lite-cosmic-sim

Generated by Doxygen 1.10.0

Chapter 1

Introduction

- Basics This codebase provides a minimal and flexible framework for running field simulations, on both CPUs and GPUs. The original purpose of this codebase is for studying scalar fields in a gravitational potential Options include setting up initial conditions, solving several different types equations, choice of numerical methods, choice of output
- Examples
```
integrate_const(stepper, eqn, workspace.state, param.t_start, param.t_end, 0.5, observer);
```
- Test

Chapter 2

Writing your own equation

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

2.1 Adding a new parameter

You will need to define a new parameter struct that contains the new κ parameter.

2.2 Adding the equation class

```
struct KappaEquation;
```

2.2.1 The equation function

You need to define at least one function in the The `odeint` library

2.2.2 The energy density function

Chapter 3

Concept Index

3.1 Concepts

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

Chapter 4

Hierarchical Index

4.1 Class Hierarchy

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

Chapter 5

Class Index

5.1 Class List

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

Chapter 6

File Index

6.1 File List

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

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

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)

Public Attributes

- Workspace & **workspace**

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

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 &param, 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**

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)

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

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**

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

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**

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

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

8.13 empty Struct Reference

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

- src/dispatcher.hpp

8.14 fftWrapperDispatcher< Vector > Struct Template Reference

Public Types

- typedef empty **D2Z**
- typedef empty **BatchedD2Z**
- typedef empty **Generic**

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

Public Member Functions

- **fftwWrapper** (int N_)
- Eigen::VectorXd **execute_d2z** (Eigen::VectorXd &in)
- Eigen::VectorXd **execute_batched_d2z** (Eigen::VectorXd &in)
- Eigen::VectorXd **execute_z2d** (Eigen::VectorXd &in)
- void **execute_z2d** (Eigen::VectorXd &in, Eigen::VectorXd &out)
- void **execute_inplace_z2d** (Eigen::VectorXd &inout)
- **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**

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.

```
#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)

Static Public Member Functions

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

Public Attributes

- Workspace & **workspace**

8.19.1 Detailed Description

The Klein Gordon equation.

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

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

Parameters

`out`	*test*	The memory area to copy to.

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

- src/equations.hpp
- src/equations.cpp

8.20 KleinGordonEquationInFRW Struct Reference

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)

Public Attributes

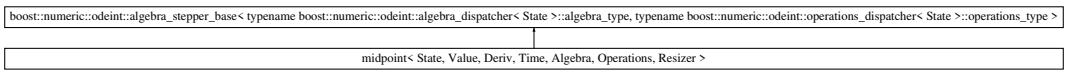
- Workspace & **workspace**

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

Public Member Functions

- **StaticEOSCosmology** (const double a1_, const double H1_, const double t1_, const double p_)
- template<typename T >
  StaticEOSCosmology (const T &param)
- double **a** (const double t) const
- double **H** (const double t) const
- double **eta** (const double t) const

Public Attributes

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

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

Public Types

- typedef Vector **State**

Public Member Functions

- template<HasLatticeParams Param>
  WorkspaceGeneric (const Param &param, 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**

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

- src/workspace.hpp

Chapter 9

File Documentation

9.1 cuda_wrapper.cuh

```
00001 #ifndef CUDA_WRAPPER_CUH
00002 #define CUDA_WRAPPER_CUH
00003
00004 #include <iostream>
00005
00006 #include <Eigen/Dense>
00007
00008 #include <thrust/device_vector.h>
00009 // #include <thrust/host_vector.h>
00010 // #include <thrust/execution_policy.h>
00011 // #include <thrust/reduce.h>
00012 // #include <thrust/functional.h>
00013 // #include <thrust/fill.h>
00014 // #include <thrust/transform.h>
00015
00016 #include "cufft.h"
00017 #include "cufftXt.h"
00018 #include <cuda_runtime.h>
00019
00020
00021
00022 typedef decltype(Eigen::VectorXd().begin()) eigen_iterator;
00023 typedef decltype(thrust::device_vector<double>().begin()) thrust_iterator;
00024 typedef thrust::detail::normal_iterator<thrust::device_ptr<const double» thrust_const_iterator;
00025 typedef Eigen::internal::pointer_based_stl_iterator<Eigen::Matrix<double, -1, 1» eigen_iterator_2;
00026
00027
00028 /*
00029   Explicit template instantiation declarations for the thrust library.
00030   They are declared here so that they are instantiatiated in cuda_wrapper.cu (and compiled with nvcc),
00031   and don't get instantiated in other translation units.
00032   This is necessary since we want to call thrust functions in translation units compiled by other
      compilers (g++ / icpx).
00033 */
00034 extern template class thrust::device_vector<double>;
00035 extern template class thrust::device_ptr<double>;
00036 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»);
00037 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);
00038
00039 extern template thrust_iterator thrust::copy(eigen_iterator, eigen_iterator, thrust_iterator);
00040 extern template eigen_iterator thrust::copy(thrust_iterator, thrust_iterator, eigen_iterator);
00041
00042 //Eigen::VectorXd copy_vector(const thrust::device_vector<double> &in);
00043 void copy_vector(Eigen::VectorXd &out, const thrust::device_vector<double> &in);
00044 //void copy_vector(Eigen::VectorXd &out, const Eigen::VectorXd &in);
00045
00046
00047 void show_gpu_memory_usage(void);
00048
00049 /*
00050   Wrapper for 3D cufftPlan3d.
00051   Performs double to complex double FFT for a N*N*N grid.
00052 */
00053 struct cufftWrapperD2Z {
```

