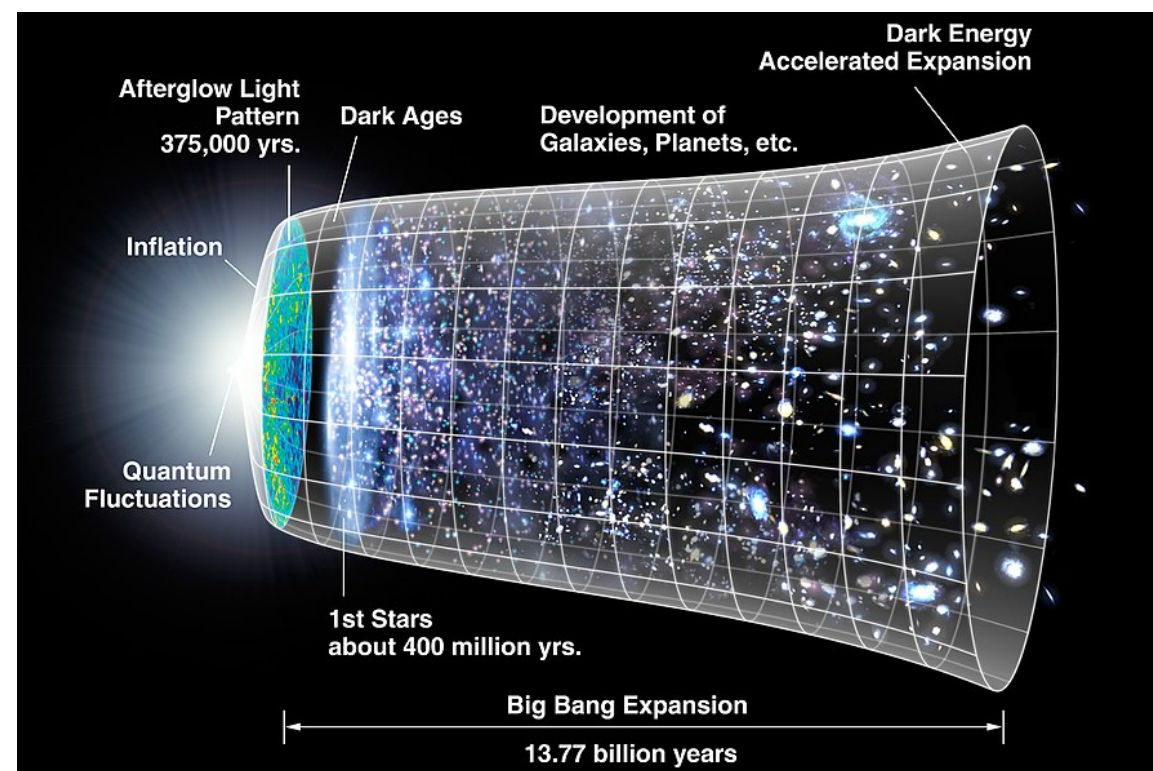


Numerical Project Milestone I

Revenge of the Cosmological Constant



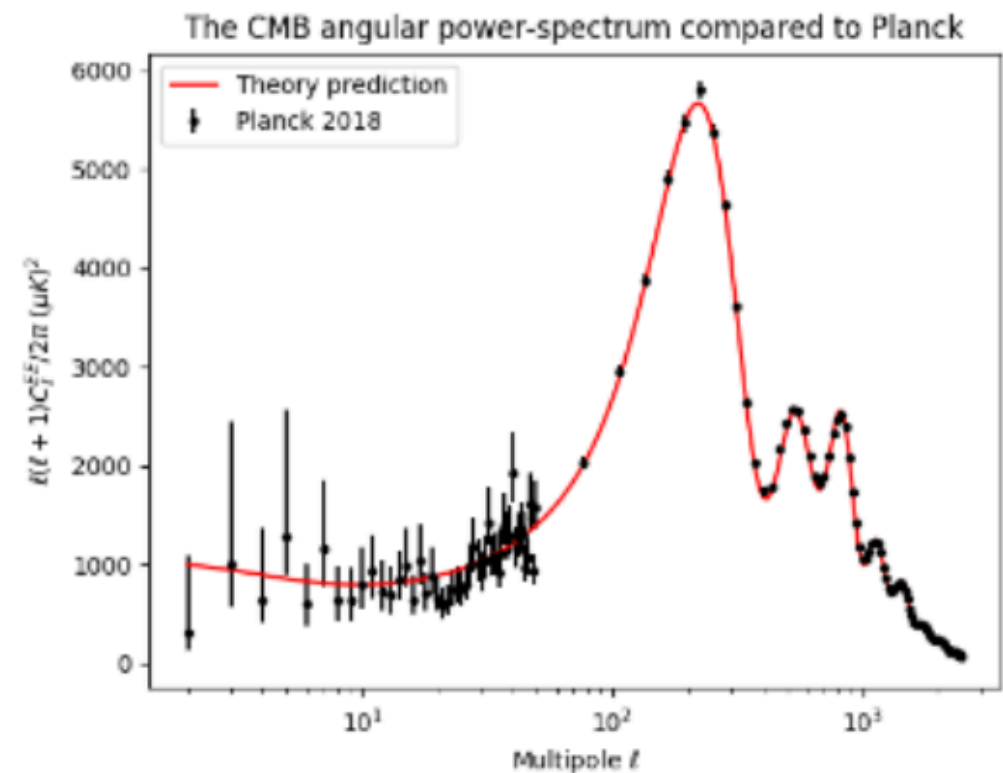
AST5220 / AST9420 Spring 2024
Hans Winther

Aim of this project

Theoretical predictions for
CMB observations

Parameters

$$\begin{aligned}h &= 0.67, \\T_{\text{CMB}0} &= 2.7255 \text{ K}, \\N_{\text{eff}} &= 3.046, \\\Omega_{\text{b}0} &= 0.05, \\\Omega_{\text{CDM}0} &= 0.267, \\\Omega_{k0} &= 0, \\\Omega_{\nu0} &= N_{\text{eff}} \cdot \frac{7}{8} \left(\frac{4}{11} \right)^{4/3} \Omega_{\gamma0}, \\\Omega_{\Lambda0} &= 1 - (\Omega_{k0} + \Omega_{\text{b}0} + \Omega_{\text{CDM}0} + \Omega_{\gamma0} + \Omega_{\nu0}), \\n_s &= 0.965, \\A_s &= 2.1 \cdot 10^{-9}, \\Y_p &= 0.245, \\z_{\text{reion}} &= 8, \\\Delta z_{\text{reion}} &= 0.5, \\z_{\text{Hereion}} &= 3.5, \\\Delta z_{\text{Hereion}} &= 0.5.\end{aligned}$$



