 compute_mode_power_spectrum()	53
9.11.3.5 compute_power_spectrum()	55
9.12 fdm3d.hpp	56
9.13 src/fdm3d_cuda.cuh File Reference	56
9.13.1 Detailed Description	57
9.14 fdm3d_cuda.cuh	57
9.15 src/fftw_wrapper.hpp File Reference	58
9.15.1 Detailed Description	58
9.16 fftw_wrapper.hpp	58
9.17 field_booster.hpp	59
9.18 src/initializer.hpp File Reference	59
9.18.1 Detailed Description	60
9.18.2 Variable Documentation	60
9.18.2.1 homogeneous_field	60
9.18.2.2 homogeneous_field_with_fluctuations	60
9.18.2.3 perturbed_grf	61
9.18.2.4 perturbed_grf_and_comoving_curvature_fft	61
9.18.2.5 perturbed_grf_without_saving_Psi	61
9.18.2.6 plane_wave	62
9.18.2.7 unperturbed_grf	62
9.18.2.8 unperturbed_grf_and_fixed_curvature	62
9.18.2.9 unperturbed_grf_with_background	63
9.18.2.10 unperturbed_grf_with_Psi	63
9.18.2.11 wave_packet	63
9.19 initializer.hpp	64
9.20 src/io.hpp File Reference	67

9.20.1 Detailed Description	68
9.21 io.hpp	68
9.22 src/midpoint.hpp File Reference	68
9.22.1 Detailed Description	68
9.23 midpoint.hpp	69
9.24 src/observer.hpp File Reference	70
9.24.1 Detailed Description	70
9.25 observer.hpp	71
9.26 src/param.hpp File Reference	72
9.26.1 Detailed Description	72
9.27 param.hpp	73
9.28 src/physics.hpp File Reference	74
9.28.1 Detailed Description	74
9.29 physics.hpp	74
9.30 src/random_field.hpp File Reference	75
9.30.1 Detailed Description	76
9.30.2 Function Documentation	76
9.30.2.1 broken_power_law_given_amplitude_3d()	76
9.30.2.2 generate_inhomogeneous_gaussian_random_field()	77
9.30.2.3 power_law_with_cutoff_given_amplitude_3d()	77
9.30.2.4 scale_invariant_spectrum_3d()	78
9.31 random_field.hpp	78
9.32 special_function.hpp	79
9.33 src/utility.hpp File Reference	79
9.33.1 Detailed Description	80
9.34 utility.hpp	80
9.35 src/wkb.hpp File Reference	81
9.35.1 Detailed Description	81
9.36 wkb.hpp	81
9.37 src/workspace.hpp File Reference	82
9.37.1 Detailed Description	82
9.38 workspace.hpp	83
Index	85

Chapter 1

lite-cosmic-sim

lite-cosmic-sim is a lightweight and modular framework for performing field simulations in cosmology. This framework was used for studying free-streaming of wave dark matter; see [arXiv:XXXX.XXXX](#) for the study and these [youtube videos](#) for visualization. The codebase contains several field equations on both CPU and GPU (CUDA), offering choices for numerical methods and simulation outputs.

1.1 Overview

This codebase aims to be:

1. As fast as possible. Users should be able to write code that exhausts hardware potential within this framework.
2. Easily modifiable and extensible. Users should be able to focus on physics-relevant code, such as that for setting initial conditions or the field equation.

To achieve these goals, the framework is written in a modular structure. This allows users to easily switch between different initial conditions, field equations, output methods, and even between using CPUs or GPUs for computation. Users have to and only have to provide the low level implementation for the physics-relevant code. This means users have full control over optimization of core routines, and they are not limited to a specific set of provided features. This flexibility makes it easy for the user to test new ideas, which is useful in research.

1.2 Sample usage

The following code initializes a homogeneous Klein Gordon field with (initially) unit field strength and zero time derivative. Then the field is evolved from $t=0$ to $t=10$. Field and density spectra are saved to disk per unit time.

```
++
#include "param.hpp"
#include "initializer.hpp"
#include "equations.hpp"
#include "observer.hpp"

struct MyParam {
    long long int N = 256; // Number of lattice sites (per axis)
    double L = 256.0; // Box size
    double m = 1.0; // Field mass
    double f = 1.0; // The initial (homogeneous) field value
    double dt_f = 0.0; // The initial (homogeneous) field time derivative value
    double t_start = 0.0; // Start time of numerical integration
    double t_end = 10.0; // End time of numerical integration
    double t_interval = 1.0; // Interval between saving outputs
}
```

```
};

int main() {
    using namespace Eigen;
    using namespace boost::numeric::odeint;

    typedef KleinGordonEquation Equation;
    typedef Eigen::VectorXd State;
    typedef WorkspaceGeneric<State> Workspace;

    MyParam param;

    Workspace workspace(param, homogeneous_field);

    Equation eqn(workspace);

    ConstIntervalObserver<Equation> observer("output/sample_equation/", param, eqn);

    auto stepper = runge_kutta4_classic<State>();

    integrate_const(stepper, eqn, workspace.state, param.t_start, param.t_end, 0.1, observer);
}
```

Here's a break down of the code:

- `MyParam` is a POD struct specifying parameters for the simulation. You may define your own struct to include new parameters (coupling strength, FRW universe parameters, time step size, etc), as long as it is a POD and contains lattice parameters `N` and `L`.
- `Workspace` is a type containing temporary variables for a simulation (e.g. the field). It is initialized with `param` and a callback `homogeneous_field`, which sets the field to homogeneous value `param.f` and time derivative `param.dt_f`. You can easily define your own callbacks (using lambdas) to set other sorts of initial conditions.
- `Equation` is the equation to be solved. Here it is the pre-defined `KleinGordonEquation`. You can of course write your own equations.
- `ConstIntervalObserver<Equation>` specifies how to save outputs during simulation. By default it saves spectra for field and density.
- `stepper` is the RK4 method provided by the `boost odeint` library. You can choose other methods (e.g. Euler, DOPRI5) in the library, or even write your own. The `odeint` library is responsible for the main numerical integration loop in this codebase.
- `integrate_const` is a convenience function in the `odeint` library. It runs the simulation and saves results to "output/sample_equation", as specified by `observer`.

1.3 How to get and build the project

Compiler requirement: a C++ compiler supporting C++20. (I used `g++ 12.2.0`.)

Required dependency: `fftw3`

Optional dependency: `CUDA Toolkit`

I also included header-only libraries `Eigen 3.4.0` and `boost 1.84` along with the codebase in the external directory.

`Makefile` is used for build system. I have tested compilation on Linux and MacOS systems. To compile the project:

- Download the project with (for example) `git clone https://github.com/hypermania/lite-cosmic-sim`.

- (If default settings don't work:) Modify the `Makefile` so that it knows where your `fftw` or `CUDA` include files / library files are.
- If you have `CUDA Toolkit` installed, simply run `make -j`.
- If you don't have `CUDA Toolkit`, run `make -j disable-cuda=true`. (I use compiler flags to comment out `CUDA`-dependent code. e.g. [CudaComovingCurvatureEquationInFRW](#))

Note: If you have a `CUDA` compatible `NVIDIA GPU`, using `CUDA` is highly recommended. In our case, it produced more than 10 times speedup.

1.4 Documentation

LaTeX version of documentation is in `documentation.pdf`. If you have `doxygen`, you can also build an html version by running `doxygen doxygen.config`.

1.5 Convenience utilities for visualizing output

Two `Mathematica` notebooks `spectra.nb`, `snapshots.nb` and a `python` script `plot_util.py` are included for visualizing outputs from the program. By default, running the entire notebook / `python` script will read sample data from `output/Growth_and_FS` and produce `spectra` and `snapshots`. If you generate new outputs from the program, you just need to change `dir` or `project_dir` variables to the new output directory.

1.6 Overview of implemented functionalities

Symbol	Description
<code>generate_inhomogeneous_gaussian_↔ random_field</code>	Function for initializing Gaussian random fields with spatially inhomogeneous variances. This procedure is crucial for generating the initial conditions used in the paper.
KleinGordonEquationInFRW and CudaKleinGordonEquationInFRW	Klein Gordon equation that runs on CPU and GPU. Used in section 4.2.1 of paper.
ComovingCurvatureEquationInFRW , CudaComovingCurvatureEquationInFRW and CudaApproximateComovingCurvatureEquationInFRW	A scalar field in the presence of external gravity that is consistent with some set of comoving curvature perturbations. Used in section 4.2.2 of paper.
CudaSqrtPotentialEquationInFRW	A scalar field with monodromy potential. Used in section 4.2.3 of paper.
CudaFixedCurvatureEquationInFRW	A scalar field in a fixed gravitational potential.
CudaLambdaEquationInFRW	A scalar field with $\lambda \phi^4$ interaction.

1.7 Notes on using CUDA

We do separate compilation of `.cpp` files and `.cu` files; `.cu` files are automatically compiled by `nvcc`, whereas `.cpp` files are compiled by the host compiler. We use the `thrust` library (included with `CUDA Toolkit`) extensively, with field state vectors having type `thrust::device_vector<double>`. Initialization procedures usually prepare some profile on the CPU and then copy it to `device_vector<double> state` in the workspace.

A straightforward way to use CUDA for a simulation is to implement an `Equation` class with `thrust::device_vector<double>` as state vector. You will probably need to write your own CUDA kernels for that purpose. See `equations_cuda.cu` for some examples. Don't worry about adapting CUDA with the numerical integrators (e.g. RK4); the files in `src/odeint_thrust` will take care of that automatically.

Chapter 2

Implementing your own equation

Here we give an example of adding a field equation with $\kappa\varphi^6$ interaction to the codebase.

$$\ddot{\varphi} - \nabla^2 \varphi + m^2 \varphi + \kappa \varphi^5 = 0$$

2.1 Adding the equation class

We use the boost odeint library for numerical integration. To use the library, we need to implement a new equation class. See [this link](#) for an example of odeint equation class. In our case, the equation class with φ^6 looks like:

```
struct KappaEquation {
    typedef Eigen::VectorXd Vector;
    typedef Vector State;
    typedef WorkspaceGeneric<State> Workspace;
    Workspace &workspace;

    KappaEquation(Workspace &workspace_) : workspace(workspace_) {}

    void operator()(const State &dxdt, State &x, const double t);
};
```

In the first few lines, the types `Vector`, `State` and `Workspace` are defined. These definitions specify what state vector the equation is going to work with: if you want to use different state vector types (e.g. GPU device vector), you will need to define different equation classes. Here we use `Eigen::VectorXd`. The equation class also has a reference to a workspace, so that it has access to essential information for evolution (e.g. mass and coupling parameters).

The most important function here is the `operator()`. When this function is called, it computes the time derivative of the state vector `x` at time `t`, and stores it to `dxdt`. Implementing this function is the minimal requirement for a class to work with the odeint library. To do this, we can simply copy the implementation for `KleinGordonEquation::operator()` and add a $\kappa\varphi^5$ term to it:

```
void KappaEquation::operator()(const State &x, State &dxdt, const double t)
{
    using namespace Eigen;
    const long long int N = workspace.N;
    const double L = workspace.L;
    const double m = workspace.m;
    const double kappa = workspace.kappa;
    const double inv_h_sqr = 1.0 / ((L / N) * (L / N));

    for(long long int a = 0; a < N; ++a){
        for(long long int b = 0; b < N; ++b){
            dxdt(seqN(IDX_OF(N, a, b, 0), N)) = x(seqN(N*N*N+IDX_OF(N, a, b, 0), N));
            dxdt(seqN(N*N*N+IDX_OF(N, a, b, 0), N)) =
                (-1.0) * m * m * x(seqN(IDX_OF(N, a, b, 0), N))
                - kappa * pow(x(seqN(IDX_OF(N, a, b, 0), N)), 5)
        }
    }
```

```

+ inv_h_sqr * (-6.0 * x(seqN(IDX_OF(N, a, b, 0), N))
+ x(seqN(IDX_OF(N, (a+1)%N, b, 0), N))
+ x(seqN(IDX_OF(N, (a+N-1)%N, b, 0), N))
+ x(seqN(IDX_OF(N, a, (b+1)%N, 0), N))
+ x(seqN(IDX_OF(N, a, (b+N-1)%N, 0), N)));
dxdt(seqN(N*N+N+IDX_OF(N, a, b, 1), N-2)) +=
inv_h_sqr * ( x(seqN(IDX_OF(N, a, b, 2), N-2))
+ x(seqN(IDX_OF(N, a, b, 0), N-2)) );
dxdt(N*N+N+IDX_OF(N, a, b, 0)) +=
inv_h_sqr * ( x(IDX_OF(N, a, b, N-1)) + x(IDX_OF(N, a, b, 1)) );
dxdt(N*N+N+IDX_OF(N, a, b, N-1)) +=
inv_h_sqr * ( x(IDX_OF(N, a, b, N-2)) + x(IDX_OF(N, a, b, 0)) );
}
}
}

```

Note the extra line `kappa * pow(x(seqN(IDX_OF(N, a, b, 0), N)), 5)` giving the $\kappa\varphi^5$ term in the equation.

2.2 Adding the coupling parameter in workspace

The code given above won't compile yet since `workspace.kappa` doesn't exist. To make the code compile, add a new field in `WorkspaceGeneric`:

```

template<typename Vector>
struct WorkspaceGeneric {
    // Stuff
    double kappa;
    // Stuff
};

```

As a general paradigm, we put data (e.g. coupling parameters, temporary variables) needed to solve the equation in a `Workspace`. Note that different equations use the same `Workspace`, and the field names (e.g. `kappa`) can mean different things for different equations. You are responsible for ensuring that your modification on `Workspace` doesn't introduce bugs for other equations. To avoid accidentally introducing a bug, you are advised to add new fields for new parameters / temporary objects.

2.3 Setting `workspace.kappa` from a parameter struct

Now suppose you define a new parameter class:

```

struct KappaParam {
    // The usual
    double kappa;
};

KappaParam param;

```

If you try calling the constructor `Workspace(param, initializer)`, `workspace.kappa` would not be automatically set to `param.kappa`. To resolve this, add the following in `workspace.hpp`:

```

template<typename Param>
concept HasKappa = requires (Param param) { TYPE_REQUIREMENT(param.kappa, double) };

// ...

WorkspaceGeneric(const Param &param, auto &initializer) :
    N(param.N), L(param.L), fft_wrapper(param.N)
{
    if constexpr (HasKappa<Param>) { kappa = param.kappa; }
    // ...
}

```

This piece of code uses concept `HasKappa` to detect if `param.kappa` exists or not, and set `workspace.kappa` to `param.kappa` in the case it exists. Having `KappaParam` is useful since it works with the utilities in `param.h`.

2.4 Add a function to compute energy density

In order to save density spectrum, you would also want to implement a function to calculate energy density profile. Again we can imitate the implementation for [KleinGordonEquation](#):

```
struct KappaEquation {
    // ...
    static Vector compute_energy_density(const Workspace &workspace, const double t);
};

KappaEquation::Vector KappaEquation::compute_energy_density(const Workspace &workspace, const double t)
{
    using namespace Eigen;
    const long long int N = workspace.N;
    const double L = workspace.L;
    const double m = workspace.m;
    const double kappa = workspace.kappa;
    const double inv_h_sqr = 1.0 / ((L / N) * (L / N));

    VectorXd rho(workspace.state.size() / 2);

    for(long long int a = 0; a < N; ++a){
        for(long long int b = 0; b < N; ++b){
            rho(seqN(IDX_OF(N, a, b, 0), N)) = 0.5 *
            ( workspace.state(seqN(N*N*N+IDX_OF(N, a, b, 0), N)).cwiseAbs2()
              + m * m * workspace.state(seqN(IDX_OF(N, a, b, 0), N)).cwiseAbs2()
              + (1.0 / 6.0) * kappa * pow(workspace.state(seqN(IDX_OF(N, a, b, 0), N)), 6)
              + 0.25 * inv_h_sqr *
                ( workspace.state(seqN(IDX_OF(N, (a+1)%N, b, 0), N))
                  - workspace.state(seqN(IDX_OF(N, (a+N-1)%N, b, 0), N)).cwiseAbs2()
                  + workspace.state(seqN(IDX_OF(N, a, (b+1)%N, 0), N))
                  - workspace.state(seqN(IDX_OF(N, a, (b+N-1)%N, 0), N)).cwiseAbs2() )
            );
            rho(seqN(IDX_OF(N, a, b, 1), N-2)) += 0.5 * 0.25 * inv_h_sqr *
            (workspace.state(seqN(IDX_OF(N, a, b, 2), N-2))
              - workspace.state(seqN(IDX_OF(N, a, b, 0), N-2)).cwiseAbs2() );
            rho(IDX_OF(N, a, b, 0)) += 0.5 * 0.25 * inv_h_sqr *
            pow(workspace.state(IDX_OF(N, a, b, 1)) - workspace.state(IDX_OF(N, a, b, N-1)), 2);
            rho(IDX_OF(N, a, b, N-1)) += 0.5 * 0.25 * inv_h_sqr *
            pow(workspace.state(IDX_OF(N, a, b, 0)) - workspace.state(IDX_OF(N, a, b, N-2)), 2);
        }
    }
    return rho;
}
```

Note the extra $\kappa\varphi^6/6$ term in the function above. Now [ConstIntervalObserver](#) knows how to compute the energy density for this theory.

2.5 Using CUDA

If you want your equation to run on GPU memory, then in [KappaEquation](#) the vector type should be set to:

```
typedef thrust::device_vector<double> Vector;
```

Here, `thrust::device_vector<double>` is a class in the `thrust` library representing a double floating point array on the GPU. Much like `std::vector<double>`, the class `thrust::device_vector<double>` takes care of GPU memory allocation / deallocation in an RAII manner, so that you don't have to call CUDA memory management API directly. See [thrust documentation](#) for more details.

Your `operator()` and `compute_energy_density` functions must now work on GPU device vectors. A straightforward way to do this is to write your own CUDA kernel. Here's an example on how to do it:

```
__global__
void kappa_equation_kernel(const double *x, double *dxdx,
    const double m, const double kappa,
    const double inv_h_sqr,
    const long long int N)
{
    int a = blockIdx.x;
    int b = blockIdx.y;
    int c = threadIdx.x;

    dxdx[IDX_OF(N, a, b, c)] = x[N*N*N+IDX_OF(N, a, b, c)];
    dxdx[N*N*N+IDX_OF(N, a, b, c)] =
        - m * m * x[IDX_OF(N, a, b, c)]
```

```

- kappa * x[IDX_OF(N, a, b, c)] * x[IDX_OF(N, a, b, c)] * x[IDX_OF(N, a, b, c)] * x[IDX_OF(N, a, b, c)]
  * x[IDX_OF(N, a, b, c)]
+ inv_h_sqr * (-6.0 * x[IDX_OF(N, a, b, c)]
  + x[IDX_OF(N, (a+1)%N, b, c)]
  + x[IDX_OF(N, (a+N-1)%N, b, c)]
  + x[IDX_OF(N, a, (b+1)%N, c)]
  + x[IDX_OF(N, a, (b+N-1)%N, c)]
  + x[IDX_OF(N, a, b, (c+1)%N)]
  + x[IDX_OF(N, a, b, (c+N-1)%N)]);
}

void KappaEquation::operator()(const State &x, State &dxdt, const double t)
{
  const long long int N = workspace.N;
  const double L = workspace.L;
  const double m = workspace.m;
  const double kappa = workspace.kappa;
  const double inv_h_sqr = 1.0 / ((L / N) * (L / N));

  dim3 threadsPerBlock((int)N, 1);
  dim3 numBlocks((int)N, (int)N);
  kappa_equation_kernel<<numBlocks, threadsPerBlock>>(thrust::raw_pointer_cast(x.data()),
    thrust::raw_pointer_cast(dxdt.data()), m, kappa, inv_h_sqr, N);
}

```

Here `kappa_equation_kernel` is the CUDA kernel, and the `__global__` specifier means this function runs on the GPU. `KappaEquation::operator()` invokes the kernel via `kappa_equation_↵kernel<<numBlocks, threadsPerBlock>>>`. Given the execution configuration `threadsPer↵Block` and `numBlocks`, the function `kappa_equation_kernel` is executed once for each a, b, c , with $0 \leq a, b, c < N$. Depending on the kernel, different execution configurations could result in varying performance, or even introduce bugs. See the [CUDA C programming guide](#) for more details.