```
00054   int N;
00055   cufftHandle plan;
00056   explicit cufftWrapperD2Z(int N_);
00057   ~cufftWrapperD2Z();
00058   thrust::device_vector<double> execute(thrust::device_vector<double> &in);
00059
00060   cufftWrapperD2Z(const cufftWrapperD2Z &) = delete;
00061   cufftWrapperD2Z &operator=(const cufftWrapperD2Z &) = delete;
00062   cufftWrapperD2Z(cufftWrapperD2Z &&) = delete;
00063   cufftWrapperD2Z &operator=(cufftWrapperD2Z &&) = delete;
00064 };
00065
00066
00067 /*
00068   Wrapper for 3D cufftPlanMany.
00069   Performs two double to complex double FFT for a N*N*N grid.
00070 */
00071 struct cufftWrapperBatchedD2Z {
00072   int N;
00073   cufftHandle plan;
00074   explicit cufftWrapperBatchedD2Z(int N_);
00075   ~cufftWrapperBatchedD2Z();
00076   thrust::device_vector<double> execute(thrust::device_vector<double> &in);
00077
00078   cufftWrapperBatchedD2Z(const cufftWrapperBatchedD2Z &) = delete;
00079   cufftWrapperBatchedD2Z &operator=(const cufftWrapperBatchedD2Z &) = delete;
00080   cufftWrapperBatchedD2Z(cufftWrapperBatchedD2Z &&) = delete;
00081   cufftWrapperBatchedD2Z &operator=(cufftWrapperBatchedD2Z &&) = delete;
00082 };
00083
00084 /*
00085   Wrapper for various cufft functions for a N*N*N grid.
00086   Different cufft plans share the same work area so that GPU memory usage is minimized.
00087 */
00088 struct cufftWrapper {
00089   int N;
00090   cufftHandle plan_d2z;
00091   cufftHandle plan_batched_d2z;
00092   cufftHandle plan_z2d;
00093   thrust::device_vector<double> work_area;
00094   explicit cufftWrapper(int N_);
00095   ~cufftWrapper();
00096
00097   thrust::device_vector<double> execute_d2z(thrust::device_vector<double> &in);
00098   thrust::device_vector<double> execute_batched_d2z(thrust::device_vector<double> &in);
00099   thrust::device_vector<double> execute_z2d(thrust::device_vector<double> &in);
00100
00101   cufftWrapper(const cufftWrapper &) = delete;
00102   cufftWrapper &operator=(const cufftWrapper &) = delete;
00103   cufftWrapper(cufftWrapper &&) = delete;
00104   cufftWrapper &operator=(cufftWrapper &&) = delete;
00105 };
00106
00107 /*
00108   Wrapper for various cufft functions for a N*N*N grid.
00109   Different cufft plans share the same work area so that GPU memory usage is minimized.
00110 */
00111 struct cufftWrapperNoBatching {
00112   int N;
00113   cufftHandle plan_d2z;
00114   cufftHandle plan_z2d;
00115   thrust::device_vector<double> work_area;
00116   explicit cufftWrapperNoBatching(int N_);
00117   ~cufftWrapperNoBatching();
00118
00119   thrust::device_vector<double> execute_d2z(thrust::device_vector<double> &in);
00120   thrust::device_vector<double> execute_batched_d2z(thrust::device_vector<double> &in);
00121   thrust::device_vector<double> execute_z2d(thrust::device_vector<double> &in);
00122   void execute_inplace_z2d(thrust::device_vector<double> &inout);
00123
00124   cufftWrapperNoBatching(const cufftWrapperNoBatching &) = delete;
00125   cufftWrapperNoBatching &operator=(const cufftWrapperNoBatching &) = delete;
00126   cufftWrapperNoBatching(cufftWrapperNoBatching &&) = delete;
00127   cufftWrapperNoBatching &operator=(cufftWrapperNoBatching &&) = delete;
00128 };
00129
00130
00131 #endif
```

9.2 dispatcher.hpp

```
00001 #ifndef DISPATCHER_HPP
00002 #define DISPATCHER_HPP
```

```
00003
00004 #include "fftw_wrapper.hpp"
00005
00006 #ifndef DISABLE_CUDA
00007 #include <thrust/device_vector.h>
00008 #include "cuda_wrapper.cuh"
00009 #define ALGORITHM_NAMESPACE thrust
00010 #else
00011 #define ALGORITHM_NAMESPACE std
00012 #endif
00013
00014
00015 // An empty placeholder object
00016 struct empty {};
00017
00018 // Dispatcher for fftWrapper* types
00019 template<typename Vector>
00020 struct fftWrapperDispatcher {
00021   typedef empty D2Z;
00022   typedef empty BatchedD2Z;
00023   typedef empty Generic;
00024 };
00025
00026 #ifndef DISABLE_CUDA
00027 template<>
00028 struct fftWrapperDispatcher<thrust::device_vector<double» {
00029   typedef cufftWrapperD2Z D2Z;
00030   typedef cufftWrapperBatchedD2Z BatchedD2Z;
00031   //typedef cufftWrapper Generic;
00032   typedef cufftWrapperNoBatching Generic;
00033 };
00034 #endif
00035
00036 template<>
00037 struct fftWrapperDispatcher<Eigen::VectorXd> {
00038   typedef empty D2Z;
00039   typedef empty BatchedD2Z;
00040   typedef fftwWrapper Generic;
00041 };
00042
00043
00044 #endif
```

9.3 eigen_wrapper.hpp

```
00001 #ifndef EIGEN_WRAPPER_HPP
00002 #define EIGEN_WRAPPER_HPP
00003
00004 #include <Eigen/Dense>
00005
00006 void copy_vector(Eigen::VectorXd &out, const Eigen::VectorXd &in);
00007
00008 #endif
```

9.4 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.
- struct KleinGordonEquationInFRW
- struct ComovingCurvatureEquationInFRW

Concepts

- concept LatticeEquationConcept

9.4.1 Detailed Description

Header for field equations that runs on the CPU.

This is the header for field equations that are supposed to run on CPU. Currently only the Klein Gordon equation $\ddot{\varphi} - \nabla^2\varphi + m^2\varphi = 0$ and the FRW Klein Gordon equation $\ddot{\varphi} + 3H\dot{\varphi} - \nabla^2\varphi/a^2 + m^2\varphi = 0$ are implemented.

Each equation struct implements both **operator()** and **compute_energy_density(const Workspace &, const double)**, which computes and save the time derivative and computes the energy density of the equation.

9.5 equations.hpp

Go to the documentation of this file.

```
00001
00013 #ifndef EQUATIONS_H
00014 #define EQUATIONS_H
00015
00016
00017 #include "Eigen/Dense"
00018
00019 #include <boost/numeric/odeint.hpp>
00020 #include <boost/numeric/odeint/external/eigen/eigen.hpp>
00021
00022 #include "odeint_eigen/eigen_operations.hpp"
00023
00024 #include "workspace.hpp"
00025
00026 template<typename Equation>
00027 concept LatticeEquationConcept = requires (Equation eqn)
00028   {
00029    //typename Equation::State;
00030    eqn.workspace;
00031    eqn.compute_energy_density(eqn.workspace, 0.0);
00032   };
00033
00034
00047 struct KleinGordonEquation {
00048   typedef Eigen::VectorXd Vector;
00049   typedef Vector State;
00050   typedef WorkspaceGeneric<State> Workspace;
00051   Workspace &workspace;
00052
00053   KleinGordonEquation(Workspace &workspace_) : workspace(workspace_) {}
00054
00055   void operator()(const State &, State &, const double);
00056
00057   static Vector compute_energy_density(const Workspace &workspace, const double t);
00058 };
00059
00060
00061 struct KleinGordonEquationInFRW {
00062   typedef Eigen::VectorXd Vector;
00063   typedef Vector State;
00064   typedef WorkspaceGeneric<State> Workspace;
00065   Workspace &workspace;
00066
00067   KleinGordonEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00068
00069   void operator()(const State &, State &, const double);
00070
00071   static Vector compute_energy_density(const Workspace &workspace, const double t);
00072 };
00073
00074
00075 struct ComovingCurvatureEquationInFRW {
00076   typedef Eigen::VectorXd Vector;
00077   typedef Vector State;
00078   typedef WorkspaceGeneric<State> Workspace;
```

```
00079   Workspace &workspace;
00080
00081   ComovingCurvatureEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00082
00083   void operator()(const State &, State &, const double);
00084
00085   static Vector compute_energy_density(Workspace &workspace, const double t);
00086 };
00087
00088
00089
00090
00091 #endif
```

9.6 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

Functions

- void **kernel_test** (const thrust::device_vector< double > &R_fft, thrust::device_vector< double > &Psi, thrust::device_vector< double > &dPsidt, const long long int N, const double L, const double m, const double a_t, const double H_t, const double eta_t, const double inv_ah_sqr, const double t, fftWrapperDispatcher< thrust::device_vector< double > >::Generic &fft_wrapper)

9.6.1 Detailed Description

Header for field equations that runs on the GPU.

9.7 equations_cuda.cuh

Go to the documentation of this file.

