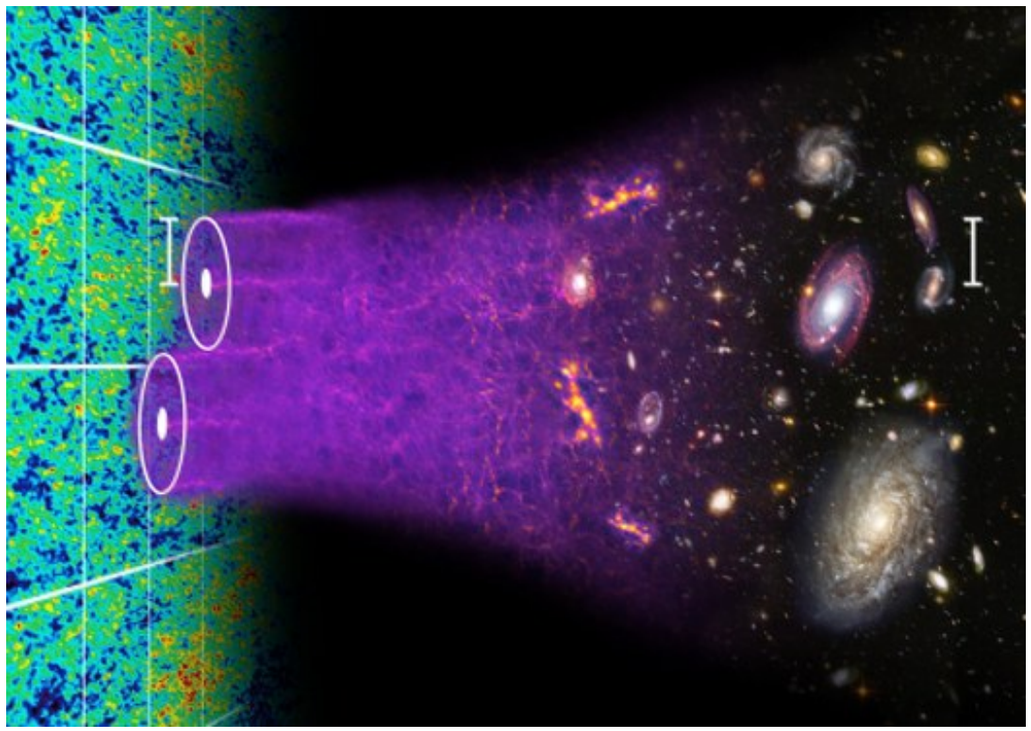
Numerical project
Milestone III

Return of the Einstein Tensor



AST5220 / AST9420 Spring 2021
Hans Winther

Overview

- We have derived the full set of Einstein-Boltzmann equations for how perturbations of the photons, baryons, CDM and the metric evolve.

- These depends on
  1. The background evolution (we have this from Milestone I)
  2. The recombination history (we have this from Milestone II)

- We know how to solve coupled ODEs and have what we need to be able to integrate these perturbations from the early Universe until today so we are all set. Doing this job is the main objective of Milestone III.

- This will then again be used in Milestone IV to finally compute directly observable quantities like the CMB angular power-spectrum and the matter power-spectrum

The Einstein-Boltzmann System

Continuity equations
Conservation of particle number

Gravitational Force

Pressure / momentum transfer from photons to baryons

For PhDs we also have equations for neutrinos and polarization

Poisson equation

Anisotropic stress

A complicated set of equations, but the underlying physics is simple:
pressure and expansion versus gravitational collapse
with a dash of interactions causing photons to drag baryons with them

The Einstein-Boltzmann System

Note: **every term that multiplies the perturbations are functions of time and scale that we know from the first two milestones**