Chapter 3

Concept Index

3.1 Concepts

Here is a list of all documented concepts with brief descriptions:

HasFa	17
HasFRWParameters	17
HasLambda	17
HasLatticeParams	17
HasMass	18
HasPsiApproximationParameters	18
LatticeEquationConcept	18

Chapter 4

Hierarchical Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

boost::numeric::odeint::algebra_stepper_base	
midpoint< State, Value, Deriv, Time, Algebra, Operations, Resizer >	35
ComovingCurvatureEquationInFRW	19
ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density >	20
CudaApproximateComovingCurvatureEquationInFRW	22
CudaComovingCurvatureEquationInFRW	23
CudaFixedCurvatureEquationInFRW	24
CudaKleinGordonEquationInFRW	24
CudaLambdaEquationInFRW	25
CudaSqrtPotentialEquationInFRW	25
cufftWrapper	26
cufftWrapperBatchedD2Z	27
cufftWrapperD2Z	27
cufftWrapperNoBatching	27
empty	28
fftWrapperDispatcher< Vector >	29
fftWrapperDispatcher< Eigen::VectorXd >	29
fftWrapperDispatcher< thrust::device_vector< double > >	29
fftwWrapper	30
KGParam	32
KleinGordonEquation	33
KleinGordonEquationInFRW	34
MyParam	36
SampleParam	37
StaticEOSCosmology	38
WKBSolutionForKleinGordonEquationInFRW	40
WorkspaceGeneric< Vector >	41

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

ComovingCurvatureEquationInFRW	
Equation for free scalar field in FRW spacetime, including comoving metric perturbations (in radiation domination)	19
ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density >	
An "observer" used to save spectra and slices during the simulation at roughly constant time intervals	20
CudaApproximateComovingCurvatureEquationInFRW	22
CudaComovingCurvatureEquationInFRW	23
CudaFixedCurvatureEquationInFRW	24
CudaKleinGordonEquationInFRW	24
CudaLambdaEquationInFRW	25
CudaSqrtPotentialEquationInFRW	25
cufftWrapper	
Wrapper for various cufft functions for a N^3 grid. Similar to fftwWrapper	26
cufftWrapperBatchedD2Z	27
cufftWrapperD2Z	27
cufftWrapperNoBatching	
Wrapper for various cufft functions for a N^3 grid. Similar to fftwWrapper	27
empty	
An empty placeholder object	28
fftWrapperDispatcher< Vector >	
Dispatcher for fftWrapper* types	29
fftWrapperDispatcher< Eigen::VectorXd >	29
fftWrapperDispatcher< thrust::device_vector< double > >	29
fftwWrapper	
Wrapper for various FFTW functions for a N^3 grid	30
KGParam	32
KleinGordonEquation	
The Klein Gordon equation, $\ddot{\varphi} - \nabla^2 \varphi + m^2 \varphi = 0$	33
KleinGordonEquationInFRW	
The Klein Gordon in FRW equation, $\ddot{\varphi} + 3H\dot{\varphi} - \nabla^2 \varphi/a^2 + m^2 \varphi = 0$	34
midpoint< State, Value, Deriv, Time, Algebra, Operations, Resizer >	35
MyParam	36
SampleParam	
A sample parameter type specifying a lambda-phi-4 theory in an FRW background	37

StaticEOSCosmology	
A convenience class used to calculate FRW related quantities for constant EOS spacetimes . .	38
WKBSolutionForKleinGordonEquationInFRW	40
WorkspaceGeneric< Vector >	
A generic workspace for storing temporary	41

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

src/cuda_wrapper.cuh	
Wrapper for CUDA Toolkit	43
src/dispatcher.hpp	
Automatically dispatching between using FFTW and CUFFT libraries	45
src/eigen_wrapper.hpp	
Wrap some Eigen functionalities	46
src/equations.hpp	
Header for field equations that runs on the CPU	47
src/equations_cuda.cuh	
Header for field equations that runs on the GPU	49
src/fdm3d.hpp	
Common procedures for manipulating / summarizing field configuration on a 3D lattice	51
src/fdm3d_cuda.cuh	
CUDA implementation for fdm3d.hpp . Common procedures for manipulating / summarizing field configuration on a 3D lattice	56
src/fftw_wrapper.hpp	
Wrapper for FFTW library	58
src/field_booster.hpp	
src/initializer.hpp	
Snippets for initializing workspaces. (e.g. field initial conditions, etc)	59
src/io.hpp	
Input/output utilities	67
src/midpoint.hpp	
Midpoint method implementation for odeint stepper	68
src/observer.hpp	
Implements "observers", which controls what gets saved during simulations	70
src/param.hpp	
Utilities for managing simulations parameters	72
src/physics.hpp	
Collection of repeatedly used physics formulas. (e.g. FRW cosmology related formulas)	74
src/random_field.hpp	
Utilities for generating Gaussian random fields of given spectrum and inhomogeneity	75
src/special_function.hpp	
src/utility.hpp	
Utilities for debugging / profiling / pretty printing	79

src/ wkb.hpp	
Implementation of the WKB solution	81
src/ workspace.hpp	
A generic "workspace" class, containing parameters / data / tools used during simulations . . .	82

Chapter 7

Concept Documentation

7.1 HasFa Concept Reference

7.1.1 Concept definition

```
template<typename Param>
concept HasFa = requires (Param param) { TYPE_REQUIREMENT(param.f_a, double) }
```

7.2 HasFRWParameters Concept Reference

7.2.1 Concept definition

```
template<typename Param>
concept HasFRWParameters = requires (Param param)
{ TYPE_REQUIREMENT(param.a1, double)

  TYPE_REQUIREMENT(param.t1, double) }
```

7.3 HasLambda Concept Reference

7.3.1 Concept definition

```
template<typename Param>
concept HasLambda = requires (Param param) { TYPE_REQUIREMENT(param.lambda, double) }
```

7.4 HasLatticeParams Concept Reference

7.4.1 Concept definition

```
template<typename Param>
concept HasLatticeParams = requires (Param param)
{ TYPE_REQUIREMENT(param.N, long long int)
  TYPE_REQUIREMENT(param.L, double) }
```

7.5 HasMass Concept Reference

7.5.1 Concept definition

```
template<typename Param>
concept HasMass = requires (Param param) { TYPE_REQUIREMENT(param.m, double) }
```

7.6 HasPsiApproximationParameters Concept Reference

7.6.1 Concept definition

```
template<typename Param>
concept HasPsiApproximationParameters = requires (Param param)
{ TYPE_REQUIREMENT(param.M, long long int) }
```

7.7 LatticeEquationConcept Concept Reference

7.7.1 Concept definition

```
template<typename Equation>
concept LatticeEquationConcept = requires (Equation eqn)
{
    eqn.workspace;
    eqn.compute_energy_density(eqn.workspace, 0.0);
}
```

Chapter 8

Class Documentation

8.1 ComovingCurvatureEquationInFRW Struct Reference

Equation for free scalar field in FRW spacetime, including comoving metric perturbations (in radiation domination).

```
#include <equations.hpp>
```

Public Types

- `typedef Eigen::VectorXd` **Vector**
- `typedef Vector` **State**
- `typedef WorkspaceGeneric< State >` **Workspace**

Public Member Functions

- **ComovingCurvatureEquationInFRW** (`Workspace` &`workspace_`)
- `void operator()` (`const` `State` &, `State` &, `const double`)

Static Public Member Functions

- `static Vector` `compute_energy_density` (`Workspace` &`workspace`, `const double t`)
Compute the energy density profile from the workspace.

Public Attributes

- `Workspace` & `workspace`

8.1.1 Detailed Description

Equation for free scalar field in FRW spacetime, including comoving metric perturbations (in radiation domination).

Equation is given by

$$\ddot{\varphi} + 3H\dot{\varphi} - e^{4\Psi} \frac{\nabla^2}{a^2} \varphi + e^{2\Psi} m^2 \varphi - 4\dot{\Psi}\dot{\varphi} = 0$$

$$\Psi_{\mathbf{k}}(t) = 2\mathcal{R}_{\mathbf{k}} \frac{\sin(k\eta/\sqrt{3}) - (k\eta/\sqrt{3}) \cos(k\eta/\sqrt{3})}{(k\eta/\sqrt{3})^3}$$

$$\dot{\Psi}_{\mathbf{k}}(t) = 2\mathcal{R}_{\mathbf{k}} H(t) \frac{3(k\eta/\sqrt{3}) \cos(k\eta/\sqrt{3}) + ((k\eta/\sqrt{3})^2 - 3) \sin(k\eta/\sqrt{3})}{(k\eta/\sqrt{3})^3}$$

$$\eta(t) = \frac{(2H_i t)^{1/2}}{a_i H_i} \quad \text{is the conformal time}$$

where $\mathcal{R}_{\mathbf{k}}$ is read from workspace variable `R_fft`. See equation (6.160) of Baumann's cosmology textbook. This implementation is not optimized. It was only used for verifying the GPU implementations [CudaComovingCurvatureEquationInFRW](#) and [CudaApproximateComovingCurvatureEquationInFRW](#).

8.1.2 Member Function Documentation

8.1.2.1 compute_energy_density()

```
ComovingCurvatureEquationInFRW::Vector ComovingCurvatureEquationInFRW::compute_energy_density
(
    Workspace & workspace,
    const double t ) [static]
```

Compute the energy density profile from the workspace.

Parameters

<code>in</code>	<code>workspace</code>	The workspace for evaluating the energy density.
	<code>t</code>	The current time parameter.

Returns

A vector of size N^3 , giving the energy density profile $\rho = \frac{1}{2}(e^{-2\Psi}\dot{\varphi}^2 + e^{2\Psi}(\nabla\varphi)^2/a(t)^2 + m^2\varphi^2)$ on the lattice.

The documentation for this struct was generated from the following files:

- [src/equations.hpp](#)
- [src/equations.cpp](#)

8.2 ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density > Struct Template Reference

An "observer" used to save spectra and slices during the simulation at roughly constant time intervals.

```
#include <observer.hpp>
```

Public Types

- `typedef` Equation::Workspace **Workspace**
- `typedef` Workspace::State **State**
- `typedef` State **Vector**

Public Member Functions

- `template<typename Param >`
ConstIntervalObserver (`const` std::string &dir_, `const` Param ¶m, Equation &eqn)
- **ConstIntervalObserver** (`const` ConstIntervalObserver &)=default
- `void operator()` (`const` State &x, double t)

Public Attributes

- Workspace & **workspace**
- `int` **idx**
- std::string **dir**
- `double` **t_start**
- `double` **t_end**
- `double` **t_interval**
- `double` **t_last**

8.2.1 Detailed Description

`template<typename Equation, bool save_field_spectrum = true, bool save_density_spectrum = true, bool save_density = false>`
`struct ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density >`

An "observer" used to save spectra and slices during the simulation at roughly constant time intervals.

Template Parameters

<i>Equation</i>	This type parameter is necessary for selecting the right compute_energy_density function. (Each system has its own way to compute energy density.)
<i>save_field_spectrum</i>	Control whether field spectrum should be saved. The output is $ \varphi_{\mathbf{k}} ^2 + \dot{\varphi}_{\mathbf{k}} ^2/\omega_k^2$ summed over directions. Saves to file <code>dir/varphi_plus_spectrum_(idx).dat</code> , where (idx) is the index of save. See compute_mode_power_spectrum for more details.
<i>save_density_spectrum</i>	Control whether energy density spectrum should be saved. The output is $ \rho_{\mathbf{k}} ^2$ summed over directions. Saves to file <code>dir/rho_spectrum_(idx).dat</code> , where (idx) is the index of save. See compute_power_spectrum for more details.
<i>save_density</i>	Control whether field spectrum should be saved. The output is a constant-z slice of density spectrum and the density spectrum averaged over the z axis. Saves to files <code>dir/rho_slice_(idx).dat</code> and <code>dir/rho_axis_average_(idx).dat</code> , where (idx) is the index of save.

Saves spectra and slices to `dir` during the simulation at roughly constant time intervals. During the simulation, the `operator()` function is called at each time step. We don't want to save a snapshot at every time step, so use `t_interval` to control when to save data.

The template parameters can be used to choose what to save. By default, the observer saves field and density spectra, but not the slices.

8.2.2 Member Data Documentation

8.2.2.1 t_end

```
template<typename Equation , bool save_field_spectrum = true, bool save_density_spectrum =
true, bool save_density = false>
double ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density
>::t_end
```

End time of simulation.

8.2.2.2 t_interval

```
template<typename Equation , bool save_field_spectrum = true, bool save_density_spectrum =
true, bool save_density = false>
double ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density
>::t_interval
```

Save to file in dir every t_interval.

8.2.2.3 t_start

```
template<typename Equation , bool save_field_spectrum = true, bool save_density_spectrum =
true, bool save_density = false>
double ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density
>::t_start
```

Start time of simulation.

The documentation for this struct was generated from the following file:

- src/observer.hpp

8.3 CudaApproximateComovingCurvatureEquationInFRW Struct Reference

Public Types

- `typedef thrust::device_vector< double > Vector`
- `typedef Vector State`
- `typedef WorkspaceGeneric< Vector > Workspace`

Public Member Functions

- **CudaApproximateComovingCurvatureEquationInFRW** ([Workspace](#) &[workspace_](#))
- **void operator()** ([const](#) State &, State &, [const double](#))

Static Public Member Functions

- [static](#) Vector **compute_energy_density** ([Workspace](#) &workspace, [const double](#) t)

Public Attributes

- [Workspace](#) & **workspace**

The documentation for this struct was generated from the following files:

- [src/equations_cuda.cuh](#)
- [src/equations_cuda.cu](#)

8.4 CudaComovingCurvatureEquationInFRW Struct Reference

Public Types

- [typedef](#) thrust::device_vector< [double](#) > **Vector**
- [typedef](#) Vector **State**
- [typedef](#) [WorkspaceGeneric](#)< Vector > **Workspace**

Public Member Functions

- **CudaComovingCurvatureEquationInFRW** ([Workspace](#) &[workspace_](#))
- **void operator()** ([const](#) State &, State &, [const double](#))

Static Public Member Functions

- [static](#) Vector **compute_energy_density** ([Workspace](#) &workspace, [const double](#) t)

Public Attributes

- [Workspace](#) & **workspace**

The documentation for this struct was generated from the following files:

- [src/equations_cuda.cuh](#)
- [src/equations_cuda.cu](#)

8.5 CudaFixedCurvatureEquationInFRW Struct Reference

Public Types

- `typedef thrust::device_vector< double > Vector`
- `typedef Vector State`
- `typedef WorkspaceGeneric< Vector > Workspace`

Public Member Functions

- `CudaFixedCurvatureEquationInFRW (Workspace &workspace_)`
- `void operator() (const State &, State &, const double)`

Static Public Member Functions

- `static Vector compute_energy_density (const Workspace &workspace, const double t)`

Public Attributes

- `Workspace & workspace`

The documentation for this struct was generated from the following files:

- `src/equations_cuda.cuh`
- `src/equations_cuda.cu`

8.6 CudaKleinGordonEquationInFRW Struct Reference

Public Types

- `typedef thrust::device_vector< double > Vector`
- `typedef Vector State`
- `typedef WorkspaceGeneric< Vector > Workspace`

Public Member Functions

- `CudaKleinGordonEquationInFRW (Workspace &workspace_)`
- `void operator() (const State &, State &, const double)`

Static Public Member Functions

- `static Vector compute_energy_density (const Workspace &workspace, const double t)`
- `static Vector compute_dot_energy_density (const Workspace &workspace, const double t)`

Public Attributes

- [Workspace](#) & **workspace**

The documentation for this struct was generated from the following files:

- [src/equations_cuda.cuh](#)
- [src/equations_cuda.cu](#)

8.7 CudaLambdaEquationInFRW Struct Reference

Public Types

- `typedef thrust::device_vector< double > Vector`
- `typedef Vector State`
- `typedef WorkspaceGeneric< Vector > Workspace`

Public Member Functions

- **CudaLambdaEquationInFRW** ([Workspace](#) &[workspace_](#))
- **void operator()** (`const` State &, State &, `const double` t)

Static Public Member Functions

- `static` Vector **compute_energy_density** (`const` [Workspace](#) &[workspace](#), `const double` t)

Public Attributes

- [Workspace](#) & **workspace**

The documentation for this struct was generated from the following files:

- [src/equations_cuda.cuh](#)
- [src/equations_cuda.cu](#)

8.8 CudaSqrtPotentialEquationInFRW Struct Reference

Public Types

- `typedef thrust::device_vector< double > Vector`
- `typedef Vector State`
- `typedef WorkspaceGeneric< Vector > Workspace`

Public Member Functions

- **CudaSqrtPotentialEquationInFRW** ([Workspace](#) &workspace_)
- **void operator()** ([const](#) State &, State &, [const double](#))

Static Public Member Functions

- [static](#) Vector **compute_energy_density** ([const Workspace](#) &workspace, [const double](#) t)

Public Attributes

- [Workspace](#) & workspace

The documentation for this struct was generated from the following files:

- [src/equations_cuda.cuh](#)
- [src/equations_cuda.cu](#)

8.9 cufftWrapper Struct Reference

Wrapper for various cufft functions for a N^3 grid. Similar to [fftwWrapper](#).

Public Member Functions

- **cufftWrapper** ([int](#) N_)
- thrust::device_vector< [double](#) > **execute_d2z** (thrust::device_vector< [double](#) > &in)
- thrust::device_vector< [double](#) > **execute_batched_d2z** (thrust::device_vector< [double](#) > &in)
- thrust::device_vector< [double](#) > **execute_z2d** (thrust::device_vector< [double](#) > &in)
- **cufftWrapper** ([const cufftWrapper](#) &)=delete
- [cufftWrapper](#) & **operator=** ([const cufftWrapper](#) &)=delete
- **cufftWrapper** ([cufftWrapper](#) &&)=delete
- [cufftWrapper](#) & **operator=** ([cufftWrapper](#) &&)=delete

Public Attributes

- [int](#) N
- [cufftHandle](#) plan_d2z
- [cufftHandle](#) plan_batched_d2z
- [cufftHandle](#) plan_z2d
- thrust::device_vector< [double](#) > work_area

8.9.1 Detailed Description

Wrapper for various cufft functions for a N^3 grid. Similar to [fftwWrapper](#).

See <https://docs.nvidia.com/cuda/cufft/index.html>.

The documentation for this struct was generated from the following files:

- [src/cuda_wrapper.cuh](#)
- [src/cuda_wrapper.cu](#)

8.10 cufftWrapperBatchedD2Z Struct Reference

Public Member Functions

- `cufftWrapperBatchedD2Z` (`int N_`)
- `thrust::device_vector< double > execute` (`thrust::device_vector< double > &in`)
- `cufftWrapperBatchedD2Z` (`const cufftWrapperBatchedD2Z &)=delete`
- `cufftWrapperBatchedD2Z & operator=` (`const cufftWrapperBatchedD2Z &)=delete`
- `cufftWrapperBatchedD2Z` (`cufftWrapperBatchedD2Z &&)=delete`
- `cufftWrapperBatchedD2Z & operator=` (`cufftWrapperBatchedD2Z &&)=delete`

Public Attributes

- `int N`
- `cufftHandle plan`

The documentation for this struct was generated from the following files:

- `src/cuda_wrapper.cuh`
- `src/cuda_wrapper.cu`

8.11 cufftWrapperD2Z Struct Reference

Public Member Functions

- `cufftWrapperD2Z` (`int N_`)
- `thrust::device_vector< double > execute` (`thrust::device_vector< double > &in`)
- `cufftWrapperD2Z` (`const cufftWrapperD2Z &)=delete`
- `cufftWrapperD2Z & operator=` (`const cufftWrapperD2Z &)=delete`
- `cufftWrapperD2Z` (`cufftWrapperD2Z &&)=delete`
- `cufftWrapperD2Z & operator=` (`cufftWrapperD2Z &&)=delete`

Public Attributes

- `int N`
- `cufftHandle plan`

The documentation for this struct was generated from the following files:

- `src/cuda_wrapper.cuh`
- `src/cuda_wrapper.cu`

8.12 cufftWrapperNoBatching Struct Reference

Wrapper for various cufft functions for a N^3 grid. Similar to `fftwWrapper`.