```
00001
00005 #ifndef EQUATIONS_CUDA_CUH
00006 #define EQUATIONS_CUDA_CUH
00007
00008 #include "equations.hpp"
00009
00010 #include <thrust/device_vector.h>
00011
00012 #include "odeint_thrust/thrust.hpp"
00013
00014 struct CudaKleinGordonEquationInFRW {
00015   typedef thrust::device_vector<double> Vector;
00016   typedef Vector State;
00017   typedef WorkspaceGeneric<Vector> Workspace;
00018   Workspace &workspace;
00019
00020   CudaKleinGordonEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00021
00022   void operator()(const State &, State &, const double);
00023
00024   static Vector compute_energy_density(const Workspace &workspace, const double t);
00025   static Vector compute_dot_energy_density(const Workspace &workspace, const double t);
00026 };
00027
00028
00029 struct CudaLambdaEquationInFRW {
00030   typedef thrust::device_vector<double> Vector;
00031   typedef Vector State;
00032   typedef WorkspaceGeneric<Vector> Workspace;
00033   Workspace &workspace;
00034
00035   CudaLambdaEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00036
00037   void operator()(const State &, State &, const double);
00038
00039   static Vector compute_energy_density(const Workspace &workspace, const double t);
00040 };
00041
00042
00043 struct CudaSqrtPotentialEquationInFRW {
00044   typedef thrust::device_vector<double> Vector;
00045   typedef Vector State;
00046   typedef WorkspaceGeneric<Vector> Workspace;
00047   Workspace &workspace;
00048
00049   CudaSqrtPotentialEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00050
00051   void operator()(const State &, State &, const double);
00052
00053   static Vector compute_energy_density(const Workspace &workspace, const double t);
00054 };
00055
00056
00057 struct CudaFixedCurvatureEquationInFRW {
00058   typedef thrust::device_vector<double> Vector;
00059   typedef Vector State;
00060   typedef WorkspaceGeneric<Vector> Workspace;
00061   Workspace &workspace;
00062
00063   CudaFixedCurvatureEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00064
00065   void operator()(const State &, State &, const double);
00066
00067   static Vector compute_energy_density(const Workspace &workspace, const double t);
00068 };
00069
00070
00071 struct CudaComovingCurvatureEquationInFRW {
00072   typedef thrust::device_vector<double> Vector;
00073   typedef Vector State;
00074   typedef WorkspaceGeneric<Vector> Workspace;
00075   Workspace &workspace;
00076
00077   CudaComovingCurvatureEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00078
00079   void operator()(const State &, State &, const double);
00080
00081   static Vector compute_energy_density(Workspace &workspace, const double t);
00082 };
00083
00084
00085 struct CudaApproximateComovingCurvatureEquationInFRW {
```

```
00086   typedef thrust::device_vector<double> Vector;
00087   typedef Vector State;
00088   typedef WorkspaceGeneric<Vector> Workspace;
00089   Workspace &workspace;
00090
00091   CudaApproximateComovingCurvatureEquationInFRW(Workspace &workspace_) : workspace(workspace_) {}
00092
00093   void operator()(const State &, State &, const double);
00094
00095   static Vector compute_energy_density(Workspace &workspace, const double t);
00096 };
00097
00098
00099 // Explicit template instantiation declaration for the thrust library.
00100 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>);
00101
00102 // Deprecated function for testing CUDA kernels.
00103 /*
00104 void compute_deriv_test(const Eigen::VectorXd &in, Eigen::VectorXd &out,
00105             const double m, const double lambda,
00106             const double a_t, const double H_t, const double inv_ah_sqr,
00107             const long long int N);
00108 */
00109 void kernel_test(const thrust::device_vector<double> &R_fft, thrust::device_vector<double> &Psi,
      thrust::device_vector<double> &dPsidt,
00110          const long long int N, const double L, const double m,
00111          const double a_t, const double H_t, const double eta_t, const double inv_ah_sqr,
00112          const double t, fftWrapperDispatcher<thrust::device_vector<double>::Generic &fft_wrapper);
00113
00114 #endif
```

9.8 src/fdm3d.hpp File Reference

Commonly used 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 linear index of a point on lattice.
- #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)
- Eigen::VectorXd **compute_mode_power_spectrum** (const long long int N, const double L, const double m, Eigen::VectorXd &state, fftWrapperDispatcher< Eigen::VectorXd >::Generic &fft_wrapper)
- Eigen::VectorXd **compute_inverse_laplacian** (const long long int N, const double L, Eigen::VectorXd &f, fftWrapperDispatcher< Eigen::VectorXd >::Generic &fft_wrapper)
- 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)
- Eigen::VectorXd **compute_cutoff_fouriers** (const long long int N, const long long int M, Eigen::VectorXd &fft)
- Eigen::VectorXd **compute_power_spectrum** (const long long int N, Eigen::VectorXd &phi)
- Eigen::VectorXd **compute_fourier** (const long long int N, const double L, Eigen::VectorXd &phi)
- Eigen::VectorXd **compute_laplacian** (const long long int N, const double L, const Eigen::VectorXd &f)

9.8.1 Detailed Description

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

9.8.2 Macro Definition Documentation

9.8.2.1 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.

9.9 fdm3d.hpp

Go to the documentation of this file.

```
00001
00005 #ifndef FDM3D_HPP
00006 #define FDM3D_HPP
00007
00008 #include "Eigen/Dense"
00009 #include "dispatcher.hpp"
00010
00011
00015 #define IDX_OF(N, i, j, k) ((N)*(N)*(i) + (N)*(j) + (k))
00016
00022 #define PADDED_IDX_OF(N, i, j, k) ((N)*2*((N)/2+1)*(i) + 2*((N)/2+1)*(j) + (k))
00023
00024
00025 Eigen::VectorXd compute_power_spectrum(const long long int N,
00026                       Eigen::VectorXd &f,
00027                       fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00028
00029 Eigen::VectorXd compute_mode_power_spectrum(const long long int N, const double L, const double m,
00030                        Eigen::VectorXd &state,
00031                        fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00032
00033 Eigen::VectorXd compute_inverse_laplacian(const long long int N, const double L,
00034                      Eigen::VectorXd &f,
00035                      fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00036
00037 Eigen::VectorXd compute_field_with_scaled_fourier_modes(const long long int N, const double L,
00038                             Eigen::VectorXd &f,
00039                             std::function<double(const double)> kernel,
00040                             fftWrapperDispatcher<Eigen::VectorXd>::Generic &fft_wrapper);
00041
00042 Eigen::VectorXd compute_cutoff_fouriers(const long long int N, const long long int M,
00043                     Eigen::VectorXd &fft);
00044
00045
00046 // Deprecated
00047 Eigen::VectorXd compute_power_spectrum(const long long int N, Eigen::VectorXd &phi);
00048 // Eigen::VectorXd compute_gradient_squared(const long long int N, const double L, const
      Eigen::VectorXd &phi);
00049 Eigen::VectorXd compute_fourier(const long long int N, const double L, Eigen::VectorXd &phi);
00050 Eigen::VectorXd compute_laplacian(const long long int N, const double L, const Eigen::VectorXd &f);
00051
00052
00053
00054 #endif
```

9.10 fdm3d_cuda.cuh