Photon temperature multipoles:

$$\Theta_0' = -\frac{ck}{\mathcal{H}}\Theta_1 - \Phi',$$
$$\Theta_1' = \frac{ck}{3\mathcal{H}}\Theta_0 - \frac{2ck}{3\mathcal{H}}\Theta_2 + \frac{ck}{3\mathcal{H}}\Psi + \tau'\left[\Theta_1 + \frac{1}{3}v_b\right],$$
$$\Theta_\ell' = \frac{\ell ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell-1} - \frac{(\ell+1)ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell+1} + \tau'\left[\Theta_\ell - \frac{1}{10}\Pi\delta_{\ell,2}\right], \qquad 2 \leq \ell < \ell_{\rm max}$$
$$\Theta_\ell' = \frac{ck}{\mathcal{H}}\Theta_{\ell-1} - c\frac{\ell+1}{\mathcal{H}\eta(x)}\Theta_\ell + \tau'\Theta_\ell, \qquad \ell = \ell_{\rm max}$$

Cold dark matter and baryons:

$$\delta_{\rm CDM}' = \frac{ck}{\mathcal{H}}v_{\rm CDM} - 3\Phi'$$
$$v_{\rm CDM}' = -v_{\rm CDM} - \frac{ck}{\mathcal{H}}\Psi$$
$$\delta_b' = \frac{ck}{\mathcal{H}}v_b - 3\Phi'$$
$$v_b' = -v_b - \frac{ck}{\mathcal{H}}\Psi + \tau' R(3\Theta_1 + v_b)$$

Metric perturbations:

$$\Phi' = \Psi - \frac{c^2k^2}{3\mathcal{H}^2}\Phi + \frac{H_0^2}{2\mathcal{H}^2}\left[\Omega_{\rm CDM}a^{-1}\delta_{\rm CDM} + \Omega_b a^{-1}\delta_b + 4\Omega_r a^{-2}\Theta_0 + 4\Omega_\nu a^{-2}\mathcal{N}_0\right]$$
$$\Psi = -\Phi - \frac{12H_0^2}{c^2k^2a^2}\left[\Omega_r\Theta_2 + \Omega_\nu\mathcal{N}_2\right]$$

The Einstein-Boltzmann System

Note: every term that multiplies the perturbations are functions of time and scale that we know from the first two milestones

Background quantities

Recombination quantities

Photon temperature multipoles:

$$\Theta_0' = -\frac{ck}{\mathcal{H}}\Theta_1 - \Phi',$$

$$\Theta_1' = \frac{ck}{3\mathcal{H}}\Theta_0 - \frac{2ck}{3\mathcal{H}}\Theta_2 + \frac{ck}{3\mathcal{H}}\Psi + \tau'\left[\Theta_1 + \frac{1}{3}v_b\right],$$

$$\Theta_\ell' = \frac{\ell ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell-1} - \frac{(\ell+1)ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell+1} + \tau'\left[\Theta_\ell - \frac{1}{10}\Pi\delta_{\ell,2}\right], \qquad 2 \le \ell < \ell_{\max}$$

$$\Theta_\ell' = \frac{ck}{\mathcal{H}}\Theta_{\ell-1} - c\frac{\ell+1}{\mathcal{H}\eta(x)}\Theta_\ell + \tau'\Theta_\ell, \qquad \ell = \ell_{\max}$$

Cold dark matter and baryons:

$$\delta_{\rm CDM}' = \frac{ck}{\mathcal{H}}v_{\rm CDM} - 3\Phi'$$

$$v_{\rm CDM}' = -v_{\rm CDM} - \frac{ck}{\mathcal{H}}\Psi$$

$$\delta_b' = \frac{ck}{\mathcal{H}}v_b - 3\Phi'$$

$$v_b' = -v_b - \frac{ck}{\mathcal{H}}\Psi + \tau' R(3\Theta_1 + v_b)$$

Metric perturbations:

$$\Phi' = \Psi - \frac{c^2k^2}{3\mathcal{H}^2}\Phi + \frac{H_0^2}{2\mathcal{H}^2}\left[\Omega_{\rm CDM}a^{-1}\delta_{\rm CDM} + \Omega_b a^{-1}\delta_b + 4\Omega_r a^{-2}\Theta_0 + 4\Omega_\nu a^{-2}\mathcal{N}_0\right]$$

$$\Psi = -\Phi - \frac{12H_0^2}{c^2k^2a^2}\left[\Omega_r\Theta_2 + \Omega_\nu\mathcal{N}_2\right]$$