Public Member Functions

- **cufftWrapperNoBatching** (**int** N_)
- **thrust::device_vector**< **double** > **execute_d2z** (**thrust::device_vector**< **double** > &in)
- **thrust::device_vector**< **double** > **execute_batched_d2z** (**thrust::device_vector**< **double** > &in)
- **thrust::device_vector**< **double** > **execute_z2d** (**thrust::device_vector**< **double** > &in)
- **void execute_inplace_z2d** (**thrust::device_vector**< **double** > &inout)
- **cufftWrapperNoBatching** (**const cufftWrapperNoBatching** &)=delete
- **cufftWrapperNoBatching** & **operator=** (**const cufftWrapperNoBatching** &)=delete
- **cufftWrapperNoBatching** (**cufftWrapperNoBatching** &&)=delete
- **cufftWrapperNoBatching** & **operator=** (**cufftWrapperNoBatching** &&)=delete

Public Attributes

- **int** N
- **cufftHandle** **plan_d2z**
- **cufftHandle** **plan_z2d**
- **thrust::device_vector**< **double** > **work_area**

8.12.1 Detailed Description

Wrapper for various cufft functions for a N^3 grid. Similar to [fftwWrapper](#).

Uses less GPU memory than [cufftWrapper](#). See <https://docs.nvidia.com/cuda/cufft/index.html>.

The documentation for this struct was generated from the following files:

- [src/cuda_wrapper.cuh](#)
- [src/cuda_wrapper.cu](#)

8.13 empty Struct Reference

An empty placeholder object.

```
#include <dispatcher.hpp>
```

8.13.1 Detailed Description

An empty placeholder object.

The documentation for this struct was generated from the following file:

- [src/dispatcher.hpp](#)

8.14 fftWrapperDispatcher< Vector > Struct Template Reference

Dispatcher for fftWrapper* types.

```
#include <dispatcher.hpp>
```

Public Types

- [typedef empty](#) **D2Z**
- [typedef empty](#) **BatchedD2Z**
- [typedef empty](#) **Generic**

8.14.1 Detailed Description

```
template<typename Vector>  
struct fftWrapperDispatcher< Vector >
```

Dispatcher for fftWrapper* types.

The documentation for this struct was generated from the following file:

- [src/dispatcher.hpp](#)

8.15 fftWrapperDispatcher< Eigen::VectorXd > Struct Reference

Public Types

- [typedef empty](#) **D2Z**
- [typedef empty](#) **BatchedD2Z**
- [typedef fftwWrapper](#) **Generic**

The documentation for this struct was generated from the following file:

- [src/dispatcher.hpp](#)

8.16 fftWrapperDispatcher< thrust::device_vector< double > > Struct Reference

Public Types

- [typedef cufftWrapperD2Z](#) **D2Z**
- [typedef cufftWrapperBatchedD2Z](#) **BatchedD2Z**
- [typedef cufftWrapperNoBatching](#) **Generic**

The documentation for this struct was generated from the following file:

- [src/dispatcher.hpp](#)

8.17 fftwWrapper Struct Reference

Wrapper for various FFTW functions for a N^3 grid.

```
#include <fftw_wrapper.hpp>
```

Public Member Functions

- [fftwWrapper](#) (int N_*)
- Eigen::VectorXd [execute_d2z](#) (Eigen::VectorXd & in)
(Double floating point) Real to complex transform.
- Eigen::VectorXd [execute_batched_d2z](#) (Eigen::VectorXd & in)
(Double floating point) Real to complex transform.
- Eigen::VectorXd [execute_z2d](#) (Eigen::VectorXd & in)
(Double floating point) Complex to real transform.
- void [execute_z2d](#) (Eigen::VectorXd & in , Eigen::VectorXd & out)
No-return version of the complex to real transform.
- void [execute_inplace_z2d](#) (Eigen::VectorXd & $inout$)
In-place version of the complex to real transform.
- [fftwWrapper](#) (const [fftwWrapper](#) &)=delete
- [fftwWrapper](#) & [operator=](#) (const [fftwWrapper](#) &)=delete
- [fftwWrapper](#) ([fftwWrapper](#) &&)=delete
- [fftwWrapper](#) & [operator=](#) ([fftwWrapper](#) &&)=delete

Public Attributes

- int N
- [fftw_plan](#) [plan_d2z](#)
- [fftw_plan](#) [plan_z2d](#)
- [fftw_plan](#) [plan_inplace_z2d](#)

8.17.1 Detailed Description

Wrapper for various FFTW functions for a N^3 grid.

See https://www.fftw.org/fftw3_doc/Multi_002dDimensional-DFTs-of-Real-Data.html for details.

8.17.2 Constructor & Destructor Documentation

8.17.2.1 fftwWrapper()

```
fftwWrapper::fftwWrapper (
    int  $N_*$  ) [explicit]
```

Constructor for grid size N .

8.17.3 Member Function Documentation

8.17.3.1 execute_batched_d2z()

```
Eigen::VectorXd fftwWrapper::execute_batched_d2z (
    Eigen::VectorXd &  $in$  )
```

(Double floating point) Real to complex transform.

Parameters

<i>in</i>	A real vector of size $2N^3$.
-----------	--------------------------------

Returns

A real vector of size $4N^2(N/2 + 1)$ (or a complex vector of size $2N^2(N/2 + 1)$), containing discrete Fourier transforms of input. The first $2N^2(N/2 + 1)$ entries of the output are the DFT of the first N^3 entries of the input, and similar for the rest.

8.17.3.2 execute_d2z()

```
Eigen::VectorXd fftwWrapper::execute_d2z (
    Eigen::VectorXd & in )
```

(Double floating point) Real to complex transform.

Parameters

<i>in</i>	A real vector of size N^3 .
-----------	-------------------------------

Returns

A real vector of size $2N^2(N/2 + 1)$ (or a complex vector of size $N^2(N/2 + 1)$), containing the discrete Fourier transform of input.

8.17.3.3 execute_inplace_z2d()

```
void fftwWrapper::execute_inplace_z2d (
    Eigen::VectorXd & inout )
```

In-place version of the complex to real transform.

Parameters

<i>inout</i>	A real vector of size $2N^2(N/2 + 1)$ (or a complex vector of size $N^2(N/2 + 1)$). After the function call the data in <i>inout</i> is changed its inverse DFT. The vector still has size $2N^2(N/2 + 1)$, but only N^3 of the entries are meaningful. The entries are in FFTW padded format.
--------------	--

Note

Make sure to access the elements inside with `PADDED_IDX_OF` (instead of `IDX_OF`). See https://www.fftw.org/fftw3_doc/Multi_002dDimensional-DFTs-of-Real-Data.html for details of the padded format.

8.17.3.4 execute_z2d() [1/2]

```
Eigen::VectorXd fftwWrapper::execute_z2d (
    Eigen::VectorXd & in )
```

(Double floating point) Complex to real transform.

Parameters

<i>in</i>	A real vector of size $2N^2(N/2 + 1)$ (or a complex vector of size $N^2(N/2 + 1)$).
-----------	--

Returns

A real vector of size N^3 , containing the inverse discrete Fourier transform of input.

Note

This function destroys the information in the input *in*. See FFTW's documentation https://www.ffmpeg.org/fftw3_doc/Planner-Flags.html.

8.17.3.5 execute_z2d() [2/2]

```
void fftwWrapper::execute_z2d (
    Eigen::VectorXd & in,
    Eigen::VectorXd & out )
```

No-return version of the complex to real transform.

Note

This version is useful if you want to reuse the same memory location for the output; doing this can reduce unnecessary memory allocation / deallocation, saving lots of time. Like the other version, this function destroys the data in input *in*.

The documentation for this struct was generated from the following files:

- [src/fftw_wrapper.hpp](#)
- [src/fftw_wrapper.cpp](#)

8.18 KGParam Struct Reference

Public Attributes

- [long long int N](#)
- [double L](#)
- [double m](#)

The documentation for this struct was generated from the following file:

- [src/field_booster.cpp](#)

8.19 KleinGordonEquation Struct Reference

The Klein Gordon equation, $\ddot{\varphi} - \nabla^2 \varphi + m^2 \varphi = 0$.

```
#include <equations.hpp>
```

Public Types

- `typedef Eigen::VectorXd` **Vector**
- `typedef Vector` **State**
- `typedef WorkspaceGeneric< State >` **Workspace**

Public Member Functions

- **KleinGordonEquation** (`Workspace` &workspace_)
- `void operator()` (`const State` &, `State` &, `const double`)

The function called by odeint library.

Static Public Member Functions

- `static Vector` `compute_energy_density` (`const Workspace` &workspace, `const double` t)

Compute the energy density profile from the workspace.

Public Attributes

- `Workspace` & workspace

8.19.1 Detailed Description

The Klein Gordon equation, $\ddot{\varphi} - \nabla^2 \varphi + m^2 \varphi = 0$.

8.19.2 Member Function Documentation

8.19.2.1 compute_energy_density()

```
KleinGordonEquation::Vector KleinGordonEquation::compute_energy_density (
    const Workspace & workspace,
    const double t ) [static]
```

Compute the energy density profile from the workspace.

Parameters

in	<i>workspace</i>	The workspace for evaluating the energy density.
	<i>t</i>	The current time parameter.

Returns

A vector of size N^3 , giving the energy density profile $\rho = \frac{1}{2}(\dot{\varphi}^2 + (\nabla\varphi)^2 + m^2\varphi^2)$ on the lattice.

8.19.2.2 operator()

```
void KleinGordonEquation::operator() (
    const State & x,
    State & dxdt,
    const double t )
```

The function called by odeint library.

Parameters

in	x	The current state of the system.
out	$dxdt$	The time derivative, dxdt of the system.
	t	The current time parameter.

The documentation for this struct was generated from the following files:

- [src/equations.hpp](#)
- [src/equations.cpp](#)

8.20 KleinGordonEquationInFRW Struct Reference

The Klein Gordon in FRW equation, $\ddot{\varphi} + 3H\dot{\varphi} - \nabla^2\varphi/a^2 + m^2\varphi = 0$.

```
#include <equations.hpp>
```

Public Types

- `typedef Eigen::VectorXd` **Vector**
- `typedef` Vector **State**
- `typedef WorkspaceGeneric< State >` **Workspace**

Public Member Functions

- **KleinGordonEquationInFRW** ([Workspace](#) &workspace_)
- `void operator()` (`const` State &, State &, `const double`)

Static Public Member Functions

- `static` Vector `compute_energy_density` (`const` [Workspace](#) &workspace, `const double` t)
Compute the energy density profile from the workspace.

Public Attributes

- [Workspace](#) & `workspace`

8.20.1 Detailed Description

The Klein Gordon in FRW equation, $\ddot{\varphi} + 3H\dot{\varphi} - \nabla^2\varphi/a^2 + m^2\varphi = 0$.

8.20.2 Member Function Documentation

8.20.2.1 `compute_energy_density()`

```
KleinGordonEquationInFRW::Vector KleinGordonEquationInFRW::compute_energy_density (
    const Workspace & workspace,
    const double t ) [static]
```

Compute the energy density profile from the workspace.

Parameters

in	<i>workspace</i>	The workspace for evaluating the energy density.
	<i>t</i>	The current time parameter.

Returns

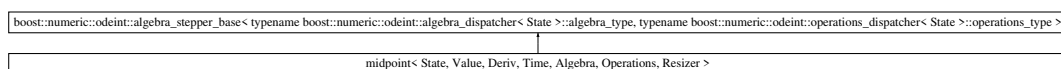
A vector of size N^3 , giving the energy density profile $\rho = \frac{1}{2}(\dot{\varphi}^2 + (\nabla\varphi)^2/a(t)^2 + m^2\varphi^2)$ given on the N^3 on the lattice.

The documentation for this struct was generated from the following files:

- [src/equations.hpp](#)
- [src/equations.cpp](#)

8.21 midpoint< State, Value, Deriv, Time, Algebra, Operations, Resizer > Class Template Reference

Inheritance diagram for midpoint< State, Value, Deriv, Time, Algebra, Operations, Resizer >:



Public Types

- `typedef State state_type`
- `typedef State deriv_type`
- `typedef Value value_type`
- `typedef Time time_type`
- `typedef unsigned short order_type`
- `typedef boost::numeric::odeint::stepper_tag stepper_category`
- `typedef boost::numeric::odeint::algebra_stepper_base< Algebra, Operations > algebra_stepper_base_type`
- `typedef algebra_stepper_base_type::algebra_type algebra_type`
- `typedef algebra_stepper_base_type::operations_type operations_type`

Public Member Functions

- `template<class System >`
`void do_step (System system, State &in, Time t, Time dt)`
- `bool resize_impl (const State &x)`

Static Public Member Functions

- `static order_type order (void)`

The documentation for this class was generated from the following file:

- [src/midpoint.hpp](#)

8.22 MyParam Struct Reference

Public Attributes

- `long long int N`
- `double L`
- `double m`
- `double lambda`
- `double f_a`
- `double k_ast`
- `double k_Psi`
- `double varphi_std_dev`
- `double Psi_std_dev`
- `double a1`
- `double H1`
- `double t1`
- `double t_start`
- `double t_end`
- `double t_interval`
- `double delta_t`
- `long long int M`
- `double f`
- `double delta_varphi_std_dev`
- `double k_delta_varphi`

The documentation for this struct was generated from the following file:

- [src/main.cpp](#)

8.23 SampleParam Struct Reference

A sample parameter type specifying a lambda-phi-4 theory in an FRW background.

```
#include <param.hpp>
```

Public Attributes

- `long long int N`
- `double L`
- `double m`
- `double lambda`
- `double k_ast`
- `double varphi_std_dev`
- `double a1`
- `double H1`
- `double t1`

8.23.1 Detailed Description

A sample parameter type specifying a lambda-phi-4 theory in an FRW background.

8.23.2 Member Data Documentation

8.23.2.1 a1

```
double SampleParam::a1
```

the scale factor at time t_1

8.23.2.2 H1

```
double SampleParam::H1
```

the Hubble parameter at time t_1

8.23.2.3 k_ast

```
double SampleParam::k_ast
```

the wavenumber k_* for the peak of the field power spectrum

8.23.2.4 L

```
double SampleParam::L
```

the length of one side of the box (i.e. $L = 10.0$ means the box has volume L^3)

8.23.2.5 lambda

```
double SampleParam::lambda
```

quartic self-interaction of the scalar field (i.e. λ in $V(\varphi) = \frac{1}{2}m^2\varphi^2 + \frac{1}{4}\lambda\varphi^4$)

8.23.2.6 m

```
double SampleParam::m
```

mass m of the scalar field

8.23.2.7 N

```
long long int SampleParam::N
```

the number of lattice points on one side of the box (i.e. $N = 256$ means 256^3 lattice sites)

8.23.2.8 t1

```
double SampleParam::t1
```

coordinate time parameter t_1 (For radiation domination, $a(t) = a_1(1 + 2H_1(t - t_1))^{1/2}$, $H(t) = H_1(1 + 2H_1(t - t_1))^{-1}$.)

8.23.2.9 varphi_std_dev

```
double SampleParam::varphi_std_dev
```

the expected RMS value $\langle\varphi^2\rangle$ for the field, averaged over the box

The documentation for this struct was generated from the following file:

- [src/param.hpp](#)

8.24 StaticEOSCosmology Struct Reference

A convenience class used to calculate FRW related quantities for constant EOS spacetimes.

```
#include <physics.hpp>
```

Public Member Functions

- **StaticEOSCosmology** (const double a1_, const double H1_, const double t1_, const double p_)
 template<typename T >
StaticEOSCosmology (const T ¶m)
This constructor assumes radiation domination.
- **StaticEOSCosmology** (void)
The default constructor gives Minkowski spacetime.
- double **a** (const double t) const
Returns the scale factor at coordinate time t .
- double **H** (const double t) const
Returns the Hubble parameter at coordinate time t .
- double **eta** (const double t) const
Returns the conformal time η at coordinate time t .

Public Attributes

- double a1
- double H1
- double t1
- double p

8.24.1 Detailed Description

A convenience class used to calculate FRW related quantities for constant EOS spacetimes.

8.24.2 Member Function Documentation

8.24.2.1 eta()

```
double StaticEOSCosmology::eta (
    const double t ) const [inline]
```

Returns the conformal time η at coordinate time t .

Converts coordinate time t to conformal time η . The conversion assumes the convention $a = a_1(\eta/\eta_1)^p$, where $\eta_1 = p/(a_1 H_1)$. In this convention, η_1 and t_1 are at the same physical time. The conversion formula is

$$\eta = \frac{p}{a_1 H_1} (1 + (1 + 1/p) H_1 (t - t_1))^{1/(1+p)}$$

.

8.24.3 Member Data Documentation

8.24.3.1 a1

```
double StaticEOSCosmology::a1
```

The scale factor at t_1 .

8.24.3.2 H1

`double StaticEOSCosmology::H1`

The Hubble parameter at t_1 .

8.24.3.3 p

`double StaticEOSCosmology::p`

Power-law between scale factor and conformal time, $a \propto \eta^p$. In terms of EOS w , we have $p = \frac{2}{1+3w}$.

8.24.3.4 t1

`double StaticEOSCosmology::t1`

A pivot coordinate time t_1 .

The documentation for this struct was generated from the following file:

- [src/physics.hpp](#)

8.25 WKBSolutionForKleinGordonEquationInFRW Struct Reference

Public Types

- `typedef Eigen::VectorXd` **Vector**
- `typedef` Vector **State**
- `typedef WorkspaceGeneric< State >` **Workspace**

Public Member Functions

- **WKBSolutionForKleinGordonEquationInFRW** ([Workspace](#) &[workspace_](#), `const double t_i_`)
- Vector **evaluate_at** (`const double t`)

Public Attributes

- [Workspace](#) & **workspace**
- `double` **t_i**
- Vector **phi_ffts**

The documentation for this struct was generated from the following files:

- [src/wkb.hpp](#)
- [src/wkb.cpp](#)

8.26 WorkspaceGeneric< Vector > Struct Template Reference

A generic workspace for storing temporary.

```
#include <workspace.hpp>
```

Public Types

- `typedef` Vector **State**

Public Member Functions

- `template<HasLatticeParams Param>`
WorkspaceGeneric (`const Param ¶m`, `auto &initializer`)

Public Attributes

- `long long int` **N**
- `double` **L**
- `double` **m**
- `StaticEOSCosmology` **cosmology** {}
- `State` **state**
- `double` **lambda** {0}
- `double` **f_a** {1.0}
- `Vector` **Psi**
- `Vector` **dPsidt**
- `Vector` **Psi_fft**
- `Vector` **dPsidt_fft**
- `Vector` **R_fft**
- `std::vector< double >` **t_list**
- `fftWrapperDispatcher< Vector >::Generic` **fft_wrapper**
- `bool` **Psi_approximation_initialized** {false}
- `long long int` **M**
- `std::unique_ptr< typename fftWrapperDispatcher< Vector >::Generic >` **fft_wrapper_M_ptr**
- `Vector` **cutoff_R_fft**

8.26.1 Detailed Description

```
template<typename Vector>
struct WorkspaceGeneric< Vector >
```

A generic workspace for storing temporary.

[WorkspaceGeneric](#) contains everything used during simulations, including the field state, gravitational potential, parameters, etc. It is initialized by a Param struct (containing just a few numbers) and an "initializer" (see [initializer.hpp](#)).