```
00001 #ifndef FDM3D_CUDA_CUH
00002 #define FDM3D_CUDA_CUH
00003
00004 #include <thrust/device_vector.h>
00005 #include <thrust/reduce.h>
00006 #include <thrust/functional.h>
00007
00008 //#include "odeint_thrust/thrust.hpp"
00009 #include "cuda_wrapper.cuh"
00010 #include "dispatcher.hpp"
00011
00012 #include "fdm3d.hpp"
00013
00014 thrust::device_vector<double> compute_mode_power_spectrum(const long long int N, const double L, const
      double m,
00015                                thrust::device_vector<double> &state,
00016                                fftWrapperDispatcher<thrust::device_vector<double»::Generic
      &fft_wrapper);
00017
00018 thrust::device_vector<double> compute_power_spectrum(const long long int N,
00019                               thrust::device_vector<double> &f,
00020                               fftWrapperDispatcher<thrust::device_vector<double»::Generic
      &fft_wrapper);
00021
00022 thrust::device_vector<double> compute_laplacian(const long long int N, const double L,
00023                          thrust::device_vector<double> &f);
00024
00025 thrust::device_vector<double> compute_inverse_laplacian(const long long int N, const double L,
00026                             thrust::device_vector<double> &f,
00027                             fftWrapperDispatcher<thrust::device_vector<double»::Generic &fft_wrapper);
00028
00029 thrust::device_vector<double> compute_cutoff_fouriers(const long long int N, const long long int M,
00030                               const thrust::device_vector<double> &fft);
00031
00032 void compute_inverse_laplacian_test(const long long int N, const double L,
00033                     thrust::device_vector<double> &fft);
00034 #endif
```

9.11 fftw_wrapper.hpp

```
00001 #ifndef FFTW_WRAPPER_HPP
00002 #define FFTW_WRAPPER_HPP
00003
00004 #include <iostream>
00005
00006 #include <Eigen/Dense>
00007 #include <fftw3.h>
00008
00009
00010 /*
00011   Wrapper for various fftw functions for a N*N*N grid.
00012 */
00013 struct fftwWrapper {
00014   int N;
00015   fftw_plan plan_d2z;
00016   fftw_plan plan_z2d;
00017   fftw_plan plan_inplace_z2d;
00018   explicit fftwWrapper(int N_);
00019   ~fftwWrapper();
00020
00021   Eigen::VectorXd execute_d2z(Eigen::VectorXd &in);
00022   Eigen::VectorXd execute_batched_d2z(Eigen::VectorXd &in);
00023   Eigen::VectorXd execute_z2d(Eigen::VectorXd &in);
00024   void execute_z2d(Eigen::VectorXd &in, Eigen::VectorXd &out);
00025   void execute_inplace_z2d(Eigen::VectorXd &inout);
00026
00027   fftwWrapper(const fftwWrapper &) = delete;
00028   fftwWrapper &operator=(const fftwWrapper &) = delete;
00029   fftwWrapper(fftwWrapper &&) = delete;
00030   fftwWrapper &operator=(fftwWrapper &&) = delete;
00031 };
00032
00033
00034 #endif
```

9.12 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
```

9.13 src/initializer.hpp File Reference

Snippets for initializing the workspace for simulation. (e.g. Setting up field realizations, gravitational potentials, comoving curvature perturbations, 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
- auto unperturbed_grf_with_background
- auto perturbed_grf
- auto perturbed_grf_without_saving_Psi
- auto unperturbed_grf_with_Psi
- auto unperturbed_grf_and_fixed_curvature
- auto **perturbed_grf_and_comoving_curvature_fft**
- auto homogeneous_field
- auto homogeneous_field_with_fluctuations
- auto plane_wave
- auto wave_packet

9.13.1 Detailed Description

Snippets for initializing the workspace for simulation. (e.g. Setting up field realizations, gravitational potentials, comoving curvature perturbations, etc.)

9.13.2 Variable Documentation

9.13.2.1 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());
  }
```

9.13.2.2 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());
  }
```

9.13.2.3 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());


  }
```

9.13.2.4 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());
  }
```

9.13.2.5 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());
  }
```

9.13.2.6 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());
  }
```

9.13.2.7 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());
  }
```

9.13.2.8 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());
  }
```

9.13.2.9 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());
  }
```

9.13.2.10 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());
  }
```