The Einstein-Boltzmann System

Photon temperature multipoles:

$$
\begin{aligned}
\Theta_0' &= -\frac{ck}{\mathcal{H}}\Theta_1 - \Phi', \\
\Theta_1' &= \frac{ck}{3\mathcal{H}}\Theta_0 - \frac{2ck}{3\mathcal{H}}\Theta_2 + \frac{ck}{3\mathcal{H}}\Psi + \tau'\left[\Theta_1 + \frac{1}{3}v_b\right], \\
\Theta_\ell' &= \frac{\ell ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell-1} - \frac{(\ell+1)ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell+1} + \tau'\left[\Theta_\ell - \frac{1}{10}\Pi\delta_{\ell,2}\right], \qquad 2 \leq \ell < \ell_{\max} \\
\Theta_\ell' &= \frac{ck}{\mathcal{H}}\Theta_{\ell-1} - c\frac{\ell+1}{\mathcal{H}\eta(x)}\Theta_\ell + \tau'\Theta_\ell, \qquad \ell = \ell_{\max}
\end{aligned}
$$

Cold dark matter and baryons:

$$
\begin{aligned}
\delta_{\rm CDM}' &= \frac{ck}{\mathcal{H}}v_{\rm CDM} - 3\Phi' \\
v_{\rm CDM}' &= -v_{\rm CDM} - \frac{ck}{\mathcal{H}}\Psi \\
\delta_b' &= \frac{ck}{\mathcal{H}}v_b - 3\Phi' \\
v_b' &= -v_b - \frac{ck}{\mathcal{H}}\Psi + \tau' R(3\Theta_1 + v_b)
\end{aligned}
$$

Metric perturbations:

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

Note: this potential does not need to be dynamically evolved - follows algebraically from other perturbations

This is a closed set of coupled first order equations that we can solve this with an ODE solver!

The ODE system

The ODE system can be written on the form

$$\frac{d\vec{Y}}{dx} = \vec{A}(\vec{y})$$

Where A is the “right hand side” that you need to implement

Each Y[i] represent one component and <u>you</u> need to specify what Y[i] means in your code and implement the equations that is consistent with this!

For example if we choose that:

$$Y[0] = \delta_{\mathrm{CDM}}$$
$$Y[1] = v_{\mathrm{CDM}}$$
$$Y[2] = \delta_{\mathrm{b}}$$
$$Y[3] = v_{\mathrm{b}}$$
$$Y[4] = \Phi$$
$$Y[5] = \Theta_0$$
$$Y[6] = \ldots$$

Then the equation:

$$\delta'_{\mathrm{CDM}} = \frac{ck}{\mathscr{H}} v_{\mathrm{CDM}} - 3\Phi'$$

Would become:

$$\frac{dY[0]}{dx} = \frac{ck}{\mathscr{H}} Y[1] - 3\frac{dY[4]}{dx}$$

Filling the vector dydx[i] is what you do in the “set right hand side” routines

(NB: for this particular one if you were to write it as above dydx[4] would have to be set first otherwise it would lead to nonsense)

One small problem: Tight coupling

- Unfortunately the equation system is numerically unstable in the early Universe due to the tight coupling between photons and baryons

$$\Theta_1' = \frac{ck}{3\mathscr{H}}\Theta_0 - \frac{2ck}{3\mathscr{H}}\Theta_2 + \frac{ck}{3\mathscr{H}}\Psi + \tau'\left[\Theta_1 + \frac{1}{3}v_b\right]$$

$$v_b' = -v_b - \frac{ck}{\mathscr{H}}\Psi + \tau' R(3\Theta_1 + v_b)$$

Huge

Tiny

- The high efficiency of Compton scattering early on washes out all higher moments and only the first 2 moments (0 = ‘Overdensity’ and 1 = ‘Velocity’) are really relevant. Forces the photon velocity to be equal to the baryon velocity: they move as one **single** fluid