The lifetime of objects in the workspace are managed by us (this codebase), instead of external libraries (e.g. odeint).

The documentation for this struct was generated from the following file:

- `src/workspace.hpp`

Chapter 9

File Documentation

9.1 src/cuda_wrapper.cuh File Reference

Wrapper for CUDA Toolkit.

```
#include <iostream>
#include <Eigen/Dense>
#include <thrust/device_vector.h>
#include "cufft.h"
#include "cufftXt.h"
#include <cuda_runtime.h>
```

Classes

- struct [cufftWrapperD2Z](#)
- struct [cufftWrapperBatchedD2Z](#)
- struct [cufftWrapper](#)
Wrapper for various cufft functions for a N^3 grid. Similar to [fftwWrapper](#).
- struct [cufftWrapperNoBatching](#)
Wrapper for various cufft functions for a N^3 grid. Similar to [fftwWrapper](#).

Typedefs

- typedef decltype(Eigen::VectorXd().begin()) [eigen_iterator](#)
- typedef decltype(thrust::device_vector< double >().begin()) [thrust_iterator](#)
- typedef thrust::detail::normal_iterator< thrust::device_ptr< const double > > [thrust_const_iterator](#)
- typedef Eigen::internal::pointer_based_stl_iterator< Eigen::Matrix< double, -1, 1 > > [eigen_iterator_2](#)

Functions

- void [copy_vector](#) (Eigen::VectorXd &out, const thrust::device_vector< double > &in)
- void [show_gpu_memory_usage](#) (void)

9.1.1 Detailed Description

Wrapper for CUDA Toolkit.

Author

Siyang Ling

9.2 cuda_wrapper.cuh

[Go to the documentation of this file.](#)

```

00001
00006 #ifndef CUDA_WRAPPER_CUH
00007 #define CUDA_WRAPPER_CUH
00008
00009 #include <iostream>
00010
00011 #include <Eigen/Dense>
00012
00013 #include <thrust/device_vector.h>
00014 // #include <thrust/host_vector.h>
00015 // #include <thrust/execution_policy.h>
00016 // #include <thrust/reduce.h>
00017 // #include <thrust/functional.h>
00018 // #include <thrust/fill.h>
00019 // #include <thrust/transform.h>
00020
00021 #include "cufft.h"
00022 #include "cufftXt.h"
00023 #include <cuda_runtime.h>
00024
00025
00026
00027 typedef decltype(Eigen::VectorXd().begin()) eigen_iterator;
00028 typedef decltype(thrust::device_vector<double>().begin()) thrust_iterator;
00029 typedef thrust::detail::normal_iterator<thrust::device_ptr<const double> > thrust_const_iterator;
00030 typedef Eigen::internal::pointer_based_stl_iterator<Eigen::Matrix<double, -1, 1> > eigen_iterator_2;
00031
00032
00033 /*
00034  * Explicit template instantiation declarations for the thrust library.
00035  * They are declared here so that they are instantiated in cuda_wrapper.cu (and compiled with nvcc),
00036  * and don't get instantiated in other translation units.
00037  * This is necessary since we want to call thrust functions in translation units compiled by other
00038  * compilers (g++ / icpx).
00039  */
00039 extern template class thrust::device_vector<double>;
00040 extern template class thrust::device_ptr<double>;
00041 extern template thrust::device_ptr<double> thrust::for_each_n(const
thrust::detail::execution_policy_base<thrust::cuda_cub::tag> &, thrust::device_ptr<double>, unsigned
long, thrust::detail::device_generate_functor<thrust::detail::fill_functor<double> >);
00042 extern template eigen_iterator thrust::copy(const
thrust::detail::execution_policy_base<thrust::cuda_cub::cross_system<thrust::cuda_cub::tag,
thrust::system::cpp::detail::tag> &, thrust_const_iterator, thrust_const_iterator, eigen_iterator);
00043
00044 extern template thrust_iterator thrust::copy(eigen_iterator, eigen_iterator, thrust_iterator);
00045 extern template eigen_iterator thrust::copy(thrust_iterator, thrust_iterator, eigen_iterator);
00046
00047 //Eigen::VectorXd copy_vector(const thrust::device_vector<double> &in);
00048 void copy_vector(Eigen::VectorXd &out, const thrust::device_vector<double> &in);
00049 //void copy_vector(Eigen::VectorXd &out, const Eigen::VectorXd &in);
00050
00051
00052 void show_gpu_memory_usage(void);
00053
00054 // Wrapper for 3D cufftPlan3d. Performs double to complex double FFT for a \f$ N^3 \f$ grid.
00055 struct cufftWrapperD2Z {
00056     int N;
00057     cufftHandle plan;
00058     explicit cufftWrapperD2Z(int N_);
00059     ~cufftWrapperD2Z();
00060     thrust::device_vector<double> execute(thrust::device_vector<double> &in);
00061
00062     cufftWrapperD2Z(const cufftWrapperD2Z &) = delete;
00063     cufftWrapperD2Z &operator=(const cufftWrapperD2Z &) = delete;
00064     cufftWrapperD2Z(cufftWrapperD2Z &&) = delete;
00065     cufftWrapperD2Z &operator=(cufftWrapperD2Z &&) = delete;

```

```

00066 };
00067
00068
00069 // Wrapper for 3D cufftPlanMany. Performs two double to complex double FFT for a \f$ N^3 \f$ grid.
00070 struct cufftWrapperBatchedD2Z {
00071     int N;
00072     cufftHandle plan;
00073     explicit cufftWrapperBatchedD2Z(int N_);
00074     ~cufftWrapperBatchedD2Z();
00075     thrust::device_vector<double> execute(thrust::device_vector<double> &in);
00076
00077     cufftWrapperBatchedD2Z(const cufftWrapperBatchedD2Z &) = delete;
00078     cufftWrapperBatchedD2Z &operator=(const cufftWrapperBatchedD2Z &) = delete;
00079     cufftWrapperBatchedD2Z(cufftWrapperBatchedD2Z &&) = delete;
00080     cufftWrapperBatchedD2Z &operator=(cufftWrapperBatchedD2Z &&) = delete;
00081 };
00082
00083 struct cufftWrapper {
00084     int N;
00085     cufftHandle plan_d2z;
00086     cufftHandle plan_batched_d2z;
00087     cufftHandle plan_z2d;
00088     thrust::device_vector<double> work_area;
00089     explicit cufftWrapper(int N_);
00090     ~cufftWrapper();
00091
00092     thrust::device_vector<double> execute_d2z(thrust::device_vector<double> &in);
00093     thrust::device_vector<double> execute_batched_d2z(thrust::device_vector<double> &in);
00094     thrust::device_vector<double> execute_z2d(thrust::device_vector<double> &in);
00095
00096     cufftWrapper(const cufftWrapper &) = delete;
00097     cufftWrapper &operator=(const cufftWrapper &) = delete;
00098     cufftWrapper(cufftWrapper &&) = delete;
00099     cufftWrapper &operator=(cufftWrapper &&) = delete;
00100 };
00101
00102 struct cufftWrapperNoBatching {
00103     int N;
00104     cufftHandle plan_d2z;
00105     cufftHandle plan_z2d;
00106     thrust::device_vector<double> work_area;
00107     explicit cufftWrapperNoBatching(int N_);
00108     ~cufftWrapperNoBatching();
00109
00110     thrust::device_vector<double> execute_d2z(thrust::device_vector<double> &in);
00111     thrust::device_vector<double> execute_batched_d2z(thrust::device_vector<double> &in);
00112     thrust::device_vector<double> execute_z2d(thrust::device_vector<double> &in);
00113     void execute_inplace_z2d(thrust::device_vector<double> &inout);
00114
00115     cufftWrapperNoBatching(const cufftWrapperNoBatching &) = delete;
00116     cufftWrapperNoBatching &operator=(const cufftWrapperNoBatching &) = delete;
00117     cufftWrapperNoBatching(cufftWrapperNoBatching &&) = delete;
00118     cufftWrapperNoBatching &operator=(cufftWrapperNoBatching &&) = delete;
00119 };
00120
00121
00122
00123
00124
00125
00126
00127
00128
00129
00130 };
00131
00132
00133 #endif

```

9.3 src/dispatcher.hpp File Reference

Automatically dispatching between using FFTW and CUFFT libraries.

```

#include "fftw_wrapper.hpp"
#include <thrust/device_vector.h>
#include "cuda_wrapper.cuh"

```

Classes

- struct [empty](#)
An empty placeholder object.
- struct [fftwWrapperDispatcher< Vector >](#)
Dispatcher for fftWrapper types.*
- struct [fftwWrapperDispatcher< thrust::device_vector< double > >](#)
- struct [fftwWrapperDispatcher< Eigen::VectorXd >](#)

Macros

- `#define ALGORITHM_NAMESPACE thrust`

9.3.1 Detailed Description

Automatically dispatching between using FFTW and CUFFT libraries.

Author

Siyang Ling

9.4 dispatcher.hpp

[Go to the documentation of this file.](#)

```
00001
00006 #ifndef DISPATCHER_HPP
00007 #define DISPATCHER_HPP
00008
00009 #include "fftw_wrapper.hpp"
00010
00011 #ifndef DISABLE_CUDA
00012 #include <thrust/device_vector.h>
00013 #include "cuda_wrapper.cuh"
00014 #define ALGORITHM_NAMESPACE thrust
00015 #else
00016 #define ALGORITHM_NAMESPACE std
00017 #endif
00018
00019
00021 struct empty {};
00022
00024 template<typename Vector>
00025 struct fftWrapperDispatcher {
00026     typedef empty D2Z;
00027     typedef empty BatchedD2Z;
00028     typedef empty Generic;
00029 };
00030
00031 #ifndef DISABLE_CUDA
00032 template<>
00033 struct fftWrapperDispatcher<thrust::device_vector<double>> {
00034     typedef cufftWrapperD2Z D2Z;
00035     typedef cufftWrapperBatchedD2Z BatchedD2Z;
00036     //typedef cufftWrapper Generic;
00037     typedef cufftWrapperNoBatching Generic;
00038 };
00039 #endif
00040
00041 template<>
00042 struct fftWrapperDispatcher<Eigen::VectorXd> {
00043     typedef empty D2Z;
00044     typedef empty BatchedD2Z;
00045     typedef fftwWrapper Generic;
00046 };
00047
00048
00049 #endif
```

9.5 src/eigen_wrapper.hpp File Reference

Wrap some Eigen functionalites.

```
#include <Eigen/Dense>
```

Functions

- `void copy_vector` ([Eigen::VectorXd](#) &out, const [Eigen::VectorXd](#) &in)

9.5.1 Detailed Description

Wrap some Eigen functionalities.

Author

Siyang Ling

9.6 eigen_wrapper.hpp

[Go to the documentation of this file.](#)

```
00001
00006 #ifndef EIGEN_WRAPPER_HPP
00007 #define EIGEN_WRAPPER_HPP
00008
00009 #include <Eigen/Dense>
00010
00011 void copy_vector(Eigen::VectorXd &out, const Eigen::VectorXd &in);
00012
00013 #endif
```

9.7 src/equations.hpp File Reference

Header for field equations that runs on the CPU.

```
#include "Eigen/Dense"
#include <boost/numeric/odeint.hpp>
#include <boost/numeric/odeint/external/eigen/eigen.hpp>
#include "odeint_eigen/eigen_operations.hpp"
#include "workspace.hpp"
```

Classes

- struct [KleinGordonEquation](#)
The Klein Gordon equation, $\ddot{\varphi} - \nabla^2 \varphi + m^2 \varphi = 0$.
- struct [KleinGordonEquationInFRW](#)
The Klein Gordon in FRW equation, $\ddot{\varphi} + 3H\dot{\varphi} - \nabla^2 \varphi/a^2 + m^2 \varphi = 0$.
- struct [ComovingCurvatureEquationInFRW](#)
Equation for free scalar field in FRW spacetime, including comoving metric perturbations (in radiation domination).

Concepts

- concept [LatticeEquationConcept](#)

9.7.1 Detailed Description

Header for field equations that runs on the CPU.

Author

Siyang Ling

This is the header for field equations that are supposed to run on CPU. Equations declared here will be used by the odeint library via `operator()`. See https://www.boost.org/doc/libs/1_85_0/libs/numeric/odeint/doc/html/boost_numeric_odeint/getting_started/short_example.html for an example of odeint equation. Typically, `compute_energy_density` is also implemented for saving energy density spectrum.

9.8 equations.hpp

[Go to the documentation of this file.](#)

```

00001
00011 #ifndef EQUATIONS_HPP
00012 #define EQUATIONS_HPP
00013
00014
00015 #include "Eigen/Dense"
00016
00017 #include <boost/numeric/odeint.hpp>
00018 #include <boost/numeric/odeint/external/eigen/eigen.hpp>
00019
00020 #include "odeint_eigen/eigen_operations.hpp"
00021
00022 #include "workspace.hpp"
00023
00024 template<typename Equation>
00025 concept LatticeEquationConcept = requires (Equation eqn)
00026 {
00027     //typename Equation::State;
00028     eqn.workspace;
00029     eqn.compute_energy_density(eqn.workspace, 0.0);
00030 };
00031
00032
00036 struct KleinGordonEquation {
00037     typedef Eigen::VectorXd Vector;
00038     typedef Vector State;
00039     typedef WorkspaceGeneric<State> Workspace;
00040     Workspace &workspace;
00041
00042     KleinGordonEquation(Workspace &workspace_) : workspace(workspace_) {}
00043
00050     void operator()(const State &, State &, const double);
00051
00058     static Vector compute_energy_density(const Workspace &workspace, const double t);
00059 };
00060
00061
00065 struct KleinGordonEquationInFRW {
00066     typedef Eigen::VectorXd Vector;
00067     typedef Vector State;
00068     typedef WorkspaceGeneric<State> Workspace;
00069     Workspace &workspace;
00070
00071     KleinGordonEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00072
00073     void operator()(const State &, State &, const double);
00074
00081     static Vector compute_energy_density(const Workspace &workspace, const double t);
00082 };
00083
00084
00100 struct ComovingCurvatureEquationInFRW {
00101     typedef Eigen::VectorXd Vector;
00102     typedef Vector State;
00103     typedef WorkspaceGeneric<State> Workspace;
00104     Workspace &workspace;

```



```

00105
00106     ComovingCurvatureEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00107
00108     void operator()(const State &, State &, const double);
00109
00110     static Vector compute_energy_density(Workspace &workspace, const double t);
00111 };
00112
00113
00114
00115
00116
00117
00118
00119
00120
00121
00122 #endif

```

9.9 src/equations_cuda.cuh File Reference

Header for field equations that runs on the GPU.

```

#include "equations.hpp"
#include <thrust/device_vector.h>
#include "odeint_thrust/thrust.hpp"

```

Classes

- struct [CudaKleinGordonEquationInFRW](#)
- struct [CudaLambdaEquationInFRW](#)
- struct [CudaSqrtPotentialEquationInFRW](#)
- struct [CudaFixedCurvatureEquationInFRW](#)
- struct [CudaComovingCurvatureEquationInFRW](#)
- struct [CudaApproximateComovingCurvatureEquationInFRW](#)

9.9.1 Detailed Description

Header for field equations that runs on the GPU.

Author

Siyang Ling

This is the header for field equations that are supposed to run on GPU (via CUDA). Equations declared here will be used by the odeint library via `operator()`. See https://www.boost.org/doc/libs/1_85_0/libs/numeric/odeint/doc/html/boost_numeric_odeint/getting_started/short_example.html for an example of odeint equation. Typically, `compute_energy_density` is also implemented for saving energy density spectrum. Also see [equations.hpp](#).

9.10 equations_cuda.cuh

[Go to the documentation of this file.](#)

```

00001
00012 #ifndef EQUATIONS_CUDA_CUH
00013 #define EQUATIONS_CUDA_CUH
00014
00015 #include "equations.hpp"
00016
00017 #include <thrust/device_vector.h>
00018
00019 #include "odeint_thrust/thrust.hpp"
00020
00021 struct CudaKleinGordonEquationInFRW {
00022     typedef thrust::device_vector<double> Vector;
00023     typedef Vector State;
00024     typedef WorkspaceGeneric<Vector> Workspace;
00025     Workspace &workspace;
00026
00027     CudaKleinGordonEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00028
00029     void operator()(const State &, State &, const double);
00030
00031     static Vector compute_energy_density(const Workspace &workspace, const double t);
00032     static Vector compute_dot_energy_density(const Workspace &workspace, const double t);
00033 };
00034
00035
00036 struct CudaLambdaEquationInFRW {
00037     typedef thrust::device_vector<double> Vector;
00038     typedef Vector State;
00039     typedef WorkspaceGeneric<Vector> Workspace;
00040     Workspace &workspace;
00041
00042     CudaLambdaEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00043
00044     void operator()(const State &, State &, const double);
00045
00046     static Vector compute_energy_density(const Workspace &workspace, const double t);
00047 };
00048
00049
00050 struct CudaSqrtPotentialEquationInFRW {
00051     typedef thrust::device_vector<double> Vector;
00052     typedef Vector State;
00053     typedef WorkspaceGeneric<Vector> Workspace;
00054     Workspace &workspace;
00055
00056     CudaSqrtPotentialEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00057
00058     void operator()(const State &, State &, const double);
00059
00060     static Vector compute_energy_density(const Workspace &workspace, const double t);
00061 };
00062
00063
00064 struct CudaFixedCurvatureEquationInFRW {
00065     typedef thrust::device_vector<double> Vector;
00066     typedef Vector State;
00067     typedef WorkspaceGeneric<Vector> Workspace;
00068     Workspace &workspace;
00069
00070     CudaFixedCurvatureEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00071
00072     void operator()(const State &, State &, const double);
00073
00074     static Vector compute_energy_density(const Workspace &workspace, const double t);
00075 };
00076
00077
00078 struct CudaComovingCurvatureEquationInFRW {
00079     typedef thrust::device_vector<double> Vector;
00080     typedef Vector State;
00081     typedef WorkspaceGeneric<Vector> Workspace;
00082     Workspace &workspace;
00083
00084     CudaComovingCurvatureEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00085
00086     void operator()(const State &, State &, const double);
00087
00088     static Vector compute_energy_density(Workspace &workspace, const double t);
00089 };
00090
00091
00092 struct CudaApproximateComovingCurvatureEquationInFRW {

```

```

00093  typedef thrust::device_vector<double> Vector;
00094  typedef Vector State;
00095  typedef WorkspaceGeneric<Vector> Workspace;
00096  Workspace &workspace;
00097
00098  CudaApproximateComovingCurvatureEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00099
00100  void operator()(const State &, State &, const double);
00101
00102  static Vector compute_energy_density(Workspace &workspace, const double t);
00103 };
00104
00105
00106 // Explicit template instantiation declaration for the thrust library.
00107 extern template double thrust::reduce(const
thrust::detail::execution_policy_base<thrust::cuda_cub::tag> &, thrust::const_iterator,
thrust::const_iterator, double, boost::numeric::odeint::detail::maximum<double>);
00108
00109 // Deprecated function for testing CUDA kernels.
00110 /*
00111 void compute_deriv_test(const Eigen::VectorXd &in, Eigen::VectorXd &out,
00112                        const double m, const double lambda,
00113                        const double a_t, const double H_t, const double inv_ah_sqr,
00114                        const long long int N);
00115 */
00116 /*
00117 void kernel_test(const thrust::device_vector<double> &R_fft, thrust::device_vector<double> &Psi,
thrust::device_vector<double> &dPsidt,
00118                const long long int N, const double L, const double m,
00119                const double a_t, const double H_t, const double eta_t, const double inv_ah_sqr,
00120                const double t, fftWrapperDispatcher<thrust::device_vector<double>>::Generic &fft_wrapper);
00121 */
00122 #endif

```

9.11 src/fdm3d.hpp File Reference

Common procedures for manipulating / summarizing field configuration on a 3D lattice.

```

#include "Eigen/Dense"
#include "dispatcher.hpp"

```

Macros

- **#define IDX_OF(N, i, j, k)** $((N)*(N)*(i) + (N)*(j) + (k))$
Give the index of a lattice point, assuming row major ordering in (i,j,k).
- **#define PADDED_IDX_OF(N, i, j, k)** $((N)*2*((N)/2+1)*(i) + 2*((N)/2+1)*(j) + (k))$
Give the index of a lattice point, assuming that the array is in FFTW padded format.