9.14 initializer.hpp

Go to the documentation of this file.

```
00001
00006 #ifndef INITIALIZER_HPP
00007 #define INITIALIZER_HPP
00008
00009 #include "fdm3d.hpp"
00010 #include "random_field.hpp"
00011 #include "fftw_wrapper.hpp"
00012 #include "special_function.hpp"
00013 //#include "dispatcher.hpp"
00014 //#include "field_booster.hpp"
00015 //#include "param.hpp"
00016 //#include "physics.hpp"
00017
00018 #ifndef DISABLE_CUDA
00019 #include <thrust/device_vector.h>
00020 #include "cuda_wrapper.cuh"
00021 #define ALGORITHM_NAMESPACE thrust
00022 #else
00023 #define ALGORITHM_NAMESPACE std
00024 #endif
00025
00026 // Initialize a field and its derivative from a white noise power spectrum with cutoff k_ast
00027 inline auto unperturbed_grf =
00028   [](const auto param, auto &workspace) {
00029     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
      param.k_ast, 0);
00030     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00031     Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f); // Initial ULDM
      field
00032     Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf); // Initial
      ULDM field time derivative
00033
00034     auto &state = workspace.state;
00035     state.resize(varphi.size() + dt_varphi.size());
00036     // thrust::copy handles both copies between Eigen::VectorXd and copies from Eigen::VectorXd to
      thrust::device_vector<double>
00037     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00038     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00039   };
00040
00041
```

```
00042 // Initialize a field and its derivative from a white noise power spectrum with cutoff k_ast
00043 inline auto unperturbed_grf_with_background =
00044   [](const auto param, auto &workspace) {
00045     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
      param.k_ast, 0);
00046     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00047     Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
00048     varphi.array() += param.varphi_mean;
00049     Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);
00050
00051     auto &state = workspace.state;
00052     state.resize(varphi.size() + dt_varphi.size());
00053     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00054     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00055   };
00056
00057
00058 /*
00059   Initialize a field and its derivative from a white noise power spectrum with cutoff k_ast,
00060   but with a large scale perturbation specified by Psi.
00061   Psi is initialized from a scale-invariant power spectrum with cutoff k_Psi.
00062 */
00063 inline auto perturbed_grf =
00064   [](const auto param, auto &workspace) {
00065     Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
      param.k_Psi, -3);
00066     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
      param.k_ast, 0);
00067     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00068     Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
00069     Eigen::VectorXd varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi, P_f);
00070     Eigen::VectorXd dt_varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi,
      P_dtf);
00071
00072     auto &state = workspace.state;
00073     state.resize(varphi.size() + dt_varphi.size());
00074     workspace.Psi.resize(Psi.size());
00075     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00076     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00077     ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
00078
00079     //std::cout « boost::typeindex::type_id_runtime(workspace.Psi).pretty_name() « '\n';
00080   };
00081
00082
00083 inline auto perturbed_grf_without_saving_Psi =
00084   [](const auto param, auto &workspace) {
00085     Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
      param.k_Psi, -3);
00086     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
      param.k_ast, 0);
00087     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00088     Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
00089     Eigen::VectorXd varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi, P_f);
00090     Eigen::VectorXd dt_varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L, Psi,
      P_dtf);
00091
00092     auto &state = workspace.state;
00093     state.resize(varphi.size() + dt_varphi.size());
00094     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00095     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00096   };
00097
00098
00099 inline auto unperturbed_grf_with_Psi =
00100   [](const auto param, auto &workspace) {
00101     Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
      param.k_Psi, -3);
00102     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
      param.k_ast, 0);
00103     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00104     Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
00105     Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
00106     Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);
00107
00108     auto &state = workspace.state;
00109     state.resize(varphi.size() + dt_varphi.size());
00110     workspace.Psi.resize(Psi.size());
00111     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00112     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00113     ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
00114   };
00115
00116
00117 // Initialize a homogeneous Gaussian random field and some scale invariant curvature perturbation.
00118 inline auto unperturbed_grf_and_fixed_curvature =
00119   [](const auto param, auto &workspace) {
```

```
00120     Spectrum P_Psi = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.Psi_std_dev,
      param.k_Psi, -3);
00121     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
      param.k_ast, 0);
00122     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00123     Eigen::VectorXd Psi = generate_gaussian_random_field(param.N, param.L, P_Psi);
00124     Eigen::VectorXd varphi = generate_gaussian_random_field(param.N, param.L, P_f);
00125     Eigen::VectorXd dt_varphi = generate_gaussian_random_field(param.N, param.L, P_dtf);
00126
00127     auto &state = workspace.state;
00128     state.resize(varphi.size() + dt_varphi.size());
00129     workspace.Psi.resize(Psi.size());
00130     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00131     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00132     ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
00133   };
00134
00135 // Initialize an inhomogeneous Gaussian random field and the fft of some scale invariant comoving
      curvature perturbation.
00136 inline auto perturbed_grf_and_comoving_curvature_fft =
00137   [](const auto param, auto &workspace) {
00138     using namespace std::numbers;
00139
00140     // Generate comoving curvature perturbation
00141     double eta_i = workspace.cosmology.eta(param.t_start);
00142     double A_s = (-576 * pow(pi, 6) * pow(eta_i, 6) * pow(param.Psi_std_dev, 2)) /
00143       (-81 * pow(param.L, 4) * (pow(param.L, 2) + 2 * pow(pi, 2) * pow(eta_i, 2)) +
00144        param.L *
00145        (81 * pow(param.L, 5) - 54 * pow(param.L, 3) * pow(pi, 2) * pow(eta_i, 2) +
00146     48 * param.L * pow(pi, 4) * pow(eta_i, 4)) *
00147        cos((4 * pi * eta_i) / (sqrt(3) * param.L)) +
00148        256 * pow(pi, 6) * pow(eta_i, 6) * Ci_pade_approximant_12_12((4 * pi * eta_i) / (sqrt(3) *
      param.L)) +
00149        4 * sqrt(3) * param.L * pi * eta_i *
00150        (27 * pow(param.L, 4) + 6 * pow(param.L, 2) * pow(pi, 2) * pow(eta_i, 2) -
00151     16 * pow(pi, 4) * pow(eta_i, 4)) *
00152        sin((4 * pi * eta_i) / (sqrt(3) * param.L)));
00153     Spectrum P_R = scale_invariant_spectrum_3d(param.N, param.L, A_s);
00154
00155     // Manual cutoff for P_R at around horizon. The effect of imposing this cutoff is negligible.
00156     // Spectrum P_R_with_cutoff = [P_R](double k){ return k <= 0.5 ? P_R(k) : 0.0; };
00157
00158     Eigen::VectorXd R = generate_gaussian_random_field(param.N, param.L, P_R);
00159     // std::cout « "A_s = " « A_s « '\n';
00160
00161     // Calculate initial gravitational potential Psi.
00162     // Convention for potentials: \mathcal{R}_k = (3 / 2) \Psi_k for superhorizon.
00163     auto kernel = [eta_i](double k){
00164           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));
00165         };
00166     auto fft_wrapper = fftwWrapper(param.N);
00167     Eigen::VectorXd Psi = compute_field_with_scaled_fourier_modes(param.N, param.L, R, kernel,
      fft_wrapper);
00168
00169     // Calculate \varphi^2, \dot{\varphi}^2 perturbations as a multiple of Psi.
00170     // See Eqn (3.17) of paper.
00171     // 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.
00172     double v = param.k_ast / (param.a1 * param.m);
00173     double alpha_varphi_sqr = 0.5 * (- 3 * pow(4*pow(v,2)+5, 2)) / (12*pow(v,4) + 50*pow(v,2) + 50);
00174     double alpha_dot_varphi_sqr = 0.5 * (25 - 20*pow(v,2)) / (12*pow(v,4) + 50*pow(v,2) + 50);
00175
00176     Spectrum P_f = power_law_with_cutoff_given_amplitude_3d(param.N, param.L, param.varphi_std_dev,
      param.k_ast, 0);
00177     Spectrum P_dtf = to_deriv_spectrum(param.m, P_f);
00178     Eigen::VectorXd varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L,
      alpha_varphi_sqr * Psi, P_f);
00179     Eigen::VectorXd dt_varphi = generate_inhomogeneous_gaussian_random_field(param.N, param.L,
      alpha_dot_varphi_sqr * Psi, P_dtf);
00180
00181     auto &state = workspace.state;
00182     state.resize(varphi.size() + dt_varphi.size());
00183     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00184     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00185
00186     // Save the comoving curvature perturbation for reference
00187     {
00188       decltype(workspace.state) R_dvec(R.size());
00189       ALGORITHM_NAMESPACE::copy(R.begin(), R.end(), R_dvec.begin());
00190       workspace.R_fft = workspace.fft_wrapper.execute_d2z(R_dvec);
00191     }
00192   };
00193
00194
00195
00196 /*
```

```
00197   Initialize a homogeneous field with amplitude f and time derivative dt_f.
00198   For testing the numerical code.
00199 */
00200 inline auto homogeneous_field =
00201   [](const auto param, auto &workspace) {
00202     const long long int N = param.N;
00203     Eigen::VectorXd varphi = Eigen::VectorXd::Constant(N*N*N, param.f);
00204     Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, param.dt_f);
00205
00206     auto &state = workspace.state;
00207     state.resize(varphi.size() + dt_varphi.size());
00208     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00209     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00210   };
00211
00212
00213 /*
00214   Initialize a homogeneous field with amplitude f, plus scale-invariant perturbations (resembling
      quantum fluctutations).
00215 */
00216 inline auto homogeneous_field_with_fluctuations =
00217   [](const auto param, auto &workspace) {
00218     const long long int N = param.N;
00219     Eigen::VectorXd varphi = Eigen::VectorXd::Constant(N*N*N, param.f);
00220     Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, 0.0);
00221
00222     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);
00223     Eigen::VectorXd delta_varphi = generate_gaussian_random_field(param.N, param.L, P_delta_varphi);
00224     varphi += delta_varphi;
00225
00226     auto &state = workspace.state;
00227     state.resize(varphi.size() + dt_varphi.size());
00228     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00229     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00230   };
00231
00232
00233 /*
00234   Plane wave initial condition.
00235   For testing the numerical code.
00236 */
00237 inline auto plane_wave =
00238   [](const auto param, auto &workspace) {
00239     const long long int N = param.N;
00240     Eigen::VectorXd varphi(N*N*N);
00241     for(int a = 0; a < N; ++a){
00242       for(int b = 0; b < N; ++b){
00243     for(int c = 0; c < N; ++c){
00244       varphi(IDX_OF(N, a, b, c)) = cos(2 * std::numbers::pi * c / N);
00245     }
00246       }
00247     }
00248
00249     Eigen::VectorXd dt_varphi = Eigen::VectorXd::Constant(N*N*N, 0);
00250
00251     auto &state = workspace.state;
00252     state.resize(varphi.size() + dt_varphi.size());
00253     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00254     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00255   };
00256
00257
00258 /*
00259   Wave packet initial condition.
00260   For understanding evolution of a wave packet in gravitational potential well.
00261 */
00262 inline auto wave_packet =
00263   [](const auto param, auto &workspace) {
00264     const long long int N = param.N;
00265     Eigen::VectorXd varphi(N*N*N);
00266     Eigen::VectorXd dt_varphi(N*N*N);
00267     Eigen::VectorXd Psi(N*N*N);
00268
00269     for(int a = 0; a < N; ++a){
00270       for(int b = 0; b < N; ++b){
00271     for(int c = 0; c < N; ++c){
00272       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);
00273       varphi(IDX_OF(N, a, b, c)) = exp(- dist_to_center * dist_to_center / 40.0);
00274       dt_varphi(IDX_OF(N, a, b, c)) = 0;
00275       //Psi(IDX_OF(N, a, b, c)) = - param.Psi_std_dev * exp( - (b - N/2) * (b - N/2) / (2 * (param.L *
      param.L / 3.0 / 3.0)));
00276       Psi(IDX_OF(N, a, b, c)) = - param.Psi_std_dev * cos(2 * std::numbers::pi * c / N);
00277     }
00278       }
00279     }
```

```
00280
00281     auto &state = workspace.state;
00282     state.resize(varphi.size() + dt_varphi.size());
00283     ALGORITHM_NAMESPACE::copy(varphi.begin(), varphi.end(), state.begin());
00284     ALGORITHM_NAMESPACE::copy(dt_varphi.begin(), dt_varphi.end(), state.begin() + varphi.size());
00285
00286     workspace.Psi.resize(Psi.size());
00287     ALGORITHM_NAMESPACE::copy(Psi.begin(), Psi.end(), workspace.Psi.begin());
00288   };
00289
00290
00291 #endif
```