- **Tiny** * **Huge** leads to problems! Small errors in the velocities becomes big errors in the equation. We must find a way of avoiding this

One small problem: Tight coupling

- **Solution:** in the tight coupling regime **we only include the first two photon multipoles 0 and 1** (and no polarisation multipoles, but all the rest of the perturbations) and use a more stable approximation for the ODE (see the website for a derivation)

$$q = \frac{-[(1-R)\tau' + (1+R)\tau''](3\Theta_1 + v_b) - \frac{ck}{\mathcal{H}}\Psi + (1-\frac{\mathcal{H}'}{\mathcal{H}})\frac{ck}{\mathcal{H}}(-\Theta_0 + 2\Theta_2) - \frac{ck}{\mathcal{H}}\Theta_0'}{(1+R)\tau' + \frac{\mathcal{H}'}{\mathcal{H}} - 1}$$

$$v_b' = \frac{1}{1+R}\left[-v_b - \frac{ck}{\mathcal{H}}\Psi + R(q + \frac{ck}{\mathcal{H}}(-\Theta_0 + 2\Theta_2) - \frac{ck}{\mathcal{H}}\Psi)\right]$$

$$\Theta_1' = \frac{1}{3}(q - v_b')$$

- Just a bit more complicated expressions, but not that bad (Here we also see why we needed to compute the derivatives of tau)

- Tight coupling is valid as long as we are before recombination and that

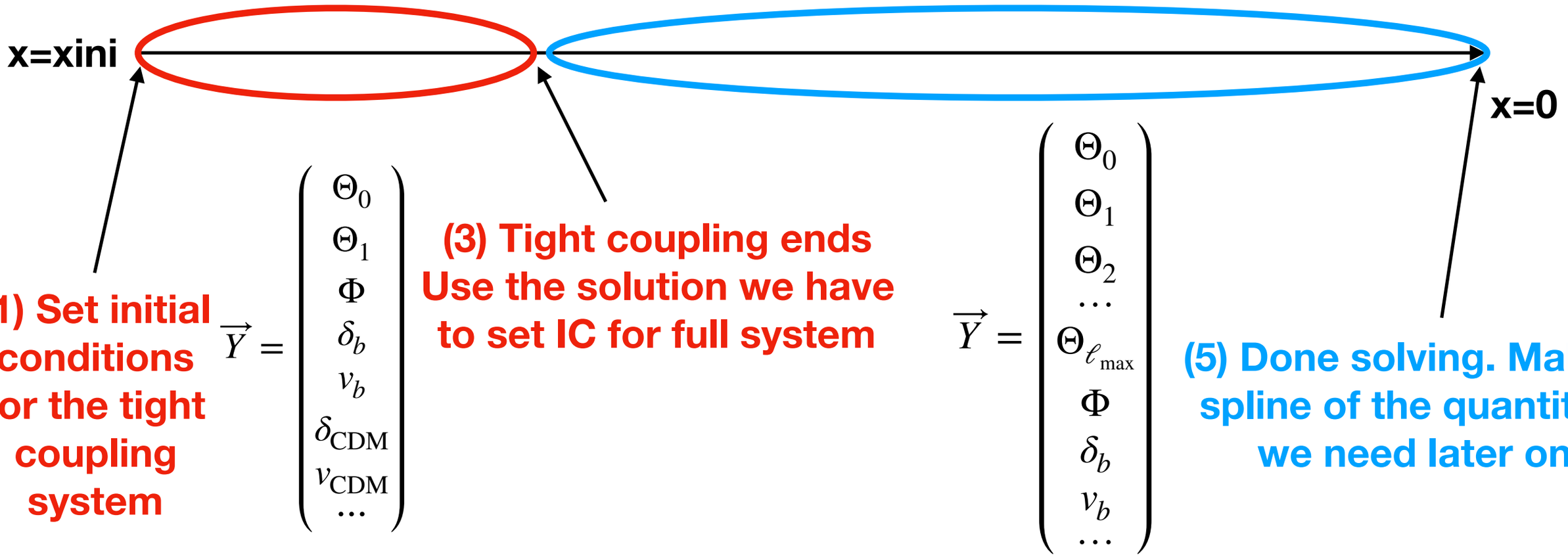
$$\left|\frac{d\tau}{dx}\right| < 10 \cdot \max(1, \frac{ck}{\mathcal{H}})$$