Functions

- [Eigen::VectorXd compute_power_spectrum](#) (const long long int N, Eigen::VectorXd &f, fftWrapperDispatcher< Eigen::VectorXd >::Generic &fft_wrapper)
Sum Fourier mode power of a field over directions.
- [Eigen::VectorXd compute_mode_power_spectrum](#) (const long long int N, const double L, const double m, const double a_t, Eigen::VectorXd &state, fftWrapperDispatcher< Eigen::VectorXd >::Generic &fft_wrapper)
Sum Fourier mode power of a field over directions, along with the power in time derivatives.
- [Eigen::VectorXd compute_inverse_laplacian](#) (const long long int N, const double L, Eigen::VectorXd &f, fftWrapperDispatcher< Eigen::VectorXd >::Generic &fft_wrapper)
Compute the inverse Laplacian of a field. AKA solve the Poisson equation.
- [Eigen::VectorXd compute_field_with_scaled_fourier_modes](#) (const long long int N, const double L, Eigen::VectorXd &f, std::function< double(const double)> kernel, fftWrapperDispatcher< Eigen::VectorXd >::Generic &fft_wrapper)
Scale each Fourier mode of a field by a kernel, returning the new field.
- [Eigen::VectorXd compute_cutoff_fouriars](#) (const long long int N, const long long int M, Eigen::VectorXd &fft)
Downsample a Fourier transform on a N^3 grid so that it looks like a Fourier transform on a M^3 grid.

9.11.1 Detailed Description

Common procedures for manipulating / summarizing field configuration on a 3D lattice.

Author

Siyang Ling

9.11.2 Macro Definition Documentation

9.11.2.1 PADDED_IDX_OF

```
#define PADDED_IDX_OF (
    N,
    i,
    j,
    k ) ( (N)*2*((N)/2+1)*(i) + 2*((N)/2+1)*(j) + (k) )
```

Give the index of a lattice point, assuming that the array is in FFTW padded format.

See https://www.fftw.org/fftw3_doc/Multi_002dDimensional-DFTs-of-Real-Data.html for details of the format.

9.11.3 Function Documentation

9.11.3.1 compute_cutoff_fouriers()

```
Eigen::VectorXd compute_cutoff_fouriers (
    const long long int N,
    const long long int M,
    Eigen::VectorXd & fft )
```

Downsample a Fourier transform on a N^3 grid so that it looks like a Fourier transform on a M^3 grid.

Parameters

N	Number of lattice points (of full grid).
M	Number of lattice points (of downsampled grid).
fft	The DFT of a real field. Should be a vector of size $2N^2(N/2 + 1)$.

Returns

The downsampled DFT the input DFT. Should be a vector of size $2M^2(M/2 + 1)$.

9.11.3.2 compute_field_with_scaled_fourier_modes()

```
Eigen::VectorXd compute_field_with_scaled_fourier_modes (
    const long long int N,
```

```

const double L,
Eigen::VectorXd & f,
std::function< double(const double)> kernel,
fftWrapperDispatcher< Eigen::VectorXd >::Generic & fft_wrapper )

```

Scale each Fourier mode of a field by a kernel, returning the new field.

Parameters

N	Number of lattice points.
L	Box size.
f	The field on a 3D lattice. Should be a vector of size N^3 with row major ordering. See IDX_OF.
$kernel$	A function K determining how the Fourier modes are scaled.
$fft_wrapper$	A fftWrapper initialized to do Fourier transforms on grid size N .

Returns

The field with $f_{\mathbf{k}} \mapsto K(k)f_{\mathbf{k}}$, where K is given by kernel.

9.11.3.3 compute_inverse_laplacian()

```

Eigen::VectorXd compute_inverse_laplacian (
    const long long int N,
    const double L,
    Eigen::VectorXd & f,
    fftWrapperDispatcher< Eigen::VectorXd >::Generic & fft_wrapper )

```

Compute the inverse Laplacian of a field. AKA solve the Poisson equation.

Parameters

N	Number of lattice points.
L	Box size.
f	The field on a 3D lattice. Should be a vector of size N^3 with row major ordering. See IDX_OF.
$fft_wrapper$	A fftWrapper initialized to do Fourier transforms on grid size N .

Returns

The solution to the Poisson equation with RHS f , namely $\nabla^{-2}f$. The output have zero homogeneous mode regardless of whether f has one.

9.11.3.4 compute_mode_power_spectrum()

```

Eigen::VectorXd compute_mode_power_spectrum (
    const long long int N,
    const double L,
    const double m,
    const double a_t,

```

```
Eigen::VectorXd & state,  
fftWrapperDispatcher< Eigen::VectorXd >::Generic & fft_wrapper )
```

Sum Fourier mode power of a field over directions, along with the power in time derivatives.

Parameters

N	Number of lattice points.
L	Box size.
m	Mass m of (free) scalar field.
a_t	Current scale factor.
$state$	The state $(\varphi, \dot{\varphi})$ of a scalar field on a 3D lattice. Should be a vector of size $2N^3$, with the first half (first N^3 indices) containing φ , and the second half containing $\dot{\varphi}$.
$fft_wrapper$	A <code>fftWrapper</code> initialized to do Fourier transforms on grid size N .

Returns

A vector of size $3(N/2)^2 + 1$, with its s index containing the power in Fourier modes with wavenumber $\sqrt{s}k_{\text{IR}}$. Specifically:

$$\begin{aligned} \text{output}[s] &= \sum_{i^2+j^2+k^2=s} |\tilde{\varphi}_{i,j,k}|^2 + \frac{|\dot{\tilde{\varphi}}_{i,j,k}|^2}{\omega_k^2} \\ \omega_k^2 &= m^2 + sk_{\text{IR}}^2/a^2(t) \end{aligned}$$

Here, $\tilde{\varphi}_{a,b,c}$ and $\dot{\tilde{\varphi}}_{a,b,c}$ are the DFT's, (i, j, k) labels a site on the reciprocal lattice, and $-N/2 + 1 \leq i, j, k \leq N/2$; see <https://garrettgoon.com/gaussian-fields/> for details on this convention. Also see `compute_power_spectrum`.

9.11.3.5 `compute_power_spectrum()`

```
Eigen::VectorXd compute_power_spectrum (
    const long long int N,
    Eigen::VectorXd & f,
    fftWrapperDispatcher< Eigen::VectorXd >::Generic & fft_wrapper )
```

Sum Fourier mode power of a field over directions.

Parameters

N	Number of lattice points.
f	The field on a 3D lattice. Should be a vector of size N^3 with row major ordering. See <code>IDX_OF</code> .
$fft_wrapper$	A <code>fftWrapper</code> initialized to do Fourier transforms on grid size N .

Returns

A vector of size $3(N/2)^2 + 1$, with its s index containing the power in Fourier modes with wavenumber $\sqrt{s}k_{\text{IR}}$. Specifically:

$$\begin{aligned} \text{output}[s] &= \sum_{i^2+j^2+k^2=s} |\tilde{f}_{i,j,k}|^2 \\ \tilde{f}_{i,j,k} &= \sum_{a,b,c} e^{-2\pi i(a,b,c) \cdot (i,j,k)/N} f_{a,b,c} \end{aligned}$$

Here, \tilde{f} is the DFT of f , (i, j, k) labels a site on the reciprocal lattice, and $-N/2 + 1 \leq i, j, k \leq N/2$. See <https://garrettgoon.com/gaussian-fields/> for details on this convention.

9.12 fdm3d.hpp

[Go to the documentation of this file.](#)

```
00001
00006 #ifndef FDM3D_HPP
00007 #define FDM3D_HPP
00008
00009 #include "Eigen/Dense"
00010 #include "dispatcher.hpp"
00011
00012
00016 #define IDX_OF(N, i, j, k) ((N)*(N)*(i) + (N)*(j) + (k))
00017
00018
00024 #define PADDED_IDX_OF(N, i, j, k) ((N)*2*((N)/2+1)*(i) + 2*((N)/2+1)*(j) + (k))
00025
00026
00041 Eigen::VectorXd compute_power_spectrum(const long long int N,
00042                                     Eigen::VectorXd &f,
00043                                     fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00044
00065 Eigen::VectorXd compute_mode_power_spectrum(const long long int N, const double L, const double m,
00066                                     const double a_t,
00067                                     Eigen::VectorXd &state,
00068                                     fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00069 // Eigen::VectorXd compute_mode_power_spectrum(const long long int N, const double L, const double m,
00070 // Eigen::VectorXd &state,
00071 // fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00072
00082 Eigen::VectorXd compute_inverse_laplacian(const long long int N, const double L,
00083                                     Eigen::VectorXd &f,
00084                                     fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00085
00086
00096 Eigen::VectorXd compute_field_with_scaled_fourier_modes(const long long int N, const double L,
00097                                     Eigen::VectorXd &f,
00098                                     std::function<double(const double)> kernel,
00099                                     fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00100
00101
00109 Eigen::VectorXd compute_cutoff_fouriers(const long long int N, const long long int M,
00110                                     Eigen::VectorXd &fft);
00111
00112
00113 // Deprecated
00114 // Eigen::VectorXd compute_power_spectrum(const long long int N, Eigen::VectorXd &phi);
00115 // Eigen::VectorXd compute_fourier(const long long int N, const double L, Eigen::VectorXd &phi);
00116 // Eigen::VectorXd compute_laplacian(const long long int N, const double L, const Eigen::VectorXd &f);
00117
00118
00119
00120 #endif
```

9.13 src/fdm3d_cuda.cuh File Reference

CUDA implementation for [fdm3d.hpp](#). Common procedures for manipulating / summarizing field configuration on a 3D lattice.

```
#include <thrust/device_vector.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include "cuda_wrapper.cuh"
#include "dispatcher.hpp"
#include "fdm3d.hpp"
```

Functions

- [thrust::device_vector< double >](#) [compute_mode_power_spectrum](#) (const long long int N, const double L, const double m, const double a_t, [thrust::device_vector< double >](#) &state, [fftWrapperDispatcher< thrust::device_vector< double > >::Generic &fft_wrapper](#))

CUDA version of identically named function in [fdm3d.hpp](#).

- `thrust::device_vector< double > compute_power_spectrum (const long long int N, thrust::device_vector< double > &f, fftWrapperDispatcher< thrust::device_vector< double > >::Generic &fft_wrapper)`

CUDA version of identically named function in [fdm3d.hpp](#).

- `thrust::device_vector< double > compute_laplacian (const long long int N, const double L, thrust::device_vector< double > &f)`
- `thrust::device_vector< double > compute_inverse_laplacian (const long long int N, const double L, thrust::device_vector< double > &f, fftWrapperDispatcher< thrust::device_vector< double > >::Generic &fft_wrapper)`

CUDA version of identically named function in [fdm3d.hpp](#).

- `thrust::device_vector< double > compute_cutoff_fouriers (const long long int N, const long long int M, const thrust::device_vector< double > &fft)`

CUDA version of identically named function in [fdm3d.hpp](#).

9.13.1 Detailed Description

CUDA implementation for [fdm3d.hpp](#). Common procedures for manipulating / summarizing field configuration on a 3D lattice.

Author

Siyang Ling

9.14 fdm3d_cuda.cuh

[Go to the documentation of this file.](#)

```
00001
00006 #ifndef FDM3D_CUDA_CUH
00007 #define FDM3D_CUDA_CUH
00008
00009 #include <thrust/device_vector.h>
00010 #include <thrust/reduce.h>
00011 #include <thrust/functional.h>
00012
00013 // #include "odeint_thrust/thrust.hpp"
00014 #include "cuda_wrapper.cuh"
00015 #include "dispatcher.hpp"
00016
00017 #include "fdm3d.hpp"
00018
00022 thrust::device_vector<double> compute_mode_power_spectrum(const long long int N, const double L, const
double m, const double a_t,
00023                                     thrust::device_vector<double> &state,
00024                                     fftWrapperDispatcher<thrust::device_vector<double>::Generic
&fft_wrapper);
00025
00029 thrust::device_vector<double> compute_power_spectrum(const long long int N,
00030 thrust::device_vector<double> &f,
00031                                     fftWrapperDispatcher<thrust::device_vector<double>::Generic
&fft_wrapper);
00032
00033 thrust::device_vector<double> compute_laplacian(const long long int N, const double L,
00034 thrust::device_vector<double> &f);
00035
00039 thrust::device_vector<double> compute_inverse_laplacian(const long long int N, const double L,
00040 thrust::device_vector<double> &f,
00041                                     fftWrapperDispatcher<thrust::device_vector<double>::Generic &fft_wrapper);
00042
00046 thrust::device_vector<double> compute_cutoff_fouriers(const long long int N, const long long int M,
00047 const thrust::device_vector<double> &fft);
00048
00049 // void compute_inverse_laplacian_test(const long long int N, const double L,
00050 // thrust::device_vector<double> &fft);
00051 #endif
```

9.15 src/fftw_wrapper.hpp File Reference

Wrapper for FFTW library.

```
#include <iostream>
#include <Eigen/Dense>
#include <fftw3.h>
```

Classes

- struct [fftwWrapper](#)
Wrapper for various FFTW functions for a N^3 grid.

9.15.1 Detailed Description

Wrapper for FFTW library.

Author

Siyang Ling

9.16 fftw_wrapper.hpp

[Go to the documentation of this file.](#)

```
00001
00006 #ifndef FFTW_WRAPPER_HPP
00007 #define FFTW_WRAPPER_HPP
00008
00009 #include <iostream>
00010
00011 #include <Eigen/Dense>
00012 #include <fftw3.h>
00013
00014
00020 struct fftwWrapper {
00021     int N;
00022     fftw_plan plan_d2z;
00023     fftw_plan plan_z2d;
00024     fftw_plan plan_inplace_z2d;
00025     explicit fftwWrapper(int N_);
00026     ~fftwWrapper();
00027
00033     Eigen::VectorXd execute_d2z(Eigen::VectorXd &in);
00034
00040     Eigen::VectorXd execute_batched_d2z(Eigen::VectorXd &in);
00041
00049     Eigen::VectorXd execute_z2d(Eigen::VectorXd &in);
00050
00056     void execute_z2d(Eigen::VectorXd &in, Eigen::VectorXd &out);
00057
00064     void execute_inplace_z2d(Eigen::VectorXd &inout);
00065
00066     fftwWrapper(const fftwWrapper &) = delete;
00067     fftwWrapper &operator=(const fftwWrapper &) = delete;
00068     fftwWrapper(fftwWrapper &&) = delete;
00069     fftwWrapper &operator=(fftwWrapper &&) = delete;
00070 };
00071
00072
00073 #endif
```

9.17 field_booster.hpp

```

00001 /*
00002     Tools related to boosting (adding velocity to) fields.
00003 */
00004 #ifndef FIELD_BOOSTER_HPP
00005 #define FIELD_BOOSTER_HPP
00006
00007 #include "Eigen/Dense"
00008
00009 void add_phase_to_state(Eigen::VectorXd &state, const Eigen::VectorXd &phase);
00010
00011 void boost_klein_gordon_field(Eigen::VectorXd &varphi, Eigen::VectorXd &dt_varphi, const
    Eigen::VectorXd &theta,
00012                               const long long int N, const double L, const double m);
00013
00014
00015 #endif

```

9.18 src/initializer.hpp File Reference

Snippets for initializing workspaces. (e.g. field initial conditions, etc)

```

#include "fdm3d.hpp"
#include "random_field.hpp"
#include "fftw_wrapper.hpp"
#include "special_function.hpp"
#include <thrust/device_vector.h>
#include "cuda_wrapper.cuh"

```

Macros

- `#define ALGORITHM_NAMESPACE thrust`

Variables

- `auto unperturbed_grf`
Initialize a field and its derivative from a white noise power spectrum with cutoff k_{ast} .
- `auto unperturbed_grf_with_background`
Initialize a field and its derivative from a white noise power spectrum with cutoff k_{ast} , plus homogeneous background.
- `auto perturbed_grf`
Setup a scalar field with inhomogeneous Gaussian random initial conditions.
- `auto perturbed_grf_without_saving_Psi`
Same as `perturbed_grf`, but does not store Ψ in workspace to save memory.
- `auto unperturbed_grf_with_Psi`
Same as `unperturbed_grf`, but with an extra scale-invariant Ψ .
- `auto unperturbed_grf_and_fixed_curvature`
Initialize a homogeneous Gaussian random field and some scale invariant curvature perturbation.
- `auto perturbed_grf_and_comoving_curvature_fft`
Initialize an inhomogeneous Gaussian random field and the fft of some scale invariant comoving curvature perturbation.
- `auto homogeneous_field`
Initialize a homogeneous field with amplitude f and time derivative dt_f . For testing the numerical code.
- `auto homogeneous_field_with_fluctuations`
Initialize a homogeneous field with amplitude f , plus scale-invariant perturbations (resembling quantum fluctuations).
- `auto plane_wave`
Plane wave initial condition. For testing the numerical code.
- `auto wave_packet`
Wave packet initial condition.

9.18.1 Detailed Description

Snippets for initializing workspaces. (e.g. field initial conditions, etc)

Author

Siyang Ling

Contains a collection of lambda functions used to initialize the workspace for simulation. Each lambda function `initializer` is meant to be passed to a workspace constructor `Workspace(param, initializer)`. The use of lambda's makes it easy to switch between `param` types and `workspace` types. Moreover, in order for easy switching between CPU code and GPU code, the initializers compute everything on CPU first, and then decide whether the results should be copied to CPU or GPU memory.

9.18.2 Variable Documentation

9.18.2.1 homogeneous_field

```
auto homogeneous_field [inline]
```

Initial value:

=

```
[](const auto param, auto &workspace) {
    const long long int N = param.N;
    Eigen::VectorXd varphi = Eigen::VectorXd::Constant(N*N*N, param.f);
    Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, param.dt_f);

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());
    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
}
```

Initialize a homogeneous field with amplitude `f` and time derivative `dt_f`. For testing the numerical code.

9.18.2.2 homogeneous_field_with_fluctuations

```
auto homogeneous_field_with_fluctuations [inline]
```

Initial value:

=

```
[](const auto param, auto &workspace) {
    const long long int N = param.N;
    Eigen::VectorXd varphi = Eigen::VectorXd::Constant(N*N*N, param.f);
    Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, 0.0);

    Spectrum P_delta_varphi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L,
        param.delta_varphi_std_dev, param.k_delta_varphi, -3);
    Eigen::VectorXd delta_varphi = generate_gaussian_random_field(param.N, param.L, P_delta_varphi);
    varphi += delta_varphi;

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());
    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
}
```

Initialize a homogeneous field with amplitude `f`, plus scale-invariant perturbations (resembling quantum fluctuations).

9.18.2.3 perturbed_grf

```
auto perturbed_grf [inline]
```

Initial value:

```
=
[] (const auto param, auto &workspace) {
    Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
        param.k_Psi, -3);
    Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
        param.k_ast, 0);
    Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
    Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
    Eigen::VectorXd varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi, P_f);
    Eigen::VectorXd dt_varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi, P_dtf);

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());
    workspace.Psi.resize(Psi.size());
    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
    ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
}
}
```

Setup a scalar field with inhomogeneous Gaussian random initial conditions.

Initialize a field and its derivative from a white noise power spectrum with cutoff k_{ast} , but with a large scale perturbation specified by Ψ . Ψ is initialized from a scale-invariant power spectrum with cutoff k_{Psi} .

9.18.2.4 perturbed_grf_and_comoving_curvature_fft

```
auto perturbed_grf_and_comoving_curvature_fft [inline]
```

Initialize an inhomogeneous Gaussian random field and the fft of some scale invariant comoving curvature perturbation.

This is the procedure used for section 4.2.2 of the paper.