Parameters

$$\begin{aligned} h &= 0.67, \\ T_{\text{CMB}0} &= 2.7255 \text{ K}, \\ N_{\text{eff}} &= 3.046, \\ \Omega_{b0} &= 0.05, \\ \Omega_{\text{CDM}0} &= 0.267, \\ \Omega_{k0} &= 0, \\ \Omega_{\nu0} &= N_{\text{eff}} \cdot \frac{7}{8} \left(\frac{4}{11} \right)^{4/3} \Omega_{\gamma0}, \\ \Omega_{\Lambda0} &= 1 - (\Omega_{k0} + \Omega_{b0} + \Omega_{\text{CDM}0} + \Omega_{\gamma0} + \Omega_{\nu0}), \\ n_s &= 0.965, \\ A_s &= 2.1 \cdot 10^{-9}, \\ Y_p &= 0.245, \\ z_{\text{reion}} &= 8, \\ \Delta z_{\text{reion}} &= 0.5, \\ z_{\text{Hereion}} &= 3.5, \\ \Delta z_{\text{Hereion}} &= 0.5. \end{aligned}$$

Background Cosmology

$$H(x), \mathcal{H}(x), \frac{d\mathcal{H}(x)}{dx}, \frac{d^2\mathcal{H}(x)}{dx^2}, \eta(x), \Omega_i(x)$$

Recombination

$$\frac{dX_e}{dx} = \frac{C_r(T_b)}{H} \left[\beta(T_b)(1 - X_e) - n_H \alpha^{(2)}(T_b) X_e^2 \right],$$

Linear perturbations

$$\begin{aligned} C_r(T_b) &= \frac{\Lambda_{2s \rightarrow 1s} + \Lambda_\alpha}{\Lambda_{2s \rightarrow 1s} + \Lambda_\alpha + \beta^{(2)}(T_b)}, \text{ (dimensionless),} \\ H, &\text{ (dimension 1/s)} \\ \Lambda_{2s \rightarrow 1s} &= 8.227 \text{ s}^{-1}, \text{ (dimension 1/s)} \\ \Lambda_\alpha &= H \frac{(3\epsilon_0)^3}{(8\pi)^2 n_{1s}}, \text{ (dimension 1/s)} \\ n_{1s} &= (1 - X_e) n_H, \text{ (dimension 1/m}^3\text{)} \\ n_H &= (1 - Y_p) \frac{3H_0^2 \Omega_{b0}}{8\pi G m_H a^3}, \text{ (dimension 1/m}^3\text{)} \\ \beta^{(2)}(T_b) &= \beta(T_b) e^{3\epsilon_0/4T_b}, \text{ (dimension 1/s)} \\ \beta(T_b) &= \alpha^{(2)}(T_b) \left(\frac{m_e T_b}{2\pi} \right)^{3/2} e^{-\epsilon_0/T_b}, \text{ (dimension 1/s)} \\ \alpha^{(2)}(T_b) &= \frac{64\pi}{\sqrt{27\pi}} \frac{\alpha^2}{m_e^2} \sqrt{\frac{\epsilon_0}{T_b}} \phi_2(T_b), \text{ (dimension m}^3\text{/s)} \\ \phi_2(T_b) &= 0.448 \ln(\epsilon_0/T_b), \text{ (dimensionless).} \end{aligned}$$

Photon temperature multipoles:

$$\begin{aligned} \Theta'_0 &= \frac{ck}{\mathcal{H}} \Theta_1 - \Phi', \\ \Theta'_1 &= \frac{ck}{3\mathcal{H}} \Theta_2 + \frac{2ck}{3\mathcal{H}} \Theta_1 + \frac{ck}{3\mathcal{H}} \Psi + \left[\Theta_1 + \frac{1}{3} v_b \right], \\ \Theta'_\ell &= \frac{\ell ck}{(2\ell+1)\mathcal{H}} \Theta_{\ell-1} - \frac{(\ell+1)ck}{(2\ell+1)\mathcal{H}} \Theta_{\ell+1} + \left[\Theta_\ell - \frac{1}{10} \Pi \delta_{\ell,2} \right], \quad 2 \leq \ell < \ell_{\text{max}} \\ \Theta'_\ell &= \frac{ck}{\mathcal{H}} \Theta_{\ell-1} - \frac{(\ell+1)ck}{\mathcal{H}\eta(x)} \Theta_{\ell+1} + \left[\Theta_\ell - \frac{1}{10} \Pi \delta_{\ell,2} \right], \quad \ell = \ell_{\text{max}} \end{aligned}$$

Photon polarization multipoles:

$$\begin{aligned} \Theta'_{P0} &= \frac{ck}{\mathcal{H}} \Theta_{P1} + \tau' \left[\Theta_{P0} - \frac{1}{2} \Pi \right], \\ \Theta'_{P\ell} &= \frac{\ell ck}{(2\ell+1)\mathcal{H}} \Theta_{P,\ell-1}^P - \frac{(\ell-1)ck}{(2\ell+1)\mathcal{H}} \Theta_{P,\ell+1}^P + \left[\Theta_{P\ell}^P + \tau' \Theta_\ell^P - \frac{1}{10} \Pi \delta_{\ell,2} \right], \quad 1 \leq \ell < \ell_{\text{max}} \\ \Theta'_{P,\ell} &= \frac{ck}{\mathcal{H}} \Theta_{P,\ell-1}^P - \frac{(\ell+1)ck}{\mathcal{H}\eta(x)} \Theta_{P,\ell+1}^P + \left[\Theta_{P\ell}^P + \tau' \Theta_\ell^P \right], \quad \ell = \ell_{\text{max}} \end{aligned}$$

Neutrino multipoles:

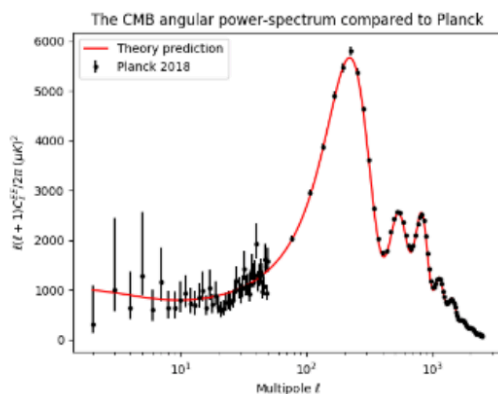
$$\begin{aligned} \mathcal{N}'_0 &= \frac{ck}{\mathcal{H}} \mathcal{N}_1 - \Phi', \\ \mathcal{N}'_1 &= \frac{ck}{3\mathcal{H}} \mathcal{N}_2 + \frac{2ck}{3\mathcal{H}} \mathcal{N}_1 + \frac{ck}{3\mathcal{H}} \Psi + \left[\mathcal{N}_1 + \frac{1}{3} v_b \right], \\ \mathcal{N}'_\ell &= \frac{\ell ck}{(2\ell+1)\mathcal{H}} \mathcal{N}_{\ell-1} - \frac{(\ell+1)ck}{(2\ell+1)\mathcal{H}} \mathcal{N}_{\ell+1} + \left[\mathcal{N}_\ell - \frac{1}{10} \Pi \delta_{\ell,2} \right], \quad 2 \leq \ell < \ell_{\text{max},\nu} \\ \mathcal{N}'_\ell &= \frac{ck}{\mathcal{H}} \mathcal{N}_{\ell-1} - \frac{(\ell+1)ck}{\mathcal{H}\eta(x)} \mathcal{N}_{\ell+1} + \left[\mathcal{N}_\ell - \frac{1}{10} \Pi \delta_{\ell,2} \right], \quad \ell = \ell_{\text{max},\nu} \end{aligned}$$

Cold dark matter and baryons:

$$\begin{aligned} \delta'_{\text{CDM}} &= \frac{ck}{\mathcal{H}} v_{\text{CDM}} - 3\Phi', \\ v'_{\text{CDM}} &= -v_{\text{CDM}} - \frac{ck}{\mathcal{H}} \Psi, \\ \delta'_b &= \frac{ck}{\mathcal{H}} v_b - 3\Phi', \\ v'_b &= -v_b - \frac{ck}{\mathcal{H}} \Psi + \tau' R (\Theta_1 + v_b) \end{aligned}$$

Metric perturbations:

$$\begin{aligned} \Phi' &= \Psi - \frac{c^2 k}{3\mathcal{H}^2} \Phi + \frac{H_0}{2\mathcal{H}^2} \left[\Omega_{\text{CDM}0} a^{-1} \delta_{\text{CDM}} + \Omega_{b0} a^{-1} \delta_b + 4\Omega_{\gamma0} a^{-2} \Theta_0 + 4\Omega_{\nu0} a^{-2} \mathcal{N}_0 \right] \\ \Psi &= -\Phi - \frac{12H_0^2}{c^2 k^2 a^2} \left[\Omega_{\gamma0} \Theta_2 + \Omega_{\nu0} \mathcal{N}_2 \right] \end{aligned}$$



Power-spectrum

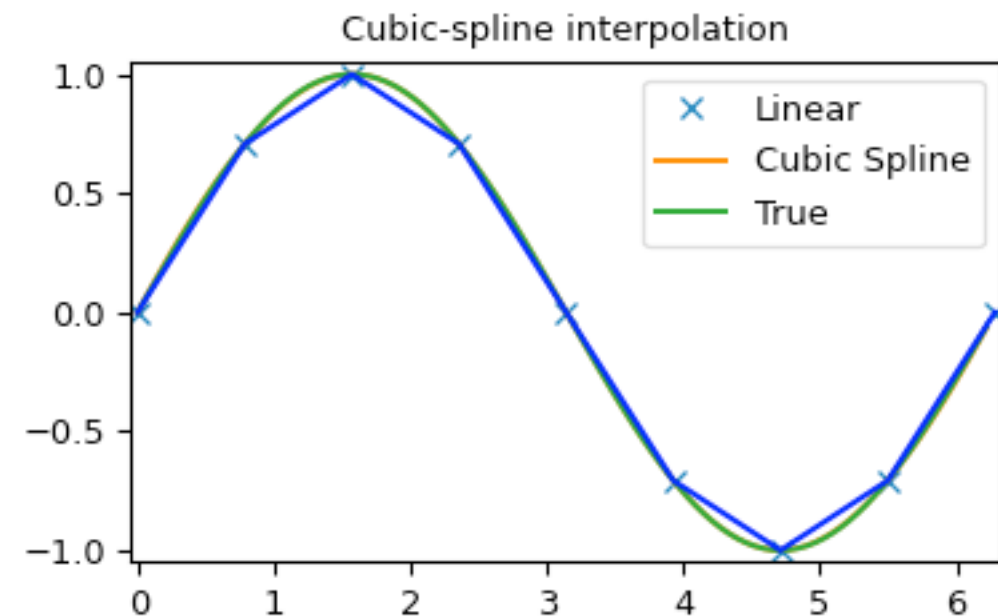
$$C_\ell = \frac{2}{\pi} \int k^2 P_{\text{primordial}}(k) \Theta_\ell^2(k) dk.$$

Two key numerical tools

ODE Solver - For solving coupled ordinary differential equations

$$\begin{pmatrix} y_1^{(n)} \\ y_2^{(n)} \\ \vdots \\ y_m^{(n)} \end{pmatrix} = \begin{pmatrix} f_1(x, y, y', y'', \dots, y^{(n-1)}) \\ f_2(x, y, y', y'', \dots, y^{(n-1)}) \\ \vdots \\ f_m(x, y, y', y'', \dots, y^{(n-1)}) \end{pmatrix}$$

Spline - For interpolation functions defined on discrete points to any point + computing derivatives



We will use these again and again so make sure you understand it and learn how to use it correctly