Summary of what to do

For every single wavenumber (and we need about 100 wave-numbers) we need to:

(2) Solve tight coupling system

(4) Solve full system



NB: what Y is in tight coupling is different from what Y is in the full system. You need to keep track of this!

Other things

- **Polarization and neutrinos:** For neutrinos we keep all the multipoles (~8) in both regimes. For polarisation we don't include any in tight coupling. If you don't include one or both of these then ignore these equations and put the multipoles to zero wherever they occur in other equations, i.e.

$$\Theta_\ell^P = 0 \qquad \mathcal{N}_\ell = 0$$

- **Truncation of the Boltzmann hierarchy:** Recall that the equation for a given ell depends on the (ell+1)th moment. Thus we need a way to truncate the hierarchy (see Callin or the website) and this leads to a special equation for the largest ell-value (thus you need to set 0,1 and ell_max by hand and in between we have a general formula).

$$\begin{aligned}
\Theta_0' &= -\frac{ck}{\mathcal{H}}\Theta_1 - \Phi', \\
\Theta_1' &= \frac{ck}{3\mathcal{H}}\Theta_0 - \frac{2ck}{3\mathcal{H}}\Theta_2 + \frac{ck}{3\mathcal{H}}\Psi + \tau'\left[\Theta_1 + \frac{1}{3}v_b\right], \\
\Theta_\ell' &= \frac{\ell ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell-1} - \frac{(\ell+1)ck}{(2\ell+1)\mathcal{H}}\Theta_{\ell+1} + \tau'\left[\Theta_\ell - \frac{1}{10}\Pi\delta_{\ell,2}\right], \qquad 2 \le \ell < \ell_{\max} \\
\Theta_\ell' &= \frac{ck}{\mathcal{H}}\Theta_{\ell-1} - c\frac{\ell+1}{\mathcal{H}\eta(x)}\Theta_\ell + \tau'\Theta_\ell, \qquad \ell = \ell_{\max}
\end{aligned}$$

- **Initial conditions:** See the website for a full list of the initial conditions. For setting IC after tight coupling we use the solution from tight coupling and for the quantities we didn't solve for in tight coupling but need for the full system we set these from the other multipoles, e.g. the IC says that
so we use this to set the IC for Theta2 from the value of Theta1 and (x, k).

$$\Theta_2 = -\frac{20ck}{45\mathcal{H}}\Theta_1$$

Code Template

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

Main

- Code runs from Main.cpp. From here we create the objects we need in this project, do the solving, output stuff etc.

- The perturbation object is defined in Perturbations.h and implemented in Perturbations.cpp

1) Create a perturbation object passing in the background and recombination history

2) Integrate perturbations and make splines

3) Output some data (in this example we output f(x,k) for k = 0.01 / Mpc)

```
//=============================================================================
// Module III
//=============================================================================

// Solve the perturbations
Perturbations pert(&cosmo, &rec);
pert.solve();
pert.info();

// Output perturbation quantities
double kvalue = 0.01 / Constants.Mpc;
pert.output(kvalue, "perturbations_k0.01.txt");

// Remove when module is completed
return 0;
```

Step 1: Initialization

- Method Perturbations::Perturbations in Perturbations.cpp

  This is the constructor that is run when the object is created. We don't need any special parameters here, just the previous objects (that holds all the parameters we need)

- All the background cosmology stuff are available via the **cosmo** pointer, e.g.
  double OmegaB = cosmo->get_OmegaB(x);
  gives you OmegaB(x)

- All recombination stuff available via the **rec** pointer, e.g.
  double tau = rec->tau_of_x(x);
  gives you tau(x).