9.18.2.5 perturbed_grf_without_saving_Psi

```
auto perturbed_grf_without_saving_Psi [inline]
```

Initial value:

```
=
[] (const auto param, auto &workspace) {
    Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
        param.k_Psi, -3);
    Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
        param.k_ast, 0);
    Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
    Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
    Eigen::VectorXd varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi, P_f);
    Eigen::VectorXd dt_varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi, P_dtf);

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());
    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
}
}
```

Same as `perturbed_grf`, but does not store Ψ in workspace to save memory.

9.18.2.6 plane_wave

```
auto plane_wave [inline]
```

Initial value:

```
=
[] (const auto param, auto &workspace) {
    const long long int N = param.N;
    Eigen::VectorXd varphi(N*N*N);
    for(int a = 0; a < N; ++a){
        for(int b = 0; b < N; ++b){
            for(int c = 0; c < N; ++c){
                varphi(Idx_OF(N, a, b, c)) = cos(2 * std::numbers::pi * c / N);
            }
        }
    }

    Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, 0);

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());
    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
}
```

Plane wave initial condition. For testing the numerical code.

9.18.2.7 unperturbed_grf

```
auto unperturbed_grf [inline]
```

Initial value:

```
=
[] (const auto param, auto &workspace) {
    Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
        param.k_ast, 0);
    Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
    Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
    Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());

    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
}
```

Initialize a field and its derivative from a white noise power spectrum with cutoff k_{ast} .

9.18.2.8 unperturbed_grf_and_fixed_curvature

```
auto unperturbed_grf_and_fixed_curvature [inline]
```

Initial value:

```
=
[] (const auto param, auto &workspace) {
    Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
        param.k_Psi, -3);
    Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
        param.k_ast, 0);
    Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
    Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
    Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
    Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());
    workspace.Psi.resize(Psi.size());
    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
    ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
}
```

Initialize a homogeneous Gaussian random field and some scale invariant curvature perturbation.

9.18.2.9 unperturbed_grf_with_background

```
auto unperturbed_grf_with_background [inline]
```

Initial value:

```
=
```

```
[](const auto param, auto &workspace) {
    Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
        param.k_ast, 0);
    Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
    Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
    varphi.array() += param.varphi_mean;
    Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());
    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
}
```

Initialize a field and its derivative from a white noise power spectrum with cutoff `k_ast`, plus homogeneous background.

9.18.2.10 unperturbed_grf_with_Psi

```
auto unperturbed_grf_with_Psi [inline]
```

Initial value:

```
=
```

```
[](const auto param, auto &workspace) {
    Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
        param.k_Psi, -3);
    Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
        param.k_ast, 0);
    Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
    Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
    Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
    Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);

    auto &state = workspace.state;
    state.resize(varphi.size() + dt_varphi.size());
    workspace.Psi.resize(Psi.size());
    ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
    ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
    ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
}
```

Same as `unperturbed_grf`, but with an extra scale-invariant `Psi`.

9.18.2.11 wave_packet

```
auto wave_packet [inline]
```

Initial value:

```
=
```

```
[](const auto param, auto &workspace) {
    const long long int N = param.N;
    Eigen::VectorXd varphi(N*N*N);
    Eigen::VectorXd dt_varphi(N*N*N);
    Eigen::VectorXd Psi(N*N*N);

    for(int a = 0; a < N; ++a){
        for(int b = 0; b < N; ++b){
            for(int c = 0; c < N; ++c){
                double dist_to_center = sqrt(std::pow(std::min((double)a, (double)std::abs(N-a)), 2) + (b - N/3) * (b - N/3) + (c - N/3) * (c - N/3)) * (param.L / param.N);
                varphi(IDX_OF(N, a, b, c)) = exp(- dist_to_center * dist_to_center / 40.0);
                dt_varphi(IDX_OF(N, a, b, c)) = 0;

                Psi(IDX_OF(N, a, b, c)) = - param.Psi_std_dev * cos(2 * std::numbers::pi * c / N);
            }
        }
    }
```

```

    }
}

auto &state = workspace.state;
state.resize(varphi.size() + dt_varphi.size());
ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());

workspace.Psi.resize(Psi.size());
ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
}

```

Wave packet initial condition.

9.19 initializer.hpp

[Go to the documentation of this file.](#)

```

00001
00012 #ifndef INITIALIZER_HPP
00013 #define INITIALIZER_HPP
00014
00015 #include "fdm3d.hpp"
00016 #include "random_field.hpp"
00017 #include "fftw_wrapper.hpp"
00018 #include "special_function.hpp"
00019 // #include "dispatcher.hpp"
00020 // #include "field_booster.hpp"
00021 // #include "param.hpp"
00022 // #include "physics.hpp"
00023
00024 #ifndef DISABLE_CUDA
00025 #include <thrust/device_vector.h>
00026 #include "cuda_wrapper.cuh"
00027 #define ALGORITHM_NAMESPACE thrust
00028 #else
00029 #define ALGORITHM_NAMESPACE std
00030 #endif
00031
00032 inline auto unperturbed_grf =
00033 [] (const auto param, auto &workspace) {
00034     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
00035 param.k_ast, 0);
00036     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00037     Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f); // Initial ULDM
00038     Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf); // Initial
00039     ULDM field time derivative
00040
00041     auto &state = workspace.state;
00042     state.resize(varphi.size() + dt_varphi.size());
00043     // thrust::copy handles both copies between Eigen::VectorXd and copies from Eigen::VectorXd to
00044     thrust::device_vector<double>
00045     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00046     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00047 };
00048
00049 inline auto unperturbed_grf_with_background =
00050 [] (const auto param, auto &workspace) {
00051     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
00052 param.k_ast, 0);
00053     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00054     Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
00055     varphi.array() += param.varphi_mean;
00056     Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);
00057
00058     auto &state = workspace.state;
00059     state.resize(varphi.size() + dt_varphi.size());
00060     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00061     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00062 };
00063
00064 inline auto perturbed_grf =
00065 [] (const auto param, auto &workspace) {
00066     Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
00067 param.k_Psi, -3);
00068     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
00069 param.k_ast, 0);
00070     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);

```



```

00076     Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
00077     Eigen::VectorXd varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi, P_f);
00078     Eigen::VectorXd dt_varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi,
P_dtf);
00079
00080     auto &state = workspace.state;
00081     state.resize(varphi.size() + dt_varphi.size());
00082     workspace.Psi.resize(Psi.size());
00083     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00084     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00085     ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
00086
00087     //std::cout << boost::typeid::type_id_runtime(workspace.Psi).pretty_name() << '\n';
00088 };
00089
00090
00091 inline auto perturbed_grf_without_saving_Psi =
00092 [] (const auto param, auto &workspace) {
00093     Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
param.k_Psi, -3);
00094     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
param.k_ast, 0);
00095     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00096     Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
00097     Eigen::VectorXd varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi, P_f);
00098     Eigen::VectorXd dt_varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi,
P_dtf);
00099
00100     auto &state = workspace.state;
00101     state.resize(varphi.size() + dt_varphi.size());
00102     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00103     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00104 };
00105
00106
00107 inline auto unperturbed_grf_with_Psi =
00108 [] (const auto param, auto &workspace) {
00109     Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
param.k_Psi, -3);
00110     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
param.k_ast, 0);
00111     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00112     Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
00113     Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
00114     Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);
00115
00116     auto &state = workspace.state;
00117     state.resize(varphi.size() + dt_varphi.size());
00118     workspace.Psi.resize(Psi.size());
00119     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00120     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00121     ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
00122 };
00123
00124 inline auto unperturbed_grf_and_fixed_curvature =
00125 [] (const auto param, auto &workspace) {
00126     Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
param.k_Psi, -3);
00127     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
param.k_ast, 0);
00128     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00129     Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
00130     Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
00131     Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);
00132
00133     auto &state = workspace.state;
00134     state.resize(varphi.size() + dt_varphi.size());
00135     workspace.Psi.resize(Psi.size());
00136     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00137     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00138     ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
00139 };
00140
00141 inline auto perturbed_grf_and_comoving_curvature_fft =
00142 [] (const auto param, auto &workspace) {
00143     using namespace std::numbers;
00144
00145     // Generate comoving curvature perturbation
00146     double eta_i = workspace.cosmology.eta(param.t_start);
00147     double A_s = (-576 * pow(pi, 6) * pow(eta_i, 6) * pow(param.Psi_std_dev, 2)) /
(-81 * pow(param.L, 4) * (pow(param.L, 2) + 2 * pow(pi, 2) * pow(eta_i, 2)) +
param.L *
(81 * pow(param.L, 5) - 54 * pow(param.L, 3) * pow(pi, 2) * pow(eta_i, 2) +
48 * param.L * pow(pi, 4) * pow(eta_i, 4)) *
cos(4 * pi * eta_i) / (sqrt(3) * param.L)) +
256 * pow(pi, 6) * pow(eta_i, 6) * Ci_pade_approximant_12_12((4 * pi * eta_i) / (sqrt(3) *

```

```

    param.L)) +
00163     4 * sqrt(3) * param.L * pi * eta_i *
00164     (27 * pow(param.L, 4) + 6 * pow(param.L, 2) * pow(pi, 2) * pow(eta_i, 2) -
00165     16 * pow(pi, 4) * pow(eta_i, 4)) *
00166     sin((4 * pi * eta_i) / (sqrt(3) * param.L));
00167 Spectrum P_R = scale_invariant_spectrum_3d(param.N, param.L, A_s);
00168
00169 // Manual cutoff for P_R at around horizon. The effect of imposing this cutoff is negligible.
00170 // Spectrum P_R_with_cutoff = [P_R](double k){ return k <= 0.5 ? P_R(k) : 0.0; };
00171
00172 Eigen::VectorXd R = generate_gaussian_random_field(param.N, param.L, P_R);
00173 // std::cout << "A_s = " << A_s << '\n';
00174
00175 // Calculate initial gravitational potential Psi.
00176 // Convention for potentials:  $\mathcal{R}_k = (3/2) \Psi_k$  for superhorizon.
00177 auto kernel = [eta_i](double k){
00178     return k == 0.0 ? 0.0 : (6 * sqrt(3) * (-(k * eta_i * cos((k * eta_i) / sqrt(3))) /
sqrt(3)) + sin((k * eta_i) / sqrt(3))) / (pow(k, 3) * pow(eta_i, 3));
00179 };
00180 auto fft_wrapper = fftwWrapper(param.N);
00181 Eigen::VectorXd Psi = compute_field_with_scaled_fourier_modes(param.N, param.L, R, kernel,
fft_wrapper);
00182
00183 // Calculate  $\varphi^2$ ,  $\dot{\varphi}^2$  perturbations as a multiple of Psi.
00184 // See Eqn (3.17) of paper.
00185 // There is an extra factor of 0.5 in front since "generate_inhomogeneous_gaussian_random_field"
use exp(2\Psi) ~ 1 + 2 \Psi for variance perturbation convention.
00186 double v = param.k_ast / (param.a1 * param.m);
00187 double alpha_varphi_sqr = 0.5 * (-3 * pow(4*pow(v,2)+5, 2)) / (12*pow(v,4) + 50*pow(v,2) + 50);
00188 double alpha_dot_varphi_sqr = 0.5 * (25 - 20*pow(v,2)) / (12*pow(v,4) + 50*pow(v,2) + 50);
00189
00190 Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
param.k_ast, 0);
00191 Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00192 Eigen::VectorXd varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L,
alpha_varphi_sqr * Psi, P_f);
00193 Eigen::VectorXd dt_varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L,
alpha_dot_varphi_sqr * Psi, P_dtf);
00194
00195 auto &state = workspace.state;
00196 state.resize(varphi.size() + dt_varphi.size());
00197 ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00198 ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00199
00200 // Save the comoving curvature perturbation for reference
00201 {
00202     decltype(workspace.state) R_dvec(R.size());
00203     ALGORITHM_NAMESPACE::copy(R.begin(), R.end(), R_dvec.begin());
00204     workspace.R_fft = workspace.fft_wrapper.execute_d2z(R_dvec);
00205 }
00206 };
00207
00208
00209
00210 inline auto homogeneous_field =
00211 [](const auto param, auto &workspace) {
00212     const long long int N = param.N;
00213     Eigen::VectorXd varphi = Eigen::VectorXd::Constant(N*N*N, param.f);
00214     Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, param.dt_f);
00215
00216     auto &state = workspace.state;
00217     state.resize(varphi.size() + dt_varphi.size());
00218     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00219     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00220 };
00221
00222 inline auto homogeneous_field_with_fluctuations =
00223 [](const auto param, auto &workspace) {
00224     const long long int N = param.N;
00225     Eigen::VectorXd varphi = Eigen::VectorXd::Constant(N*N*N, param.f);
00226     Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, 0.0);
00227
00228     Spectrum P_delta_varphi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L,
param.delta_varphi_std_dev, param.k_delta_varphi, -3);
00229     Eigen::VectorXd delta_varphi = generate_gaussian_random_field(param.N, param.L, P_delta_varphi);
00230     varphi += delta_varphi;
00231
00232     auto &state = workspace.state;
00233     state.resize(varphi.size() + dt_varphi.size());
00234     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00235     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00236 };
00237
00238 inline auto plane_wave =
00239 [](const auto param, auto &workspace) {

```

```

00253     const long long int N = param.N;
00254     Eigen::VectorXd varphi(N*N*N);
00255     for(int a = 0; a < N; ++a){
00256         for(int b = 0; b < N; ++b){
00257             for(int c = 0; c < N; ++c){
00258                 varphi(IDX_OF(N, a, b, c)) = cos(2 * std::numbers::pi * c / N);
00259             }
00260         }
00261     }
00262
00263     Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, 0);
00264
00265     auto &state = workspace.state;
00266     state.resize(varphi.size() + dt_varphi.size());
00267     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00268     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00269 };
00270
00271 inline auto wave_packet =
00272 [] (const auto param, auto &workspace) {
00273     const long long int N = param.N;
00274     Eigen::VectorXd varphi(N*N*N);
00275     Eigen::VectorXd dt_varphi(N*N*N);
00276     Eigen::VectorXd Psi(N*N*N);
00277
00278     for(int a = 0; a < N; ++a){
00279         for(int b = 0; b < N; ++b){
00280             for(int c = 0; c < N; ++c){
00281                 double dist_to_center = sqrt(std::pow(std::min((double)a, (double)std::abs(N-a)), 2) + (b - N/3)
00282 * (b - N/3) + (c - N/3) * (c - N/3)) * (param.L / param.N);
00283                 varphi(IDX_OF(N, a, b, c)) = exp(- dist_to_center * dist_to_center / 40.0);
00284                 dt_varphi(IDX_OF(N, a, b, c)) = 0;
00285                 //Psi(IDX_OF(N, a, b, c)) = - param.Psi_std_dev * exp( - (b - N/2) * (b - N/2) / (2 * (param.L *
00286 param.L / 3.0 / 3.0)));
00287                 Psi(IDX_OF(N, a, b, c)) = - param.Psi_std_dev * cos(2 * std::numbers::pi * c / N);
00288             }
00289         }
00290     }
00291
00292     auto &state = workspace.state;
00293     state.resize(varphi.size() + dt_varphi.size());
00294     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00295     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00296
00297     workspace.Psi.resize(Psi.size());
00298     ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
00299 };
00300 #endif

```

9.20 src/io.hpp File Reference

Input/output utilities.

```

#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <iomanip>
#include <Eigen/Dense>

```

Functions

- `std::vector< double > load_vector_from_file (std::string filename)`
- `void write_vector_to_file (std::vector< double > vector, std::string filename)`
- `void write_data_to_file (const char *buf, ssize_t size, std::string filename)`
- `void write_VectorXd_to_file (const Eigen::VectorXd &vector, std::string filename)`
- `void write_VectorXd_to_filename_template (const Eigen::VectorXd &vector, const std::string format_string, const int idx)`
- `Eigen::VectorXd load_VectorXd_from_file (const std::string &filename)`

9.20.1 Detailed Description

Input/output utilities.

Author

Siyang Ling

9.21 io.hpp

[Go to the documentation of this file.](#)

```
00001
00006 #ifndef IO_HPP
00007 #define IO_HPP
00008 #include <cstdlib>
00009 #include <iostream>
00010 #include <fstream>
00011 #include <string>
00012 #include <vector>
00013 #include <iomanip>
00014
00015 #include <Eigen/Dense>
00016
00017
00018 std::vector<double> load_vector_from_file(std::string filename);
00019 void write_vector_to_file(std::vector<double> vector, std::string filename);
00020 void write_data_to_file(const char *buf, ssize_t size, std::string filename);
00021
00022 void write_VectorXd_to_file(const Eigen::VectorXd &vector, std::string filename);
00023 void write_VectorXd_to_filename_template(const Eigen::VectorXd &vector, const std::string
format_string, const int idx);
00024 Eigen::VectorXd load_VectorXd_from_file(const std::string &filename);
00025
00026
00027 #endif
```

9.22 src/midpoint.hpp File Reference

Midpoint method implementation for odeint stepper.

```
#include <boost/numeric/odeint/algebra/default_operations.hpp>
#include <boost/numeric/odeint/algebra/algebra_dispatcher.hpp>
#include <boost/numeric/odeint/algebra/operations_dispatcher.hpp>
#include <boost/numeric/odeint/util/state_wrapper.hpp>
#include <boost/numeric/odeint/util/is_resizeable.hpp>
#include <boost/numeric/odeint/util/resizer.hpp>
#include "cuda_wrapper.cuh"
```

Classes

- class [midpoint< State, Value, Deriv, Time, Algebra, Operations, Resizer >](#)

9.22.1 Detailed Description

Midpoint method implementation for odeint stepper.