9.15 io.hpp

```
00001 #ifndef IO_HPP
00002 #define IO_HPP
00003 #include <cstdlib>
00004 #include <iostream>
00005 #include <fstream>
00006 #include <string>
00007 #include <vector>
00008 #include <iomanip>
00009
00010 #include <Eigen/Dense>
00011
00012
00013 std::vector<double> load_vector_from_file(std::string filename);
00014 void write_vector_to_file(std::vector<double> vector, std::string filename);
00015 void write_data_to_file(const char *buf, ssize_t size, std::string filename);
00016
00017 void write_VectorXd_to_file(const Eigen::VectorXd &vector, std::string filename);
00018 void write_VectorXd_to_filename_template(const Eigen::VectorXd &vector, const std::string
      format_string, const int idx);
00019 Eigen::VectorXd load_VectorXd_from_file(const std::string &filename);
00020
00021
00022 #endif
```

9.16 midpoint.hpp

```
00001 #ifndef MIDPOINT_HPP
00002 #define MIDPOINT_HPP
00003
00004
00005
00006 //#include <boost/numeric/odeint/stepper/base/explicit_stepper_base.hpp>
00007 //#include <boost/numeric/odeint/algebra/range_algebra.hpp>
00008 #include <boost/numeric/odeint/algebra/default_operations.hpp>
00009 #include <boost/numeric/odeint/algebra/algebra_dispatcher.hpp>
00010 #include <boost/numeric/odeint/algebra/operations_dispatcher.hpp>
00011
00012 #include <boost/numeric/odeint/util/state_wrapper.hpp>
00013 #include <boost/numeric/odeint/util/is_resizeable.hpp>
00014 #include <boost/numeric/odeint/util/resizer.hpp>
00015
00016 #ifndef DISABLE_CUDA
00017 #include "cuda_wrapper.cuh"
00018 #endif
00019
00020 template<
00021   class State,
00022   class Value = double,
00023   class Deriv = State,
00024   class Time = Value,
00025   class Algebra = typename boost::numeric::odeint::algebra_dispatcher<State>::algebra_type,
00026   class Operations = typename boost::numeric::odeint::operations_dispatcher<State>::operations_type,
00027   class Resizer = boost::numeric::odeint::initially_resizer //boost::numeric::odeint::always_resizer
00028   >
00029 class midpoint : public boost::numeric::odeint::algebra_stepper_base<Algebra, Operations>
00030 {
00031 public :
00032   typedef State state_type;
00033   typedef State deriv_type;
00034   typedef Value value_type;
00035   typedef Time time_type;
00036   typedef unsigned short order_type;
00037   typedef boost::numeric::odeint::stepper_tag stepper_category;
00038
```

```
00039   typedef boost::numeric::odeint::algebra_stepper_base<Algebra, Operations> algebra_stepper_base_type;
00040   typedef typename algebra_stepper_base_type::algebra_type algebra_type;
00041   typedef typename algebra_stepper_base_type::operations_type operations_type;
00042
00043   static order_type order(void) { return 2; }
00044
00045   midpoint(){}
00046
00047   template<class System>
00048   void do_step(System system, State &in, Time t, Time dt)
00049   {
00050     static const Value val1 = static_cast<Value>(1);
00051     const Time dh = dt / static_cast<Value>(2);
00052     const Time th = t + dh;
00053
00054     //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));
00055     m_resizer.adjust_size(in, [&](const auto &arg){ return resize_impl(arg); });
00056
00057     typename boost::numeric::odeint::unwrap_reference<System>::type &sys = system;
00058
00059     sys(in, deriv_tmp.m_v, t);
00060     algebra_stepper_base_type::m_algebra.for_each3(state_tmp.m_v, in, deriv_tmp.m_v,
00061                             typename operations_type::template scale_sum2<Value, Time>(val1, dh));
00062
00063     sys(state_tmp.m_v, deriv_tmp.m_v, th);
00064     algebra_stepper_base_type::m_algebra.for_each3(state_tmp.m_v, in, deriv_tmp.m_v,
00065                             typename operations_type::template scale_sum2<Value, Time>(val1, dt));
00066
00067     in.swap(state_tmp.m_v);
00068
00069     // Release memory
00070     //m_resizer.adjust_size(State(), [&](const auto &arg){ return resize_impl(arg); });
00071     // deriv_tmp.m_v.clear();
00072     // State().swap(deriv_tmp.m_v);
00073     // state_tmp.m_v.clear();
00074     // State().swap(state_tmp.m_v);
00075   }
00076
00077   // template<class StateType>
00078   // void adjust_size(const StateType &x)
00079   // {
00080   //   resize_impl(x);
00081   // }
00082
00083   bool resize_impl(const State &x)
00084   {
00085     bool resized = false;
00086     resized |= boost::numeric::odeint::adjust_size_by_resizeability(deriv_tmp, x, typename
      boost::numeric::odeint::is_resizeable<State>::type());
00087     resized |= boost::numeric::odeint::adjust_size_by_resizeability(state_tmp, x, typename
      boost::numeric::odeint::is_resizeable<State>::type());
00088     return resized;
00089   }
00090
00091 private:
00092   Resizer m_resizer;
00093
00094   boost::numeric::odeint::state_wrapper<State> deriv_tmp;
00095   boost::numeric::odeint::state_wrapper<State> state_tmp;
00096 };
00097
00098
00099
00100 #endif
```