- All the solving it done by running **solve()** on the object we have created

Objects that we take in

Store them in the class

```
//================================================
// Constructors
//================================================

Perturbations::Perturbations(
    BackgroundCosmology *cosmo,
    RecombinationHistory *rec) :
  cosmo(cosmo),
  rec(rec)
{}

//================================================
// Do all the solving
//================================================

void Perturbations::solve(){

  // Integrate all the perturbation equation and spline the resul
  integrate_perturbations();

  // Compute source functions and spline the result
  compute_source_functions();
}
```

Integrate perturbations

Compute source functions (for Milestone IV)

Step 2: Integrate perturbations

- Method Perturbations::integrate_perturbations() in Perturbations.cpp

  Integrate the perturbations for all the k-values we need

Make vector of all k's we need

$$\mathrm{e.g.}\ \ k_{\min} = 0.5/\eta_0,\ \ k_{\max} = 3000/\eta_0$$

Tight coupling (TC) regime:
1) Set IC
2) Compute xTC when TC ends
3) Integrate from start to xTC

The full regime:
1) Set IC from TC solution
2) Integrate from xTC till today

Make splines

```
//==========================================================
// The main work: integrate all the perturbations
// and spline the results
//==========================================================

void Perturbations::integrate_perturbations(){
  Utils::StartTiming("integrateperturbation");

  //===============================================================
  // TODO: Set up the k-array for the k's we are going to integrate over
  // Start at k_min end at k_max with n_k points with either a
  // quadratic or a logarithmic spacing
  //===============================================================
  Vector k_array(n_k);

  // Loop over all wavenumbers
  for(int ik = 0; ik < n_k; ik++){

      //===============================================================
      // TODO: Tight coupling integration
      // Remember to implement the routines:
      // set_ic : The IC at the start
      // rhs_tight_coupling_ode : The dydx for our coupled ODE system
      //===============================================================

      // Set up initial conditions in the tight coupling regime
      auto y_tight_coupling_ini = set_ic(x_start, k);

      // The tight coupling ODE system
      ODEFunction dydx_tight_coupling = [&](double x, const double *y, double *dydx){
        return rhs_tight_coupling_ode(x, k, y, dydx);
      };

      // Integrate from x_start -> x_end_tight

      //=========================================================
      // TODO: Full equation integration
      // Remember to implement the routines:
      // set_ic_after_tight_coupling : The IC after tight coupling ends
      // rhs_full_ode : The dydx for our coupled ODE system
      //=========================================================

      // Set up initial conditions (y_tight_coupling is the solution at the end of tight coupling)
      // auto y_full_ini = set_ic_after_tight_coupling(y_tight_coupling, x_end_tight, k);

      // The full ODE system
      ODEFunction dydx_full = [&](double x, const double *y, double *dydx){
        return rhs_full_ode(x, k, y, dydx);
      };

  //=============================================================================
  // TODO: Make all splines needed: Theta0,Theta1,Theta2,Phi,Psi,...
  //=============================================================================
  // ...
  // ...
  // ...
}
```

Loop over all k's

Step 3: Set IC (Tight coupling)

- Method Perturbations::set_ic(const double x, const double k) in Perturbations.cpp

  Set the IC at the start of the run

Example of an indexing scheme for what component of Ytc contains what quantity (do it however you want)