Author

Siyang Ling

9.23 midpoint.hpp

[Go to the documentation of this file.](#)

```

00001
00006 #ifndef MIDPOINT_HPP
00007 #define MIDPOINT_HPP
00008
00009
00010
00011 // #include <boost/numeric/odeint/stepper/base/explicit_stepper_base.hpp>
00012 // #include <boost/numeric/odeint/algebra/range_algebra.hpp>
00013 #include <boost/numeric/odeint/algebra/default_operations.hpp>
00014 #include <boost/numeric/odeint/algebra/algebra_dispatcher.hpp>
00015 #include <boost/numeric/odeint/algebra/operations_dispatcher.hpp>
00016
00017 #include <boost/numeric/odeint/util/state_wrapper.hpp>
00018 #include <boost/numeric/odeint/util/is_resizeable.hpp>
00019 #include <boost/numeric/odeint/util/resizer.hpp>
00020
00021 #ifndef DISABLE_CUDA
00022 #include "cuda_wrapper.cuh"
00023 #endif
00024
00025 template<
00026     class State,
00027     class Value = double,
00028     class Deriv = State,
00029     class Time = Value,
00030     class Algebra = typename boost::numeric::odeint::algebra_dispatcher<State>::algebra_type,
00031     class Operations = typename boost::numeric::odeint::operations_dispatcher<State>::operations_type,
00032     class Resizer = boost::numeric::odeint::initially_resizer //boost::numeric::odeint::always_resizer
00033 >
00034 class midpoint : public boost::numeric::odeint::algebra_stepper_base<Algebra, Operations>
00035 {
00036 public :
00037     typedef State state_type;
00038     typedef State deriv_type;
00039     typedef Value value_type;
00040     typedef Time time_type;
00041     typedef unsigned short order_type;
00042     typedef boost::numeric::odeint::stepper_tag stepper_category;
00043
00044     typedef boost::numeric::odeint::algebra_stepper_base<Algebra, Operations> algebra_stepper_base_type;
00045     typedef typename algebra_stepper_base_type::algebra_type algebra_type;
00046     typedef typename algebra_stepper_base_type::operations_type operations_type;
00047
00048     static order_type order(void) { return 2; }
00049
00050     midpoint(){}
00051
00052     template<class System>
00053     void do_step(System system, State &in, Time t, Time dt)
00054     {
00055         static const Value vall = static_cast<Value>(1);
00056         const Time dh = dt / static_cast<Value>(2);
00057         const Time th = t + dh;
00058
00059         //m_resizer.adjust_size(in, boost::numeric::odeint::detail::bind(&stepper_type::template
resize_impl<State>, boost::numeric::odeint::detail::ref(*this), boost::numeric::odeint::detail::_1));
00060         m_resizer.adjust_size(in, [&](const auto &arg){ return resize_impl(arg); });
00061
00062         typename boost::numeric::odeint::unwrap_reference<System>::type &sys = system;
00063
00064         sys(in, deriv_tmp.m_v, t);
00065         algebra_stepper_base_type::m_algebra.for_each3(state_tmp.m_v, in, deriv_tmp.m_v,
00066             typename operations_type::template scale_sum2<Value, Time>(vall, dh));
00067
00068         sys(state_tmp.m_v, deriv_tmp.m_v, th);
00069         algebra_stepper_base_type::m_algebra.for_each3(state_tmp.m_v, in, deriv_tmp.m_v,
00070             typename operations_type::template scale_sum2<Value, Time>(vall, dt));
00071
00072         in.swap(state_tmp.m_v);
00073
00074         // Release memory
00075         //m_resizer.adjust_size(State(), [&](const auto &arg){ return resize_impl(arg); });
00076         // deriv_tmp.m_v.clear();
00077         // State().swap(deriv_tmp.m_v);
00078         // state_tmp.m_v.clear();
00079         // State().swap(state_tmp.m_v);
00080     }
00081
00082     // template<class StateType>
00083     // void adjust_size(const StateType &x)
00084     // {
00085     //     resize_impl(x);

```

```

00086 // }
00087
00088 bool resize_impl(const State &x)
00089 {
00090     bool resized = false;
00091     resized |= boost::numeric::odeint::adjust_size_by_resizeability(deriv_tmp, x, typename
boost::numeric::odeint::is_resizeable<State>::type());
00092     resized |= boost::numeric::odeint::adjust_size_by_resizeability(state_tmp, x, typename
boost::numeric::odeint::is_resizeable<State>::type());
00093     return resized;
00094 }
00095
00096 private:
00097     Resizer m_resizer;
00098
00099     boost::numeric::odeint::state_wrapper<State> deriv_tmp;
00100     boost::numeric::odeint::state_wrapper<State> state_tmp;
00101 };
00102
00103
00104
00105 #endif

```

9.24 src/observer.hpp File Reference

Implements "observers", which controls what gets saved during simulations.

```

#include <cstdlib>
#include <iostream>
#include <string>
#include <type_traits>
#include "Eigen/Dense"
#include <boost/numeric/odeint.hpp>
#include <boost/numeric/odeint/external/eigen/eigen.hpp>
#include "odeint_eigen/eigen_operations.hpp"
#include "eigen_wrapper.hpp"
#include "fdm3d.hpp"
#include "io.hpp"
#include "physics.hpp"
#include "workspace.hpp"
#include "cuda_wrapper.cuh"
#include "fdm3d_cuda.cuh"

```

Classes

- struct [ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density >](#)
An "observer" used to save spectra and slices during the simulation at roughly constant time intervals.

9.24.1 Detailed Description

Implements "observers", which controls what gets saved during simulations.

Author

Siyang Ling

Observers are used by the odeint library. The `operator()` function of the observer is called at each time step. See https://www.boost.org/doc/libs/1_85_0/libs/numeric/odeint/doc/html/boost_numeric_odeint/odeint_in_detail/integrate_functions.html for details on how observers are used for a simulation.

9.25 observer.hpp

[Go to the documentation of this file.](#)

```

00001
00011 #ifndef OBSERVER_HPP
00012 #define OBSERVER_HPP
00013
00014 #include <cstdlib>
00015 #include <iostream>
00016 #include <string>
00017 #include <type_traits>
00018
00019 #include "Eigen/Dense"
00020
00021 #include <boost/numeric/odeint.hpp>
00022 #include <boost/numeric/odeint/external/eigen/eigen.hpp>
00023
00024 #include "odeint_eigen/eigen_operations.hpp"
00025
00026 #include "eigen_wrapper.hpp"
00027 #include "fdm3d.hpp"
00028 #include "io.hpp"
00029 #include "physics.hpp"
00030 #include "workspace.hpp"
00031
00032 #ifndef DISABLE_CUDA
00033 #include "cuda_wrapper.cuh"
00034 #include "fdm3d_cuda.cuh"
00035 #endif
00036
00064 template<typename Equation,
00065         bool save_field_spectrum = true,
00066         bool save_density_spectrum = true,
00067         bool save_density = false>
00068 struct ConstIntervalObserver {
00069     typedef typename Equation::Workspace Workspace;
00070     typedef typename Workspace::State State;
00071     typedef State Vector;
00072     Workspace &workspace;
00073     int idx;
00074     std::string dir;
00075     double t_start;
00076     double t_end;
00077     double t_interval;
00078     double t_last;
00079
00080     template<typename Param>
00081     ConstIntervalObserver(const std::string &dir_, const Param &param, Equation &eqn) :
00082         workspace(eqn.workspace), idx(0), dir(dir_),
00083         t_start(param.t_start), t_end(param.t_end), t_interval(param.t_interval), t_last(param.t_start) {}
00084
00085     ConstIntervalObserver(const ConstIntervalObserver &) = default;
00086
00087     void operator()(const State &x, double t)
00088     {
00089         if(t >= t_last + t_interval || t == t_end || t == t_start) {
00090             const long long int N = workspace.N;
00091             const double L = workspace.L;
00092             const double m = workspace.m;
00093             const double a_t = workspace.cosmology.a(t);
00094
00095             if constexpr(save_field_spectrum) {
00096                 Vector varphi_plus_spectrum = compute_mode_power_spectrum(N, L, m, a_t, workspace.state,
00097 workspace.fft_wrapper);
00098                 Eigen::VectorXd varphi_plus_spectrum_out(varphi_plus_spectrum.size());
00099                 copy_vector(varphi_plus_spectrum_out, varphi_plus_spectrum);
00099                 write_VectorXd_to_filename_template(varphi_plus_spectrum_out, dir +
00100 "varphi_plus_spectrum_%d.dat", idx);
00101             }
00102
00103             if constexpr(save_density_spectrum) {
00104                 Vector rho = Equation::compute_energy_density(workspace, t);
00105                 Vector rho_spectrum = compute_power_spectrum(N, rho, workspace.fft_wrapper);
00106                 Eigen::VectorXd rho_spectrum_out(rho_spectrum.size());
00107                 copy_vector(rho_spectrum_out, rho_spectrum);
00108                 write_VectorXd_to_filename_template(rho_spectrum_out, dir + "rho_spectrum_%d.dat", idx);
00109             }
00110
00111             if constexpr(save_density) {
00112                 Vector rho = Equation::compute_energy_density(workspace, t);
00113                 Eigen::VectorXd rho_copy(rho.size());
00114                 copy_vector(rho_copy, rho);
00115                 Eigen::VectorXd rho_slice = rho_copy.head(N*N); // Save the density for a = 0 slice.
00116                 Eigen::VectorXd rho_axis_average = rho_copy.resized(N*N, N).rowwise().mean(); // Save the
00117 density overaged over a axis.

```

```

00116
00117     write_VectorXd_to_filename_template(rho_slice, dir + "rho_slice_%d.dat", idx);
00118     write_VectorXd_to_filename_template(rho_axis_average, dir + "rho_axis_average_%d.dat", idx);
00119 }
00120
00121     workspace.t_list.push_back(t);
00122     t_last = t;
00123     ++idx;
00124 }
00125 }
00126 };
00127
00128
00129
00130 #endif

```

9.26 src/param.hpp File Reference

Utilities for managing simulations parameters.

```

#include "utility.hpp"
#include "boost/pfr.hpp"
#include "boost/type_index.hpp"
#include <fstream>
#include <string>

```

Classes

- struct [SampleParam](#)
A sample parameter type specifying a lambda-phi-4 theory in an FRW background.

Functions

- template<typename T>
void print_param (const T ¶m)
Pretty prints a parameter struct T.
- template<typename T>
void save_param_names (const std::string &filename)
Save the member names of parameter struct T to filename.
- template<typename T>
void save_param_Mathematica_formats (const std::string &filename)
Save the member types of parameter struct T to filename. Type names are in Mathematica convention.
- template<typename T>
void save_param_for_Mathematica (const T ¶m, const std::string &dir)
Save member names, types and values of param to directory dir.
- template<typename T>
void save_param_types (const std::string &filename)

9.26.1 Detailed Description

Utilities for managing simulations parameters.

Author

Siyang Ling

This header file contains utilities for pretty-printing and saving parameters of a simulation. By convention, we collect all parameters in a (trivial, standard layout) struct containing double's or long long int's. (e.g. [SampleParam](#)) The utilities here are generic for different parameter structs; you can define your own new type containing new parameters, and use the utilities here as usual. Typically, we use these utilities to export a struct along with some meta-information, so that external code (Mathematica / Python) can also use the parameters.

9.27 param.hpp

[Go to the documentation of this file.](#)

```

00001
00014 #ifndef PARAM_HPP
00015 #define PARAM_HPP
00016
00017 #include "utility.hpp"
00018 #include "boost/pfr.hpp"
00019 #include "boost/type_index.hpp"
00020 #include <fstream>
00021 #include <string>
00022
00026 struct SampleParam {
00027     long long int N;
00028     double L;
00029     double m;
00030     double lambda;
00031     double k_ast;
00032     double varphi_std_dev;
00033     double a1;
00034     double H1;
00035     double t1;
00036 };
00037
00041 template<typename T>
00042 void print_param(const T &param) {
00043     auto names = boost::pfr::names_as_array<T>();
00044     auto func = [&](const auto &field, std::size_t i) {
00045         std::cout << names[i] << ": " << field
00046             << " (" << boost::typeindex::type_id_runtime(field) << ")\n";
00047     };
00048     // std::cout << line_separator_with_description("The parameters for the simulation") << '\n';
00049     // boost::pfr::for_each_field(param, func);
00050     // std::cout << line_separator_with_description() << '\n';
00051     auto c = [&]() { boost::pfr::for_each_field(param, func); };
00052     run_and_print("The parameters for the simulation", c);
00053 }
00054
00058 template<typename T>
00059 void save_param_names(const std::string &filename) {
00060     std::ofstream outstream(filename);
00061     auto names = boost::pfr::names_as_array<T>();
00062     for(auto name : names) {
00063         outstream << name << '\n';
00064     }
00065 }
00066
00067 /*
00068 // Compiles with Intel icpx, but doesn't compile with gcc due to "Explicit template specialization
    cannot have a storage class"
00069 template<typename T> std::string_view Mathematica_format;
00070
00071 template<> constexpr static std::string_view Mathematica_format<double> = "Real64";
00072
00073 template<> constexpr static std::string_view Mathematica_format<long long int> = "Integer64";
00074 */
00075
00076 /*
00077 // Compiles with gcc, fails at link stage with Intel icpx due to multiple definitions
00078 template<typename T> std::string_view Mathematica_format;
00079
00080 template<> constexpr static std::string_view Mathematica_format<double> = "Real64";
00081
00082 template<> constexpr static std::string_view Mathematica_format<long long int> = "Integer64";
00083 */
00084
00085 namespace {
00086     template<typename T> std::string_view Mathematica_format;
00087
00088     template<> constexpr static std::string_view Mathematica_format<double> = "Real64";
00089
00090     template<> constexpr static std::string_view Mathematica_format<long long int> = "Integer64";
00091 }
00092
00096 template<typename T>
00097 void save_param_Mathematica_formats(const std::string &filename) {
00098     std::ofstream outstream(filename);
00099     auto func = [&](const auto &field) {
00100         typedef std::remove_const_t<std::remove_reference_t<decltype(field)>> type_of_field;
00101         outstream << Mathematica_format<type_of_field> << '\n';
00102     };
00103     boost::pfr::for_each_field(T(), func);
00104 }
00105

```

```

00109 template<typename T>
00110 static void save_param(const T &param, const std::string &filename){
00111     std::ofstream outstream(filename, std::ios::binary);
00112     if(outstream.is_open()){
00113         outstream.write((const char *)&param, sizeof(T));
00114     }
00115 }
00116
00120 template<typename T>
00121 void save_param_for_Mathematica(const T &param, const std::string &dir) {
00122     save_param_names<T>(dir + "paramNames.txt");
00123     save_param_Mathematica_formats<T>(dir + "paramTypes.txt");
00124     save_param<T>(param, dir + "param.dat");
00125 }
00126
00127
00128 template<typename T>
00129 void save_param_types(const std::string &filename) {
00130     std::ofstream outstream(filename);
00131     auto func = [&](const auto &field) {
00132         outstream << boost::typeindex::type_id_runtime(field) << '\n';
00133     };
00134     boost::pfr::for_each_field(T(), func);
00135 }
00136
00137
00138
00139
00140 #endif

```

9.28 src/physics.hpp File Reference

Collection of repeatedly used physics formulas. (e.g. FRW cosmology related formulas)

```
#include <cmath>
```

Classes

- struct [StaticEOSCosmology](#)

A convenience class used to calculate FRW related quantities for constant EOS spacetimes.

9.28.1 Detailed Description

Collection of repeatedly used physics formulas. (e.g. FRW cosmology related formulas)

Author

Siyang Ling

9.29 physics.hpp

[Go to the documentation of this file.](#)

```

00001
00006 #ifndef PHYSICS_HPP
00007 #define PHYSICS_HPP
00008
00009 #include <cmath>
00010 // #include "param.hpp"
00011
00015 struct StaticEOSCosmology {
00016     double a1;

```

```

00017 double H1;
00018 double t1;
00019 double p;
00021 StaticEOSCosmology(const double a1_, const double H1_, const double t1_, const double p_)
00022     : a1(a1_), H1(H1_), t1(t1_), p(p_) {}
00023
00025 template<typename T>
00026 StaticEOSCosmology(const T &param)
00027     : a1(param.a1), H1(param.H1), t1(param.t1), p(1.0) {}
00028
00030 StaticEOSCosmology(void)
00031     : a1(1.0), H1(0), t1(0), p(1.0) {}
00032
00034 double a(const double t) const {
00035     return a1 * pow(1 + (1 + 1 / p) * H1 * (t - t1), p / (1 + p));
00036 }
00037
00039 double H(const double t) const {
00040     return H1 * pow(1 + (1 + 1 / p) * H1 * (t - t1), -1);
00041 }
00042
00052 double eta(const double t) const {
00053     //return etal + (p / (a1 * H1)) * (-1 + pow(1 + (1 + 1 / p) * H1 * (t - t1), 1 / (1 + p)));
00054     return (p / (a1 * H1)) * pow(1 + (1 + 1 / p) * H1 * (t - t1), 1 / (1 + p));
00055 }
00056 };
00057
00058
00059 #endif

```

9.30 src/random_field.hpp File Reference

Utilities for generating Gaussian random fields of given spectrum and inhomogeneity.

```

#include "Eigen/Dense"
#include <functional>
#include <random>
#include <vector>

```

Typedefs

- `typedef std::function< double(const double)> Spectrum`
Typedef for spectrum $P(k)$. Given momentum k , the spectrum should return $P(k)$.

Functions

- `void RandomNormal::set_generator_seed (std::mt19937::result_type seed)`
- `std::mt19937 RandomNormal::get_generator_from_device ()`
- `double RandomNormal::generate_random_normal ()`
- `Spectrum power_law_with_cutoff_given_amplitude_3d (const long long int N, const double L, const double sigma, const double k_ast, const double alpha)`
 k^α power law spectrum with a sharp cutoff at k_ .*
- `Spectrum broken_power_law_given_amplitude_3d (const long long int N, const double L, const double sigma, const double k_ast, const double alpha, const double beta)`
Broken power law spectrum with the break at k_ .*
- `Spectrum scale_invariant_spectrum_3d (const long long int N, const double L, const double As)`
 k^α power law spectrum with a sharp cutoff at k_ .*
- `Spectrum to_deriv_spectrum (const double m, const Spectrum &P_f)`
Given spectrum P_φ , return a new spectrum given by $P_\varphi(k) = (k^2 + m^2)P_\varphi(k)$.
- `Spectrum to_deriv_spectrum (const double m, const double a, const Spectrum &P_f)`

Given spectrum P_φ , return a new spectrum given by $P_\varphi(k) = (k^2/a^2 + m^2)P_\varphi(k)$.

- [Eigen::VectorXd](#) **generate_gaussian_random_field** ([const long long int](#) N, [const double](#) L, [const Spectrum](#) &P)

Special case of *generate_inhomogeneous_gaussian_random_field*.

- [Eigen::VectorXd](#) **generate_inhomogeneous_gaussian_random_field** ([const long long int](#) N, [const double](#) L, [const Eigen::VectorXd](#) &Psi, [const Spectrum](#) &P)

Generate an inhomogeneous 3D real Gaussian random field from spectral data $P(k)$.

9.30.1 Detailed Description

Utilities for generating Gaussian random fields of given spectrum and inhomogeneity.

Author

Siyang Ling

This file contains utilities for generating Gaussian random fields (GRF), including some example spectra and a function for generating field realizations from a spectra. See function `generate_inhomogeneous_gaussian_random_field` for details.

9.30.2 Function Documentation

9.30.2.1 broken_power_law_given_amplitude_3d()

```
Spectrum broken_power_law_given_amplitude_3d (
    const long long int N,
    const double L,
    const double sigma,
    const double k_ast,
    const double alpha,
    const double beta )
```

Broken power law spectrum with the break at k_* .

Parameters

<i>N</i>	Number of lattice points.
<i>L</i>	Box size.
<i>sigma</i>	Standard deviation σ of generated function f .
<i>k_ast</i>	The break k_* .
<i>alpha</i>	Power law index α .
<i>beta</i>	Power law index β .

Returns

The spectrum P , which can be called to get $P(k)$.

The spectrum is given by

$$\begin{aligned} P(0) &= 0 \\ P(k) &= P(k_0)(k/k_0)^\alpha \text{ for } k < k_0 \\ P(k) &= P(k_0)(k/k_0)^\beta \text{ for } k > k_0 \\ \overline{f^2} &= \sigma^2 \end{aligned}$$

9.30.2.2 generate_inhomogeneous_gaussian_random_field()

```
Eigen::VectorXd generate_inhomogeneous_gaussian_random_field (
    const long long int N,
    const double L,
    const Eigen::VectorXd & Psi,
    const Spectrum & P )
```

Generate an inhomogeneous 3D real Gaussian random field from spectral data $P(k)$.

Parameters

N	Number of lattice points.
L	Box size.
Ψ	The inhomogeneity function ψ , given in terms of values on the lattice (of size N^3).
P	The spectrum P .

Returns

The generated GRF, as values on the lattice (of size N^3).

Generate an inhomogeneous Gaussian random field f , such that the spectrum of f is P , and the variance of the field has inhomogeneity like $\langle f^2(x) \rangle \approx \overline{f^2} e^{2\psi(x)}$. See section 3.2 of paper for details of this procedure.

9.30.2.3 power_law_with_cutoff_given_amplitude_3d()

```
Spectrum power_law_with_cutoff_given_amplitude_3d (
    const long long int N,
    const double L,
    const double sigma,
    const double k_ast,
    const double alpha )
```

k^α power law spectrum with a sharp cutoff at k_* .

Parameters

N	Number of lattice points.
L	Box size.
σ	Standard deviation σ of generated function f .
k_ast	Cutoff k_* .
α	Power law index α .

Returns

The spectrum P , which can be called to get $P(k)$.

The spectrum is given by

$$\begin{aligned} P(0) &= 0 \\ P(k) &= P(k_0)(k/k_0)^\alpha \text{ for } k < k_0 \\ \overline{f^2} &= \sigma^2 \end{aligned}$$

9.30.2.4 scale_invariant_spectrum_3d()

```
Spectrum scale_invariant_spectrum_3d (
    const long long int N,
    const double L,
    const double As )
```

k^α power law spectrum with a sharp cutoff at k_* .

Parameters

N	Number of lattice points.
L	Box size.
A_s	The height of the spectrum A_s .

Returns

The spectrum P , which can be called to get $P(k)$.