Overview Milestone I

- **Very simple task**: implement a class / module that solves the background.
- Take cosmological parameters as input, compute derived parameters (e.g. Ω_R follows from T_{cmb} , Ω_{ν} follows from Ω_R , H_0 follows from 'h', ...)
- Provide functions to get the Hubble function and derivatives $H(x)$, $dH/dx(x)$, ..., density functions $\Omega_B(x)$, $\Omega_{\text{CDM}}(x)$...
- Solve the conformal time ODE and the cosmic time ODE and spline the result.
- Also compute some key times in the history of the Universe for our fiducial cosmology: matter-radiation equality, onset of acceleration, etc. (see website for list of deliverables).
- Use your code to derive constraints on the parameters by fitting to supernova data.
- If you are a master-student you are free to ignore curvature and neutrinos.

```
class BackgroundCosmology{
private:
    // Cosmological parameters
    double h; // Little h = H0/(100km/s/Mpc)
    double OmegaB; // Baryon density today
    double OmegaCDM; // CDM density today
    double OmegaLambda; // Dark energy density today
    double Neff; // Effective number of relativis
    double TCMB; // Temperature of the CMB today

    // Derived parameters
    double OmegaR; // Photon density today (follows
    double OmegaNu; // Neutrino density today (follo
    double OmegaK; // Curvature density = 1 - Omega
    double H0; // The Hubble parameter today H0

    // Start and end of x-integration (can be changed)
    double x_start = Constants.x_start;
    double x_end = Constants.x_end;

    // Splines to be made
    Spline eta_of_x_spline{"eta"};

public:
    // Constructors
    BackgroundCosmology() = delete,
    BackgroundCosmology(
        double h,
        double OmegaB,
        double OmegaCDM,
        double OmegaLambda,
        double Neff,
        double TCMB
    );

    // Print some useful info about the class
    void info() const;

    // Do all the solving
    void solve();

    // Output some results to file
    void output(const std::string filename) const;

    // Get functions that we must implement
    double eta_of_x(double x) const;
    double H_of_x(double x) const;
    double Hp_of_x(double x) const;
    double dHpdx_of_x(double x) const;
    double ddHpddx_of_x(double x) const;
    double get_OmegaB(double x = 0.0) const;
    double get_OmegaM(double x = 0.0) const;
    double get_OmegaR(double x = 0.0) const;
    double get_OmegaRtot(double x = 0.0) const;
    double get_OmegaNu(double x = 0.0) const;
    double get_OmegaCDM(double x = 0.0) const;
    double get_OmegaLambda(double x = 0.0) const;
    double get_OmegaK(double x = 0.0) const;
    double get_OmegaMnu(double x = 0.0) const;
    double get_H0() const;
    double get_h() const;
    double get_Neff() const;
    double get_TCMB() const;
```

Download the code template

- **You can get the code from GitHub:** <https://github.com/HAWinther/AST5220-Cosmology/> (run [`git clone https://github.com/HAWinther/AST5220-Cosmology.git`] in your terminal)
- This has C++ templates (**highly recommended** - this is what I will assume in these notes) plus old Fortran and Python templates. We can also offer some support with Julia (talk to Herman). But you are free to use any programming language you want and feel free to change whatever you want in the templates. It's your code and you do it the way you think is best!
- Once downloaded. Edit the **Makefile**. You need to set paths to the **GSL library**. See the README in the code template for how to install this. Once this is done try to compile the codes by running [`make clean; make`] and run it [`./cmb`]. If this works you are ready to start!
- In the C++ template there is an Examples.cpp file that you should take a look at for how to do simple tasks like solving ODEs and making splines. Try running [`make examples`] to compile the examples and run it [`./examples`].
- For this milestone the relevant files to look at are `src/BackgroundCosmology.h` and `src/BackgroundCosmology.cpp`. Look for `// TODO: ...` for hints on how to get started.
- For each milestone after this you need to edit the `src/Main.cpp` file (comment out “Remove when module is completed” return statement).

Getting started

- There are some other utils in the **Utils:: namespace**. E.g. create a linear spaced arrays using `auto x_array = Utils::linspace(xmin, xmax, n)`, compute `exp, sin, cos, ...` on an array is simply `auto a = exp(x_array)`. Bessel functions etc.
- Let me know if you have any problems installing and running the code and I can help out.
- It is possible to get an account on ITA and run it from there (by using ssh), let me know if you want this and I can set it up.
- If you are stuck on something get in touch as soon as possible and we'll sort it out together.

Getting started

- How the output of a run of the code looks like “out of the box”. Some parameters show “nonsense” values as they have not been set (your job) and we get an error “Spline has not been created” because we try to evaluate the conformal time spline which has not yet been created (your job).

```
Info about cosmology class:
OmegaB:      0.05
OmegaCDM:    0.25
OmegaLambda: 0.7
OmegaK:      4.44659e-323
OmegaNu:     4.94066e-324
OmegaR:      4.94066e-324
Neff:        3.046
h:           0.7
TCMB:        2.7255

Error Spline::eval [eta] Spline has not been created!
terminate called after throwing an instance of 'char const*'
Aborted
```


Code Template

**For more info about the code template and C++ see
<https://cmb.wintherscoming.no/about.php>**

Some basic info about C++

Definition file:

- Each milestone is defined as a class.
- In the template every class has a **definition file** (.h file) and an **implementation file** (.cpp file). The definition file tells us what is available within the class (private) and what is available outside the class (public).
- If you want to add a new class variable then you need to add it in the h-file and then you can use it within any function in the implementation file.
- Likewise if you want to add a new class function you must declare it in the h-file.
- To make a class variable available outside the class you can place it in the public-section (not a great idea as it can be modified outside the class) or make a function that returns a copy of it (better solution) like you can see on the right (get_h, get_H0 etc.)
- Arrays start from 0 and go up to N-1, e.g. `auto x = Vector(3);` declares a 3 element vector of real numbers that is accessed via `a[0]`, `a[1]`, `a[2]`.

```
class BackgroundCosmology{
private:
    // Cosmological parameters
    double h; // Little h = H0/(100km/s/Mpc)
    double OmegaB; // Baryon density today
    double OmegaCDM; // CDM density today
    double OmegaLambda; // Dark energy density today
    double Neff; // Effective number of relativis
    double TCMB; // Temperature of the CMB today

    // Derived parameters
    double OmegaR; // Photon density today (follows
    double OmegaNu; // Neutrino density today (follo
    double OmegaK; // Curvature density = 1 - Omega
    double H0; // The Hubble parameter today H0

    // Start and end of x-integration (can be changed)
    double x_start = Constants.x_start;
    double x_end = Constants.x_end;

    // Splines to be made
    Spline eta_of_x_spline("eta");

public:
    // Constructors
    BackgroundCosmology() = delete,
    BackgroundCosmology(
        double h,
        double OmegaB,
        double OmegaCDM,
        double OmegaLambda,
        double Neff,
        double TCMB
    );

    // Print some useful info about the class
    void info() const;

    // Do all the solving
    void solve();

    // Output some results to file
    void output(const std::string filename) const;

    // Get functions that we must implement
    double eta_of_x(double x) const;
    double H_of_x(double x) const;
    double Hp_of_x(double x) const;
    double dHpdx_of_x(double x) const;
    double ddHpddx_of_x(double x) const;
    double get_OmegaB(double x = 0.0) const;
    double get_OmegaM(double x = 0.0) const;
    double get_OmegaR(double x = 0.0) const;
    double get_OmegaRtot(double x = 0.0) const;
    double get_OmegaNu(double x = 0.0) const;
    double get_OmegaCDM(double x = 0.0) const;
    double get_OmegaLambda(double x = 0.0) const;
    double get_OmegaK(double x = 0.0) const;
    double get_OmegaMnu(double x = 0.0) const;
    double get_H0() const;
    double get_h() const;
    double get_Neff() const;
    double get_TCMB() const;
```