Fill the vector Ytc using the expressions for the initial conditions

```cpp
//================================================
// Set IC at the start of the run (this is in the
// tight coupling regime)
//================================================
Vector Perturbations::set_ic(const double x, const double k) const{

  // The vector we are going to fill
  Vector y_tc(Constants.n_ell_tot_tc);

  //========================================================================
  // Compute where in the y_tc array each component belongs
  // This is just an example of how to do it to make it easier
  // Feel free to organize the component any way you like
  //========================================================================

  // For integration of perturbations in tight coupling regime (Only 2 photon multipoles + neutrinos needed)
  const int n_ell_theta_tc      = Constants.n_ell_theta_tc;
  const int n_ell_neutrinos_tc  = Constants.n_ell_neutrinos_tc;
  const int n_ell_tot_tc        = Constants.n_ell_tot_tc;
  const bool polarization       = Constants.polarization;
  const bool neutrinos          = Constants.neutrinos;

  // References to the tight coupling quantities
  double &delta_cdm    =  y_tc[Constants.ind_deltacdm_tc];
  double &delta_b      =  y_tc[Constants.ind_deltab_tc];
  double &v_cdm        =  y_tc[Constants.ind_vcdm_tc];
  double &v_b          =  y_tc[Constants.ind_vb_tc];
  double &Phi          =  y_tc[Constants.ind_Phi_tc];
  double *Theta        = &y_tc[Constants.ind_start_theta_tc];
  double *Nu           = &y_tc[Constants.ind_start_nu_tc];

  //========================================================================
  // TODO: Set the initial conditions in the tight coupling regime
  //========================================================================
  // ...
  // ...

  // SET: Scalar quantities (Gravitational potential, baryons and CDM)
  // ...
  // ...

  // SET: Photon temperature perturbations (Theta_ell)
  // ...
  // ...

  // SET: Neutrino perturbations (N_ell)
  if(neutrinos){
    // ...
    // ...
  }

  return y_tc;
}
```

Step 4: Compute when tight coupling ends

- Method Perturbations::get_tight_coupling_time(const double k) in Perturbations.cpp

  Compute and return the x when tight coupling ends for a given k

$$\left|\frac{d\tau}{dx}\right| < 10 \cdot \max\left(1, \frac{ck}{\mathcal{H}}\right)$$

```cpp
//==================================================
// The time when tight coupling end
//==================================================

double Perturbations::get_tight_coupling_time(const double k) const{
  double x_tight_coupling_end = 0.0;

  //============================================================================
  // TODO: compute and return x for when tight coupling ends
  // Remember all the three conditions in Callin
  //============================================================================
  // ...
  // ...

  return x_tight_coupling_end;
}
```

Step 5: Set right hand side of ODE system (Tight coupling)

- Method Perturbations::rhs_tight_coupling_ode(double x, double k, const double *y, double *dydx) in Perturbations.cpp

  Fill the right hand side of the tight coupling ODE system

Example of an indexing scheme for what component of Ytc contains what quantity (do it however you want). Just be consistent!

Define the right hand side, i.e. set all the dydx[i]

```cpp
//====================================================
// The right hand side of the perturbations ODE
// in the tight coupling regime
//====================================================

// Derivatives in the tight coupling regime
int Perturbations::rhs_tight_coupling_ode(double x, double k, const double *y, double *dydx){

  //============================================================================
  // Compute where in the y / dydx array each component belongs
  // This is just an example of how to do it to make it easier
  // Feel free to organize the component any way you like
  //============================================================================

  // For integration of perturbations in tight coupling regime (Only 2 photon multipoles + neutrinos needed)
  const int n_ell_theta_tc      = Constants.n_ell_theta_tc;
  const int n_ell_neutrinos_tc  = Constants.n_ell_neutrinos_tc;
  const bool neutrinos          = Constants.neutrinos;

  // The different quantities in the y array
  const double &delta_cdm       =  y[Constants.ind_deltacdm_tc];
  const double &delta_b         =  y[Constants.ind_deltab_tc];
  const double &v_cdm           =  y[Constants.ind_vcdm_tc];
  const double &v_b             =  y[Constants.ind_vb_tc];
  const double &Phi             =  y[Constants.ind_Phi_tc];
  const double *Theta           = &y[Constants.ind_start_theta_tc];
  const double *Nu              = &y[Constants.ind_start_nu_tc];

  // References to the quantities we are going to set in the dydx array
  double &ddelta_cdmdx    =  dydx[Constants.ind_deltacdm_tc];
  double &ddelta_bdx      =  dydx[Constants.ind_deltab_tc];
  double &dv_cdmdx        =  dydx[Constants.ind_vcdm_tc];
  double &dv_bdx          =  dydx[Constants.ind_vb_tc];
  double &dPhidx          =  dydx[Constants.ind_Phi_tc];
  double *dThetadx        = &dydx[Constants.ind_start_theta_tc];
  double *dNudx           = &dydx[Constants.ind_start_nu_tc];

  //============================================================================
  // TODO: fill in the expressions for all the derivatives
  //============================================================================

  // SET: Scalar quantities (Phi, delta, v, ...)
  // ...
  // ...
  // ...

  // SET: Photon multipoles (Theta_ell)
  // ...
  // ...

  // SET: Neutrino mutlipoles (Nu_ell)
  if(neutrinos){
    // ...
    // ...
    // ...
  }

  return GSL_SUCCESS;
}
```