9.17 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 >

9.17.1 Detailed Description

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

9.18 observer.hpp

Go to the documentation of this file.

```
00001
00007 #ifndef OBSERVER_HPP
00008 #define OBSERVER_HPP
00009
00010 #include <cstdlib>
00011 #include <iostream>
00012 #include <string>
00013 #include <type_traits>
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 "eigen_wrapper.hpp"
00023 #include "fdm3d.hpp"
00024 #include "io.hpp"
00025 #include "physics.hpp"
00026 #include "workspace.hpp"
00027
00028 #ifndef DISABLE_CUDA
00029 #include "cuda_wrapper.cuh"
00030 #include "fdm3d_cuda.cuh"
00031 #endif
00032
00033
00034 template<typename Equation,
00035      bool save_field_spectrum = true,
00036      bool save_density_spectrum = true,
00037      bool save_density = false>
00038 struct ConstIntervalObserver {
00039   typedef typename Equation::Workspace Workspace;
00040   typedef typename Workspace::State State;
00041   typedef State Vector;
00042   Workspace &workspace;
00043   int idx;
00044   std::string dir;
00045   double t_start;
00046   double t_end;
00047   double t_interval;
00048   double t_last;
00049
00050   template<typename Param>
00051   ConstIntervalObserver(const std::string &dir_, const Param &param, Equation &eqn) :
00052     workspace(eqn.workspace), idx(0), dir(dir_),
```

```
00053     t_start(param.t_start), t_end(param.t_end), t_interval(param.t_interval), t_last(param.t_start) {}
00054
00055   ConstIntervalObserver(const ConstIntervalObserver &) = default;
00056
00057   void operator()(const State &x, double t)
00058   {
00059     if(t >= t_last + t_interval || t == t_end || t == t_start) {
00060           long long int N = workspace.N;
00061       double L = workspace.L;
00062       double m = workspace.m;
00063       if constexpr(save_field_spectrum) {
00064       Vector varphi_plus_spectrum = compute_mode_power_spectrum(N, L, m, workspace.state,
      workspace.fft_wrapper);
00065       Eigen::VectorXd varphi_plus_spectrum_out(varphi_plus_spectrum.size());
00066       copy_vector(varphi_plus_spectrum_out, varphi_plus_spectrum);
00067       write_VectorXd_to_filename_template(varphi_plus_spectrum_out, dir +
      "varphi_plus_spectrum_%d.dat", idx);
00068     }
00069
00070       if constexpr(save_density_spectrum) {
00071       Vector rho = Equation::compute_energy_density(workspace, t);
00072       Vector rho_spectrum = compute_power_spectrum(N, rho, workspace.fft_wrapper);
00073       Eigen::VectorXd rho_spectrum_out(rho_spectrum.size());
00074       copy_vector(rho_spectrum_out, rho_spectrum);
00075       write_VectorXd_to_filename_template(rho_spectrum_out, dir + "rho_spectrum_%d.dat", idx);
00076     }
00077
00078       if constexpr(save_density) {
00079       Vector rho = Equation::compute_energy_density(workspace, t);
00080       Eigen::VectorXd rho_copy(rho.size());
00081       copy_vector(rho_copy, rho);
00082       Eigen::VectorXd rho_slice = rho_copy.head(N*N); // Save the density for a = 0 slice.
00083       Eigen::VectorXd rho_axis_average = rho_copy.reshaped(N*N, N).rowwise().mean(); // Save the
      density overaged over a axis.
00084
00085       write_VectorXd_to_filename_template(rho_slice, dir + "rho_slice_%d.dat", idx);
00086       write_VectorXd_to_filename_template(rho_axis_average, dir + "rho_axis_average_%d.dat", idx);
00087     }
00088
00089       workspace.t_list.push_back(t);
00090       t_last = t;
00091       ++idx;
00092     }
00093   }
00094 };
00095
00096
00097
00098 #endif
```

9.19 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 &param)

  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 &param, 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)

9.19.1 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.

9.20 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 std::string_view Mathematica_format<double> = "Real64";
00081
00082 template<> constexpr 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 std::string_view Mathematica_format<double> = "Real64";
00089
00090 template<> constexpr 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
```

9.21 src/physics.hpp File Reference

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

```
#include <cmath>
```

Classes

- struct StaticEOSCosmology

9.21.1 Detailed Description

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

9.22 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
00012 struct StaticEOSCosmology {
00013   double a1;
00014   double H1;
00015   double t1;
00016   double p;
00017
00018   StaticEOSCosmology(const double a1_, const double H1_, const double t1_, const double p_)
00019     : a1(a1_), H1(H1_), t1(t1_), p(p_) {}
00020
00021   // The default constructor from a param assumes radiation domination
00022   template<typename T>
00023   StaticEOSCosmology(const T &param)
00024     : a1(param.a1), H1(param.H1), t1(param.t1), p(1.0) {}
00025
00026   StaticEOSCosmology(void)
00027     : a1(1.0), H1(0), t1(0), p(1.0) {}
00028
00029   double a(const double t) const {
00030     return a1 * pow(1 + (1 + 1 / p) * H1 * (t - t1), p / (1 + p));
00031   }
00032   double H(const double t) const {
00033     return H1 * pow(1 + (1 + 1 / p) * H1 * (t - t1), -1);
00034   }
00035   // We use convention eta1 = p / (a1 * H1).
00036   double eta(const double t) const {
00037     //return eta1 + (p / (a1 * H1)) * (-1 + pow(1 + (1 + 1 / p) * H1 * (t - t1), 1 / (1 + p)));
00038     return (p / (a1 * H1)) * pow(1 + (1 + 1 / p) * H1 * (t - t1), 1 / (1 + p));
00039   }
00040 };
00041
00042
00043 #endif
```

9.23 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 $P_{\dot\varphi}(k) = (k^2 + m^2)P_\varphi(k)$.
- Spectrum **to_deriv_spectrum** (const double m, const double a, const Spectrum &P_f)

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

9.23.1 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.

9.23.2 Function Documentation

9.23.2.1 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

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

Returns

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

The spectrum is given by

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

9.23.2.2 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.
Psi	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.