How to make a Spline

See Examples.cpp

```
void make_spline(){  
  
    const double xmin = 0.0;  
    const double xmax = 1.0;  
    const int     npts = 10;  
  
    Vector x_array = Utils::linspace(xmin, xmax, npts);  
    Vector y_array = exp(x_array);  
  
    Spline f_spline(x_array, y_array, "Function y = exp(x)");  
  
    std::cout << "e^log(2) = " << f_spline( log(2) ) << "\n";  
}
```

Gives an error if you try to use it before its made.
Can show warning if you are out of bounds (turn this on!).

To learn more about the algorithms used to make such a spline see
https://cmb.wintherscoming.no/theory_numerical.php

How to solve an ODE

See Examples.cpp

```
void solve_coupled_ode(){  
  
    // Domain over which we want to solve the ODE  
    const double xmin = 0.0;  
    const double xmax = 1.0;  
    const int     npts = 10;  
  
    // Array of points to store the solution at  
    Vector x_array = Utils::linspace(xmin, xmax, npts);  
  
    // Define the ODE  $y_0' = y_1$  ;  $y_1' = -y_0$   
    ODEFunction dydx = [&](double x, const double *y, double *dydx){  
        dydx[0] = y[1];  
        dydx[1] = -y[0];  
        return GSL_SUCCESS;  
    };  
  
    // Initial conditions  
    double y0_ini = 0.0;  
    double y1_ini = 1.0;  
    Vector y_ic{y0_ini, y1_ini};  
  
    // Solve the ODE  
    ODESolver ode;  
    ode.solve(dydx, x_array, y_ic);  
  
    // Get the data: this is a Vector2D with data[i] = {y0(xi), y1(xi)}  
    auto result = ode.get_data();  
}
```

To learn more about the algorithms used to solve ODEs see
https://cmb.wintherscoming.no/theory_numerical.php

Good coding practices

- Use proper names for variables and functions. If you have an array containing the conformal time call it `eta_arr` or `conformal_time_arr` or similar. Don't call things `a,b,c,d,e,f,g`! Its unreadable! Code should be self-explanatory.
- Document the code with comments. Especially if you do something special that is not obvious from reading the code! Related to the thing above - if you use proper names then the code will be readable without much comments.
- Turn on error-checks for splines so you get a warning if it tries to evaluate the function out of bounds so you know if you messed up something.
- Don't allocate memory using [**new**], its a really bad idea. Always use standard containers like `Vector = std::vector<double>`. You will forget to free something and get memory leaks!

```
// Bad
double e, t, om, a, b, c, d, ...;
void comp(){
    //...
};

// Better
double eta, tau, OmegaM, ... ;
void compute_conformal_time(){
    //...
}
```

```
// Solve the Saha equation system by using
// the iterative method in Callin
while( (f_electron - f_electron_old) > 1e-10){
    //...
}
```

```
// Don't do this!
double *bad_idea = new double[100];

//...you will access elements outside the range
bad_idea[101] = 10.0;

//...and you will forget to free the memory
delete[] bad_idea;

// Use a standard container
Vector better(100);

// And turn on the compiler flags to get an error if you do this
better[101] = 10.0; // Throws error!
```

Main

- Code runs from **Main.cpp**. From here we create the objects we need in this project, do the solving, output stuff etc.
- The background object is defined in **BackgroundCosmology.h** and implemented in **BackgroundCosmology.cpp**

1) Set the parameters

2) Create a Background object by passing in the parameters and initialise it

3) Call the solve method that does all the solving we need

6) Remove this line when done and you are ready to move on to milestone II

```
int main(int argc, char **argv){
    Utils::StartTiming("Everything");

    //=====
    // Parameters
    //=====

    // Background parameters
    double h          = 0.7;
    double OmegaB      = 0.05;
    double OmegaCDM    = 0.25;
    double OmegaLambda = 0.7;
    double Neff        = 3.046;
    double TCMB        = 2.7255;

    // Recombination parameters
    double Yp          = 0.24;

    //=====
    // Module I
    //=====

    // Set up and solve the background
    BackgroundCosmology cosmo(h, OmegaB, OmegaCDM, OmegaLambda, Neff, TCMB);
    cosmo.solve();
    cosmo.info();

    // Output background evolution quantities
    cosmo.output("cosmology.txt");

    // Remove when module is completed
    return 0;

    //=====
    // Module II
    //=====
```

4) Print some info to screen

5) Output results to file for plotting

Step 1: Initialization

- Method `BackgroundCosmology::BackgroundCosmology` in `BackgroundCosmology.cpp`

This is the constructor that is run when the object is created. This is where you should do all the initialisation.

```
BackgroundCosmology::BackgroundCosmology(  
    double h,  
    double OmegaB,  
    double OmegaCDM,  
    double OmegaLambda,  
    double Neff,  
    double TCMB) :  
    h(h),  
    OmegaB(OmegaB),  
    OmegaCDM(OmegaCDM),  
    OmegaLambda(OmegaLambda),  
    Neff(Neff),  
    TCMB(TCMB)  
{  
  
    //=====   
    // TODO: Compute OmegaR, OmegaNu, OmegaK, H0, ...  
    //=====   
    //...  
    //...  
    //...  
    //...  
}
```

Parameters that
the object
takes in

Store them in
the class variables
with the same name

Set H0, OmegaR, ... and do
any other initialisation


Step 2: Implementing functions needed for the solving etc.