The spectrum is given by

$$\begin{aligned} P(0) &= 0 \\ P(k) &= A_s \end{aligned}$$

9.31 random_field.hpp

[Go to the documentation of this file.](#)

```
00001
00010 #ifndef RANDOM_FIELD_HPP
00011 #define RANDOM_FIELD_HPP
00012
00013 #include "Eigen/Dense"
00014
00015 #include <functional>
00016 #include <random>
00017 #include <vector>
00018
00019
00020 // A self-initializing random number generator for standard normal distribution
00021 namespace RandomNormal
00022 {
00023     void set_generator_seed(std::mt19937::result_type seed);
00024     std::mt19937 get_generator_from_device();
00025     double generate_random_normal();
00026 }
```

```

00027
00031 typedef std::function<double(const double)> Spectrum;
00032
00033 // Typical spectra.
00034
00051 Spectrum power_law_with_cutoff_given_amplitude_3d(const long long int N, const double L, const double
sigma, const double k_ast, const double alpha);
00052
00071 Spectrum broken_power_law_given_amplitude_3d(const long long int N, const double L, const double
sigma, const double k_ast, const double alpha, const double beta);
00072
00086 Spectrum scale_invariant_spectrum_3d(const long long int N, const double L, const double As);
00087
00091 Spectrum to_deriv_spectrum(const double m, const Spectrum &P_f);
00092
00096 Spectrum to_deriv_spectrum(const double m, const double a, const Spectrum &P_f);
00097
00101 Eigen::VectorXd generate_gaussian_random_field(const long long int N, const double L, const Spectrum
&P);
00102
00115 Eigen::VectorXd generate_inhomogeneous_gaussian_random_field(const long long int N, const double L,
const Eigen::VectorXd &Psi, const Spectrum &P);
00116
00117
00118
00119
00120 #endif

```

9.32 special_function.hpp

```

00001 #ifndef SPECIAL_FUNCTION_HPP
00002 #define SPECIAL_FUNCTION_HPP
00003
00004 // Pade approximant for Si(x), with m=15, n=12
00005 inline double Si_pade_approximant_15_12(double x) {
00006     using namespace std;
00007     return (x - 0.045439340981633 * pow(x, 3) + 0.0011545722575101668 * pow(x, 5) -
00008             0.000014101853682133025 * pow(x, 7) + 9.432808094387131e-8 * pow(x, 9) -
00009             3.5320197899716837e-10 * pow(x, 11) + 7.08240282274876e-13 * pow(x, 13) -
00010             6.053382120104225e-16 * pow(x, 15)) /
00011             (1. + 0.010116214573922555 * pow(x, 2) + 0.000049917511616975513 * pow(x, 4) +
00012             1.556549863087456e-7 * pow(x, 6) + 3.280675710557897e-10 * pow(x, 8) +
00013             4.5049097575386586e-13 * pow(x, 10) + 3.211070511937122e-16 * pow(x, 12));
00014 }
00015
00016 // Pade approximant for Ci(x), with m=12, n=12
00017 inline double Ci_pade_approximant_12_12(double x) {
00018     using namespace std;
00019     return log(x) + (0.5772156649015329 - 0.24231497614160186 * pow(x, 2) +
00020             0.007139183039136621 * pow(x, 4) - 0.00011466618094101764 * pow(x, 6) +
00021             8.443734405201243e-7 * pow(x, 8) - 3.060472574705558e-9 * pow(x, 10) +
00022             4.328624073851291e-12 * pow(x, 12)) /
00023             (1. + 0.013313955815300189 * pow(x, 2) + 0.00008836441800952094 * pow(x, 4) +
00024             3.800404484365274e-7 * pow(x, 6) + 1.1376490214488613e-9 * pow(x, 8) +
00025             2.297129602871981e-12 * pow(x, 10) + 2.510407760855278e-15 * pow(x, 12));
00026 }
00027
00028 #endif

```

9.33 src/utility.hpp File Reference

Utilities for debugging / profiling / pretty printing.

```

#include <iostream>
#include <iomanip>
#include <chrono>
#include <filesystem>

```

Functions

- `template<typename Callable >`
`void profile_function (long long int repeat, Callable &&c)`

9.33.1 Detailed Description

Utilities for debugging / profiling / pretty printing.

Author

Siyang Ling

9.34 utility.hpp

[Go to the documentation of this file.](#)

```

00001
00006 #ifndef UTILITY_HPP
00007 #define UTILITY_HPP
00008
00009 #include <iostream>
00010 #include <iomanip>
00011 #include <chrono>
00012 #include <filesystem>
00013
00014 // Pretty print functions
00015 inline static std::string line_separator_with_description(const std::string &description) {
00016     std::string result(80, '=');
00017     const int length = description.length() + 2;
00018     result.replace(80 / 2 - length / 2, length, " " + description + " ");
00019     return result;
00020 }
00021
00022
00023 inline static std::string line_separator_with_description(void) {
00024     std::string result(80, '=');
00025     return result;
00026 }
00027
00028
00029 template<typename Callable>
00030 static void run_and_print(const std::string &description, const Callable &c) {
00031     std::cout << line_separator_with_description(description) << '\n';
00032     c();
00033     std::cout << line_separator_with_description() << '\n';
00034 }
00035
00036
00037 template<typename Callable>
00038 static void run_and_measure_time(const std::string &description, const Callable &c) {
00039     std::cout << line_separator_with_description(description) << '\n';
00040     auto time_start = std::chrono::system_clock::now();
00041     c();
00042     auto time_end = std::chrono::system_clock::now();
00043     std::chrono::duration<double> time_diff = time_end - time_start;
00044     std::cout << std::fixed << std::setprecision(9) << std::left;
00045     std::cout << std::setw(9) << "time spent = " << time_diff.count() << " s" << '\n';
00046     std::cout << line_separator_with_description() << '\n';
00047 }
00048
00049
00050 static void prepare_directory_for_output(const std::string &dir) {
00051     const std::filesystem::path dir_path(dir);
00052     std::error_code ec;
00053     std::cout << line_separator_with_description("Preparing directory for output") << '\n';
00054     std::cout << "Saving results in directory: " << dir << '\n';
00055     std::filesystem::create_directories(dir_path, ec);
00056     std::cout << "ErrorCode = " << ec.message() << '\n';
00057     std::cout << line_separator_with_description() << '\n';
00058 }
00059
00060
00061 // Simple profiler for a big task, taking many cycles
00062 // Note that the function call incurs some time cost, so this is not totally accurate
00063 template<typename Callable>
00064 inline void profile_function(long long int repeat, Callable &&c) {
00065     auto time_start = std::chrono::system_clock::now();
00066     for(long long int i = 0; i < repeat; ++i) {
00067         c();
00068     }
00069     std::cout << line_separator_with_description("Profiling a callable") << '\n';
00070     auto time_end = std::chrono::system_clock::now();

```



```

00071     std::chrono::duration<double> time_diff = time_end - time_start;
00072     std::cout << std::fixed << std::setprecision(9) << std::left;
00073     std::cout << std::setw(9) << "total time spent = " << time_diff.count() << " s" << '\n';
00074     std::cout << std::setw(9) << "time spent per iteration = " << time_diff.count() / repeat << " s" << '\n';
00075     std::cout << line_separator_with_description() << '\n';
00076 }
00077
00078
00079
00080 #endif

```

9.35 src/wkb.hpp File Reference

Implementation of the WKB solution.

```

#include "Eigen/Dense"
#include "workspace.hpp"

```

Classes

- struct [WKBSolutionForKleinGordonEquationInFRW](#)

9.35.1 Detailed Description

Implementation of the WKB solution.

Used to extend an existing field profile to a later time.

9.36 wkb.hpp

[Go to the documentation of this file.](#)

```

00001
00007 #ifndef WKB_HPP
00008 #define WKB_HPP
00009
00010 #include "Eigen/Dense"
00011 #include "workspace.hpp"
00012
00013 struct WKBSolutionForKleinGordonEquationInFRW {
00014
00015     typedef Eigen::VectorXd Vector;
00016     typedef Vector State;
00017     typedef WorkspaceGeneric<State> Workspace;
00018
00019     Workspace &workspace;
00020     double t_i;
00021     Vector phi_ffts;
00022
00023     WKBSolutionForKleinGordonEquationInFRW(Workspace &workspace_, const double t_i_);
00024
00025     Vector evaluate_at(const double t);
00026
00027 };
00028
00029 #endif

```

9.37 src/workspace.hpp File Reference

A generic "workspace" class, containing parameters / data / tools used during simulations.

```
#include <memory>
#include "param.hpp"
#include "physics.hpp"
#include "fftw_wrapper.hpp"
#include "dispatcher.hpp"
```

Classes

- struct [WorkspaceGeneric< Vector >](#)
A generic workspace for storing temporary.

Concepts

- concept [HasLatticeParams](#)
- concept [HasMass](#)
- concept [HasLambda](#)
- concept [HasFa](#)
- concept [HasFRWParameters](#)
- concept [HasPsiApproximationParameters](#)

Macros

- `#define TYPE_REQUIREMENT(value, type) {std::remove_cvref_t<decltype((value))>()}-> std::same_as<type>;`

9.37.1 Detailed Description

A generic "workspace" class, containing parameters / data / tools used during simulations.

Author

Siyang Ling

9.38 workspace.hpp

[Go to the documentation of this file.](#)

```

00001
00006 #ifndef WORKSPACE_HPP
00007 #define WORKSPACE_HPP
00008
00009 #include <memory>
00010
00011 #include "param.hpp"
00012 #include "physics.hpp"
00013 #include "fftw_wrapper.hpp"
00014 #include "dispatcher.hpp"
00015
00016 #define TYPE_REQUIREMENT(value, type) {std::remove_cvref_t<decltype((value))>()>() ->
    std::same_as<type>;
00017
00018
00019
00020 template<typename Param>
00021 concept HasLatticeParams = requires (Param param)
00022 { TYPE_REQUIREMENT(param.N, long long int)
00023   TYPE_REQUIREMENT(param.L, double) };
00024
00025 template<typename Param>
00026 concept HasMass = requires (Param param) { TYPE_REQUIREMENT(param.m, double) };
00027
00028 template<typename Param>
00029 concept HasLambda = requires (Param param) { TYPE_REQUIREMENT(param.lambda, double) };
00030
00031 template<typename Param>
00032 concept HasFa = requires (Param param) { TYPE_REQUIREMENT(param.f_a, double) };
00033
00034 template<typename Param>
00035 concept HasFRWParameters = requires (Param param)
00036 { TYPE_REQUIREMENT(param.a1, double)
00037   TYPE_REQUIREMENT(param.H1, double)
00038   TYPE_REQUIREMENT(param.t1, double) };
00039
00040 template<typename Param>
00041 concept HasPsiApproximationParameters = requires (Param param)
00042 { TYPE_REQUIREMENT(param.M, long long int) };
00043
00044
00053 template<typename Vector>
00054 struct WorkspaceGeneric {
00055     typedef Vector State;
00056     long long int N;
00057     double L;
00058     double m;
00059     StaticEOSCosmology cosmology{};
00060     State state;
00061     double lambda{0};
00062     double f_a{1.0};
00063     Vector Psi;
00064     Vector dPsidt;
00065     Vector Psi_fft;
00066     Vector dPsidt_fft;
00067     Vector R_fft;
00068     std::vector<double> t_list;
00069     typename fftWrapperDispatcher<Vector>::Generic fft_wrapper;
00070
00071     bool Psi_approximation_initialized{false};
00072     long long int M;
00073     std::unique_ptr<typename fftWrapperDispatcher<Vector>::Generic> fft_wrapper_M_ptr;
00074     Vector cutoff_R_fft;
00075
00076     template<HasLatticeParams Param>
00077     WorkspaceGeneric(const Param &param, auto &initializer) :
00078         N(param.N), L(param.L), fft_wrapper(param.N)
00079     {
00080         //static_assert(HasLatticeParams<Param>, "HasLatticeParams<Param> test failed.");
00081         if constexpr(HasFRWParameters<Param>) { cosmology = StaticEOSCosmology(param); }
00082         if constexpr(HasMass<Param>) { m = param.m; }
00083         if constexpr(HasLambda<Param>) { lambda = param.lambda; }
00084         if constexpr(HasFa<Param>) { f_a = param.f_a; }
00085         if constexpr(HasPsiApproximationParameters<Param>) { M = param.M;
00086             assert(N >= M); }
00087         initializer(param, *this);
00088     }
00089 };
00090
00091
00092
00093
00094 #endif

```


Index

a1

SampleParam, [37](#)
StaticEOSCosmology, [39](#)

broken_power_law_given_amplitude_3d
random_field.hpp, [76](#)

ComovingCurvatureEquationInFRW, [19](#)
compute_energy_density, [20](#)
compute_cutoff_fouriers
fdm3d.hpp, [52](#)
compute_energy_density
ComovingCurvatureEquationInFRW, [20](#)
KleinGordonEquation, [33](#)
KleinGordonEquationInFRW, [35](#)

compute_field_with_scaled_fourier_modes
fdm3d.hpp, [52](#)

compute_inverse_laplacian
fdm3d.hpp, [53](#)

compute_mode_power_spectrum
fdm3d.hpp, [53](#)

compute_power_spectrum
fdm3d.hpp, [55](#)

ConstIntervalObserver< Equation, save_field_spectrum,
save_density_spectrum, save_density >, [20](#)
t_end, [22](#)
t_interval, [22](#)
t_start, [22](#)

CudaApproximateComovingCurvatureEquationInFRW,
[22](#)

CudaComovingCurvatureEquationInFRW, [23](#)

CudaFixedCurvatureEquationInFRW, [24](#)

CudaKleinGordonEquationInFRW, [24](#)

CudaLambdaEquationInFRW, [25](#)

CudaSqrtPotentialEquationInFRW, [25](#)

cufftWrapper, [26](#)

cufftWrapperBatchedD2Z, [27](#)

cufftWrapperD2Z, [27](#)

cufftWrapperNoBatching, [27](#)

empty, [28](#)

eta

StaticEOSCosmology, [39](#)

execute_batched_d2z

fftwWrapper, [30](#)

execute_d2z

fftwWrapper, [31](#)

execute_inplace_z2d

fftwWrapper, [31](#)

execute_z2d

fftwWrapper, [31](#), [32](#)

fdm3d.hpp

compute_cutoff_fouriers, [52](#)

compute_field_with_scaled_fourier_modes, [52](#)

compute_inverse_laplacian, [53](#)

compute_mode_power_spectrum, [53](#)

compute_power_spectrum, [55](#)

PADDED_IDX_OF, [52](#)

fftWrapperDispatcher< Eigen::VectorXd >, [29](#)

fftWrapperDispatcher< thrust::device_vector< double
> >, [29](#)

fftWrapperDispatcher< Vector >, [29](#)

fftwWrapper, [30](#)

execute_batched_d2z, [30](#)

execute_d2z, [31](#)

execute_inplace_z2d, [31](#)

execute_z2d, [31](#), [32](#)

fftwWrapper, [30](#)

generate_inhomogeneous_gaussian_random_field
random_field.hpp, [77](#)

H1

SampleParam, [37](#)

StaticEOSCosmology, [39](#)

HasFa, [17](#)

HasFRWParameters, [17](#)

HasLambda, [17](#)

HasLatticeParams, [17](#)

HasMass, [18](#)

HasPsiApproximationParameters, [18](#)

homogeneous_field

initializer.hpp, [60](#)

homogeneous_field_with_fluctuations

initializer.hpp, [60](#)

Implementing your own equation, [5](#)

initializer.hpp

homogeneous_field, [60](#)

homogeneous_field_with_fluctuations, [60](#)

perturbed_grf, [60](#)

perturbed_grf_and_comoving_curvature_fft, [61](#)

perturbed_grf_without_saving_Psi, [61](#)

plane_wave, [61](#)

unperturbed_grf, [62](#)

unperturbed_grf_and_fixed_curvature, [62](#)

unperturbed_grf_with_background, [62](#)

unperturbed_grf_with_Psi, [63](#)

wave_packet, [63](#)

- k_ast
 - SampleParam, [37](#)
- KGParam, [32](#)
- KleinGordonEquation, [33](#)
 - compute_energy_density, [33](#)
 - operator(), [34](#)
- KleinGordonEquationInFRW, [34](#)
 - compute_energy_density, [35](#)
- L
 - SampleParam, [37](#)
- lambda
 - SampleParam, [37](#)
- LatticeEquationConcept, [18](#)
- lite-cosmic-sim, [1](#)
- m
 - SampleParam, [38](#)
- midpoint< State, Value, Deriv, Time, Algebra, Operations, Resizer >, [35](#)
- MyParam, [36](#)
- N
 - SampleParam, [38](#)
- operator()
 - KleinGordonEquation, [34](#)
- p
 - StaticEOSCosmology, [40](#)
- PADDED_IDX_OF
 - fdm3d.hpp, [52](#)
- perturbed_grf
 - initializer.hpp, [60](#)
- perturbed_grf_and_comoving_curvature_fft
 - initializer.hpp, [61](#)
- perturbed_grf_without_saving_Psi
 - initializer.hpp, [61](#)
- plane_wave
 - initializer.hpp, [61](#)
- power_law_with_cutoff_given_amplitude_3d
 - random_field.hpp, [77](#)
- random_field.hpp
 - broken_power_law_given_amplitude_3d, [76](#)
 - generate_inhomogeneous_gaussian_random_field, [77](#)
 - power_law_with_cutoff_given_amplitude_3d, [77](#)
 - scale_invariant_spectrum_3d, [78](#)
- SampleParam, [37](#)
 - a1, [37](#)
 - H1, [37](#)
 - k_ast, [37](#)
 - L, [37](#)
 - lambda, [37](#)
 - m, [38](#)
 - N, [38](#)
 - t1, [38](#)
 - varphi_std_dev, [38](#)
- scale_invariant_spectrum_3d
 - random_field.hpp, [78](#)
- src/cuda_wrapper.cuh, [43](#), [44](#)
- src/dispatcher.hpp, [45](#), [46](#)
- src/eigen_wrapper.hpp, [46](#), [47](#)
- src/equations.hpp, [47](#), [48](#)
- src/equations_cuda.cuh, [49](#), [50](#)
- src/fdm3d.hpp, [51](#), [56](#)
- src/fdm3d_cuda.cuh, [56](#), [57](#)
- src/fftw_wrapper.hpp, [58](#)
- src/field_booster.hpp, [59](#)
- src/initializer.hpp, [59](#), [64](#)
- src/io.hpp, [67](#), [68](#)
- src/midpoint.hpp, [68](#), [69](#)
- src/observer.hpp, [70](#), [71](#)
- src/param.hpp, [72](#), [73](#)
- src/physics.hpp, [74](#)
- src/random_field.hpp, [75](#), [78](#)
- src/special_function.hpp, [79](#)
- src/utility.hpp, [79](#), [80](#)
- src/wkb.hpp, [81](#)
- src/workspace.hpp, [82](#), [83](#)
- StaticEOSCosmology, [38](#)
 - a1, [39](#)
 - eta, [39](#)
 - H1, [39](#)
 - p, [40](#)
 - t1, [40](#)
- t1
 - SampleParam, [38](#)
 - StaticEOSCosmology, [40](#)
- t_end
 - ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density >, [22](#)
- t_interval
 - ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density >, [22](#)
- t_start
 - ConstIntervalObserver< Equation, save_field_spectrum, save_density_spectrum, save_density >, [22](#)
- unperturbed_grf
 - initializer.hpp, [62](#)
- unperturbed_grf_and_fixed_curvature
 - initializer.hpp, [62](#)
- unperturbed_grf_with_background
 - initializer.hpp, [62](#)
- unperturbed_grf_with_Psi
 - initializer.hpp, [63](#)
- varphi_std_dev
 - SampleParam, [38](#)
- wave_packet
 - initializer.hpp, [63](#)
- WKBSolutionForKleinGordonEquationInFRW, [40](#)
- WorkspaceGeneric< Vector >, [41](#)