9.23.2.3 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.
sigma	Standard deviation σ of generated function f.
k_ast	Cutoff k_*.
alpha	Power law index α.

Returns

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

The spectrum is given by

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

9.23.2.4 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.
As	The height of the spectrum A_s.

Returns

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

The spectrum is given by

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

9.24 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
```

9.25 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           0.00001410185368213302 5 * 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) + 0.00004991751161697551 3 * 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
```

9.26 utility.hpp

```
00001 #ifndef UTILITY_H
00002 #define UTILITY_H
00003
00004 #include <iostream>
00005 #include <iomanip>
00006 #include <chrono>
00007 #include <filesystem>
00008
00009 // Pretty print functions
00010 inline static std::string line_separator_with_description(const std::string &description) {
00011   std::string result(80, '=');
00012   const int length = description.length() + 2;
00013   result.replace(80 / 2 - length / 2, length, " " + description + " ");
00014   return result;
00015 }
00016
00017
00018 inline static std::string line_separator_with_description(void) {
00019   std::string result(80, '=');
00020   return result;
00021 }
00022
00023
00024 template<typename Callable>
00025 static void run_and_print(const std::string &description, const Callable &c) {
00026   std::cout << line_separator_with_description(description) << '\n';
00027   c();
00028   std::cout << line_separator_with_description() << '\n';
00029 }
00030
00031
00032 template<typename Callable>
00033 static void run_and_measure_time(const std::string &description, const Callable &c) {
00034   std::cout << line_separator_with_description(description) << '\n';
00035   auto time_start = std::chrono::system_clock::now();
00036   c();
00037   auto time_end = std::chrono::system_clock::now();
00038   std::chrono::duration<double> time_diff = time_end - time_start;
00039   std::cout << std::fixed << std::setprecision(9) << std::left;
00040   std::cout << std::setw(9) << "time spent = " << time_diff.count() << " s" << '\n';
00041   std::cout << line_separator_with_description() << '\n';
00042 }
00043
00044
00045 static void prepare_directory_for_output(const std::string &dir) {
00046   const std::filesystem::path dir_path(dir);
00047   std::error_code ec;
00048   std::cout << line_separator_with_description("Preparing directory for output") << '\n';
00049   std::cout << "Saving results in directory: " << dir << '\n';
00050   std::filesystem::create_directories(dir_path, ec);
00051   std::cout << "ErrorCode = " << ec.message() << '\n';
00052   std::cout << line_separator_with_description() << '\n';
00053 }
00054
00055
00056 // Simple profiler for a big task, taking many cycles
00057 // Note that the function call incurs some time cost, so this is not totally accurate
00058 template<typename Callable>
00059 inline void profile_function(long long int repeat, Callable &&c) {
00060   auto time_start = std::chrono::system_clock::now();
00061   for(long long int i = 0; i < repeat; ++i) {
00062     c();
00063   }
00064   std::cout << line_separator_with_description("Profiling a callable") << '\n';
00065   auto time_end = std::chrono::system_clock::now();
00066   std::chrono::duration<double> time_diff = time_end - time_start;
00067   std::cout << std::fixed << std::setprecision(9) << std::left;
00068   std::cout << std::setw(9) << "total time spent = " << time_diff.count() << " s" << '\n';
00069   std::cout << std::setw(9) << "time spent per iteration = " << time_diff.count() / repeat << " s" << '\n';
00070   std::cout << line_separator_with_description() << '\n';
00071 }
00072
00073
00074
00075 #endif
```

9.27 src/wkb.hpp File Reference

Implementation of the WKB solution.

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

Classes

- struct WKBSolutionForKleinGordonEquationInFRW

9.27.1 Detailed Description

Implementation of the WKB solution.

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

9.28 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
```

9.29 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 >

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>;

9.29.1 Detailed Description

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

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).

9.30 workspace.hpp

Go to the documentation of this file.

```
00001
00008 #ifndef WORKSPACE_H
00009 #define WORKSPACE_H
00010
00011 #include <memory>
00012
00013 #include "param.hpp"
00014 #include "physics.hpp"
00015 #include "fftw_wrapper.hpp"
00016 #include "dispatcher.hpp"
00017
00018 #define TYPE_REQUIREMENT(value, type) {std::remove_cvref_t<decltype((value))>()} ->
      std::same_as<type>;
00019
00020
00021
00022 template<typename Param>
00023 concept HasLatticeParams = requires (Param param)
00024   { TYPE_REQUIREMENT(param.N, long long int)
00025     TYPE_REQUIREMENT(param.L, double) };
00026
00027 template<typename Param>
00028 concept HasMass = requires (Param param) { TYPE_REQUIREMENT(param.m, double) };
00029
00030 template<typename Param>
00031 concept HasLambda = requires (Param param) { TYPE_REQUIREMENT(param.lambda, double) };
00032
00033 template<typename Param>
00034 concept HasFa = requires (Param param) { TYPE_REQUIREMENT(param.f_a, double) };
00035
00036 template<typename Param>
00037 concept HasFRWParameters = requires (Param param)
00038   { TYPE_REQUIREMENT(param.a1, double)
00039     TYPE_REQUIREMENT(param.H1, double)
00040     TYPE_REQUIREMENT(param.t1, double) };
00041
00042 template<typename Param>
00043 concept HasPsiApproximationParameters = requires (Param param)
00044   { TYPE_REQUIREMENT(param.M, long long int) };
00045
00046
00047 /*
00048    The workspace for solving equations.
```

```
00049    The lifetime of objects in the workspace are managed by us (this codebase), instead of external
       libraries (e.g. odeint).
00050 */
00051 template<typename Vector>
00052 struct WorkspaceGeneric {
00053   typedef Vector State;
00054   long long int N;
00055   double L;
00056   double m;
00057   StaticEOSCosmology cosmology{};
00058   State state;
00059   double lambda{0};
00060   double f_a{1.0};
00061   Vector Psi;
00062   Vector dPsidt;
00063   Vector Psi_fft;
00064   Vector dPsidt_fft;
00065   Vector R_fft;
00066   std::vector<double> t_list;
00067   typename fftWrapperDispatcher<Vector>::Generic fft_wrapper;
00068
00069   bool Psi_approximation_initialized{false};
00070   long long int M;
00071   std::unique_ptr<typename fftWrapperDispatcher<Vector>::Generic> fft_wrapper_M_ptr;
00072   Vector cutoff_R_fft;
00073
00074   template<HasLatticeParams Param>
00075   WorkspaceGeneric(const Param &param, auto &initializer) :
00076     N(param.N), L(param.L), fft_wrapper(param.N)
00077   {
00078     //static_assert(HasLatticeParams<Param>, "HasLatticeParams<Param> test failed.");
00079     if constexpr(HasFRWParameters<Param>) { cosmology = StaticEOSCosmology(param); }
00080     if constexpr(HasMass<Param>) { m = param.m; }
00081     if constexpr(HasLambda<Param>) { lambda = param.lambda; }
00082     if constexpr(HasFa<Param>) { f_a = param.f_a; }
00083     if constexpr(HasPsiApproximationParameters<Param>) { M = param.M;
00084     assert(N >= M); }
00085     initializer(param, *this);
00086   }
00087 };
00088
00089
00090
00091
00092 #endif
```

Index