- Method `BackgroundCosmology::H_of_x(double x)` in `BackgroundCosmology.cpp`

In order to solve for the conformal time and the age of the Universe we need to implement the Hubble function (as function of $x = \log(a)$). You also need to implement $H_p = aH$, derivatives of this (needed for future milestones) and the density functions (Omega's).

```
double BackgroundCosmology::H_of_x(double x) const{  
  
    //=====   
    // TODO: Implement...   
    //=====   
    //...   
    //...   
  
    return 0.0;  
}
```

**Return the value of
the Hubble
function $H(x)$**



**NB: $x = \log(a)$ is our time-coordinate
so if e.g. $H = 1/a$ then $H(x) = \exp(-x)$**

Step 3: Doing the solving

- Method `BackgroundCosmology::solve` in `BackgroundCosmology.cpp`

This is where the solving is done. Set up an $x = \log(a)$ array and solve the ODE to get the conformal time at all those points. Spline the result in the `eta_of_x_spline` (that I already defined for you in the h-file). What is not added here is to compute the lifetime of the Universe and you should also do that here

Make array of x-points from early Universe till today

Define RHS of the conformal time ODE

**Spline the result
(and use the spline in `get_eta(x)`
to return the value)**

```
//=====
// Do all the solving. Compute eta(x)
//=====

// Solve the background
void BackgroundCosmology::solve(){
    Utils::StartTiming("Eta");

    //=====
    // TODO: Set the range of x and the number of points for the splines
    // For this Utils::linspace(x_start, x_end, npts) is useful
    //=====
    Vector x_array;

    // The ODE for deta/dx
    ODEFunction detadx = [&](double x, const double *eta, double *detadx){

        //=====
        // TODO: Set the rhs of the detadx ODE
        //=====
        //...
        //...

        detadx[0] = 0.0;

        return GSL_SUCCESS;
    };

    //=====
    // TODO: Set the initial condition, set up the ODE system, solve and make
    // the spline eta_of_x_spline
    //=====
    // ...
    // ...
    // ...
    // ...

    Utils::EndTiming("Eta");
}
```

Step 4: Checking the results and outputting data

- Once you have implemented everything you should make some results. First test the result by calling the `info()` function to output some info and check that it is correct (you can add printing the lifetime of the Universe here).
- Then you can use the `output()` function to output some data to file (you can change this as you want, its just an example).

```
//=====
// Print out info about the class
//=====
void BackgroundCosmology::info() const{
    std::cout << "\n";
    std::cout << "Info about cosmology class:\n";
    std::cout << "OmegaB:      " << OmegaB      << "\n";
    std::cout << "OmegaCDM:    " << OmegaCDM    << "\n";
    std::cout << "OmegaLambda: " << OmegaLambda << "\n";
    std::cout << "OmegaK:      " << OmegaK      << "\n";
    std::cout << "OmegaNu:     " << OmegaNu     << "\n";
    std::cout << "OmegaR:      " << OmegaR      << "\n";
    std::cout << "Neff:        " << Neff        << "\n";
    std::cout << "h:           " << h           << "\n";
    std::cout << "TCMB:        " << TCMB        << "\n";
    std::cout << std::endl;
}
```

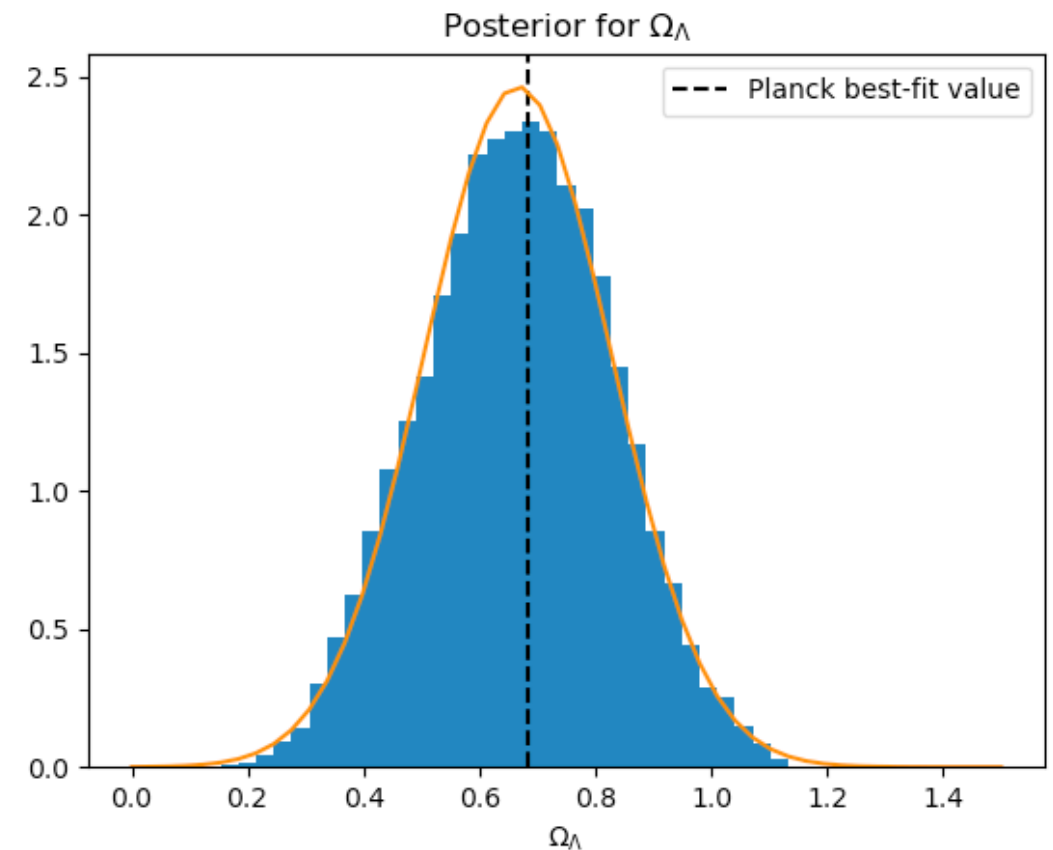
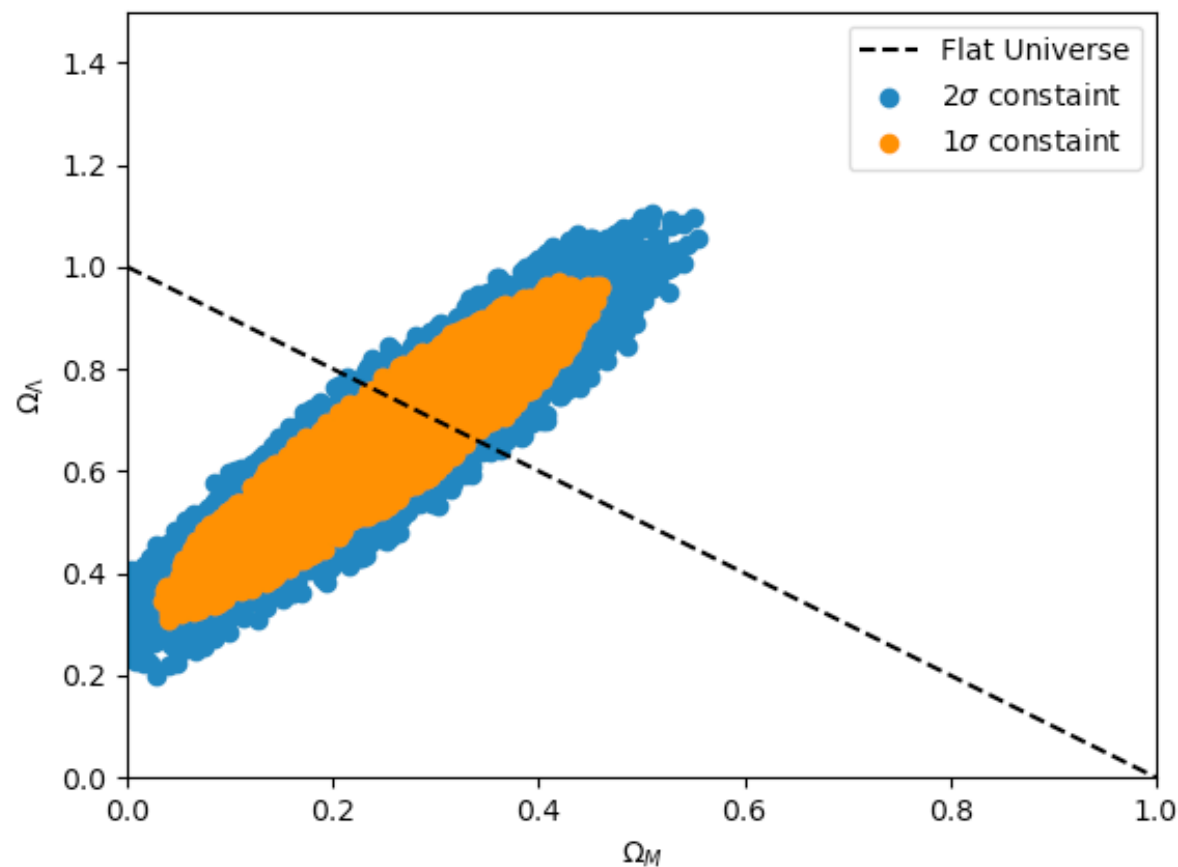
```
//=====
// Output some data to file
//=====
void BackgroundCosmology::output(const std::string filename) const{
    const double x_min = -10.0;
    const double x_max = 0.0;
    const int    n_pts = 100;

    Vector x_array = Utils::linspace(x_min, x_max, n_pts);

    std::ofstream fp(filename.c_str());
    auto print_data = [&] (const double x) {
        fp << x << " ";
        fp << eta_of_x(x) << " ";
        fp << Hp_of_x(x) << " ";
        fp << dHpdx_of_x(x) << " ";
        fp << get_OmegaB(x) << " ";
        fp << get_OmegaCDM(x) << " ";
        fp << get_OmegaLambda(x) << " ";
        fp << get_OmegaR(x) << " ";
        fp << get_OmegaNu(x) << " ";
        fp << get_OmegaK(x) << " ";
        fp << "\n";
    };
    std::for_each(x_array.begin(), x_array.end(), print_data);
}
```

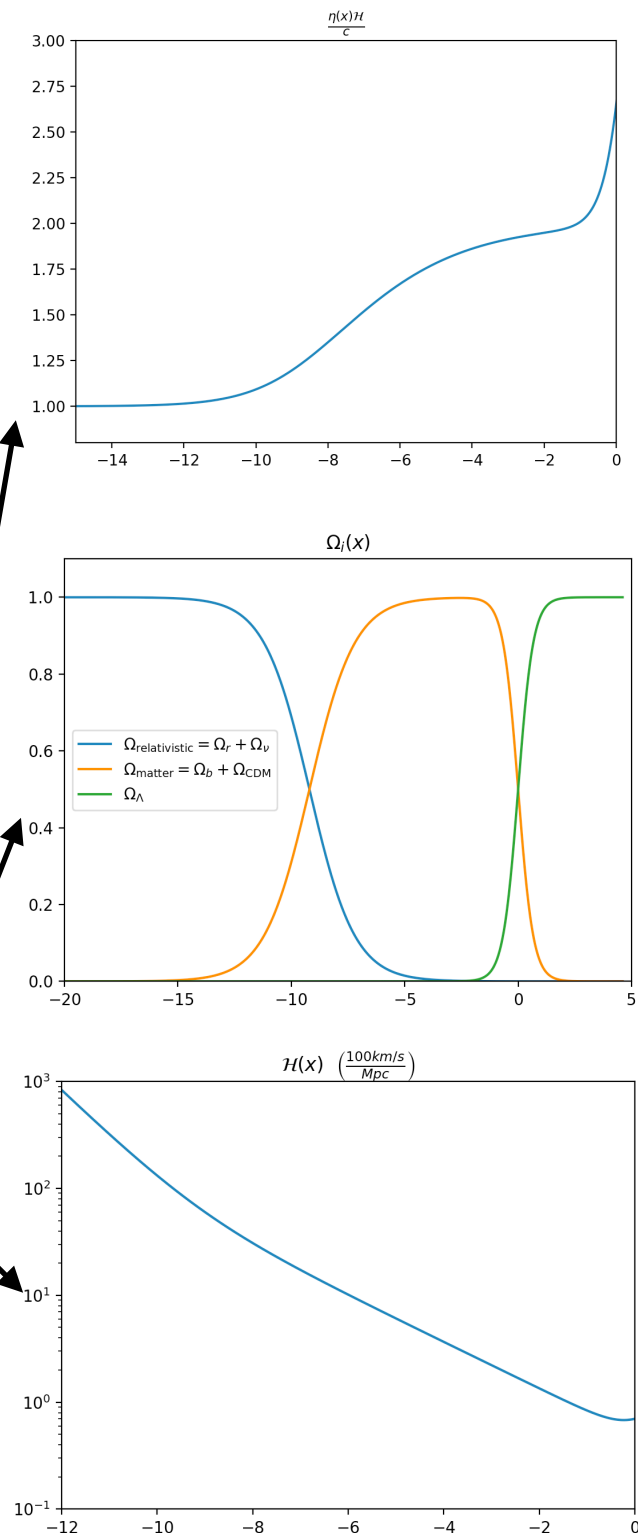
Step 5: Derive constraints from supernova data (Demonstrate that the cosmological constant is non-zero)

- Run Monte Carlo Markov Chains to derive constraints on the cosmological parameters using data from Supernova observations (using the redshift-luminosity distance relation).
- Data is on website. The code to run the chains is provided.



Step 6: Plotting

- For the report you need to make plots. The easiest thing is to just dump results to a text-file and use **Matplotlib** in **Python**, **Gnuplot** or whatever you prefer to make the plots.
- NB: use sensible axes for the plots. If you for example plot $H(a)$ vs a then you must use logarithmic axes otherwise you see nothing. Plots should be informative.
- Useful to add vertical lines denoting relevant times if they are relevant for understanding the plot (e.g. matter-radiation equality, onset of acceleration).
- See the website for plots you can compare your results to (for different parameters than you are meant to use for the report).



Step 7: Writing the report

- See <https://cmb.wintherscoming.no/milestone1.php> for a list of all the things you should compute / plots to make. You are of course free to make and include other plots if you want to.
- You write it **using the LaTeX template of the Astronomy and Astrophysics journal** so that you get experience writing an actual paper. Try to download and use it and let me know if you have problems.
- See <https://cmb.wintherscoming.no/report.php> for how to write the report.
- If you have problems with that one option is to write the report online using for example Overleaf (its free as long as you don't have many projects and has the template already included).
- For the equations you need in the report you can just copy that from the website (right-click an equation in your browser and press "Show Math As... -> Latex").
- Send me the code by email (or just a link to a GitHub repository if you have that) and the report when you are done.



Constants and Units

- In this course you will need constants of nature and also to deal a bit with units. The code-template has constants of nature included and by default this is in SI units (one could change this by adjusting the constants m,s,kg,K). See [src/Utils.h](#) This struct can also be used to set numerical settings like x-ranges, k-ranges etc. to use
- You can access the constants here anywhere in the template by simply writing e.g. [Constants.Mpc](#) to get a megaparsec in SI units (meter) or [Constants.hbar](#) to get Plancks constant (in units of Js).
- For example the Hubble parameter today (in units of 1/s) is then:
$$H0 = 100 * h * \text{Constants.km} / \text{Constants.s} / \text{Constants.Mpc};$$
or simply
$$H0 = \text{Constants.H0_over_h} * h;$$
- In future milestones you also have to make sure that the equations are in the right units (i.e. that constants like c, hbar, kb are restored). More info about units: https://cmb.wintherscoming.no/theory_units.php

```
// The constants used in this code. Everything is here in SI units
extern struct ConstantsAndUnits {
    // Basic units (here we use SI)
    const double m      = 1.0;           // Length (in meters)
    const double s      = 1.0;           // Time (in seconds)
    const double kg     = 1.0;           // Kilo (in kilos)
    const double K      = 1.0;           // Temperature (in Kelvins)

    // Derived units
    const double km     = 1e3 * m;       // Kilometers
    const double N      = kg*m/(s*s);    // Newton
    const double J      = N*m;           // Joule
    const double W      = J/s;           // Watt
    const double Mpc    = 3.08567758e22 * m; // Megaparsec
    const double eV     = 1.60217653e-19 * J; // Electronvolt

    // Physical constants
    const double k_b     = 1.38064852e-23 * J/K; // Boltzmanns constant
    const double m_e     = 9.10938356e-31 * kg; // Mass of electron
    const double m_H     = 1.6735575e-27 * kg; // Mass of hydrogen atom
    const double c       = 2.99792458e8 * m/s; // Speed of light
    const double G       = 6.67430e-11 * N*m*m/(kg*kg); // Gravitational constant
    const double hbar    = 1.054571817e-34 * J*s; // Reduced Plancks constant
    const double sigma_T = 6.6524587158e-29 * m*m; // Thomas scattering cross-section
    const double lambda_2s1s = 8.227 / s; // Transition time between 2s and 1s in Hydrogen
    const double H0_over_h = 100 * km/s/Mpc; // H0 / h
    const double epsilon_0 = 13.605693122994 * eV; // Ionization energy for the ground state of hydrogen
    const double xhi0     = 24.587387 * eV; // Ionization energy for neutral Helium
    const double xhi1     = 4.0 * epsilon_0; // Ionization energy for singly ionized Helium
```

Useful literature

- **Read through this paper:** <https://arxiv.org/pdf/astro-ph/0606683.pdf> It's low level and describes all the things you have to do! Good reference to have!
- More advanced paper describing all we are going to do in more detail: Ma & Bertschinger "Cosmological Perturbation Theory in the Synchronous and Conformal Newtonian Gauges" <https://arxiv.org/pdf/astro-ph/9506072v1.pdf>
- If you don't know C++ at all see https://www.w3schools.com/cpp/cpp_intro.asp for an introduction
- Good Luck!

How to calculate the CMB spectrum

Petter Callin¹

Department of Physics, University of Oslo, N-0316 Oslo, Norway

(Dated: June 28, 2006)

We present a self-contained description of everything needed to write a program that calculates the CMB power spectrum for the standard model of cosmology (Λ CDM). This includes the equations used, assumptions and approximations imposed on their solutions, and most importantly the algorithms and programming tricks needed to make the code actually work. The resulting program is compared to CMBFAST and typically agrees to within 0.1% – 0.4%. It includes both helium, reionization, neutrinos and the polarization power spectrum. The methods presented here could serve as a starting point for people wanting to write their own CMB program from scratch, for instance to look at more exotic cosmological models where CMBFAST or the other standard programs can't be used directly.

Cosmological Perturbation Theory in the Synchronous and Conformal Newtonian Gauges

Chung-Pei Ma¹

Theoretical Astrophysics 130-33, California Institute of Technology, Pasadena, CA 91125
and

Edmund Bertschinger²

Department of Physics, Massachusetts Institute of Technology, Cambridge, MA 02139