Step 6: Set IC (Full system)

- Method Perturbations::set_ic_after_tight_coupling (const Vector &y_tc, const double x, const double k) in Perturbations.cpp

  Set the IC after tight coupling ends. Some quantities you have from y_tc, the rest are set using the IC from the start of the run + y_tc quantities.

Example of an indexing scheme for what component of Y contains what quantity (do it however you want)

Fill the vector Y using the expressions for the initial conditions

```
Vector Perturbations::set_ic_after_tight_coupling(
    const Vector &y_tc,
    const double x,
    const double k) const{

  // Make the vector we are going to fill
  Vector y(Constants.n_ell_tot_full);

  //====================================================================
  // Compute where in the y array each component belongs and where corresponding
  // components are located in the y_tc array
  // This is just an example of how to do it to make it easier
  // Feel free to organize the component any way you like
  //====================================================================

  // Number of multipoles we have in the full regime
  const int n_ell_theta         = Constants.n_ell_theta;
  const int n_ell_thetap        = Constants.n_ell_thetap;
  const int n_ell_neutrinos     = Constants.n_ell_neutrinos;
  const bool polarization       = Constants.polarization;
  const bool neutrinos          = Constants.neutrinos;

  // Number of multipoles we have in the tight coupling regime
  const int n_ell_theta_tc      = Constants.n_ell_theta_tc;
  const int n_ell_neutrinos_tc  = Constants.n_ell_neutrinos_tc;

  // References to the tight coupling quantities
  const double &delta_cdm_tc    =  y_tc[Constants.ind_deltacdm_tc];
  const double &delta_b_tc      =  y_tc[Constants.ind_deltab_tc];
  const double &v_cdm_tc        =  y_tc[Constants.ind_vcdm_tc];
  const double &v_b_tc          =  y_tc[Constants.ind_vb_tc];
  const double &Phi_tc          =  y_tc[Constants.ind_Phi_tc];
  const double *Theta_tc        = &y_tc[Constants.ind_start_theta_tc];
  const double *Nu_tc           = &y_tc[Constants.ind_start_nu_tc];

  // References to the quantities we are going to set
  double &delta_cdm       =  y[Constants.ind_deltacdm_tc];
  double &delta_b         =  y[Constants.ind_deltab_tc];
  double &v_cdm           =  y[Constants.ind_vcdm_tc];
  double &v_b             =  y[Constants.ind_vb_tc];
  double &Phi             =  y[Constants.ind_Phi_tc];
  double *Theta           = &y[Constants.ind_start_theta_tc];
  double *Theta_p         = &y[Constants.ind_start_thetap_tc];
  double *Nu              = &y[Constants.ind_start_nu_tc];

  //====================================================================
  // TODO: fill in the initial conditions for the full equation system below
  // NB: remember that we have different number of multipoles in the two
  // regimes so be careful when assigning from the tc array
  //====================================================================
  // ...
  // ...
  // ...

  // SET: Scalar quantities (Gravitational potental, baryons and CDM)
  // ...
  // ...

  // SET: Photon temperature perturbations (Theta_ell)
  // ...
  // ...

  // SET: Photon polarization perturbations (Theta_p_ell)
  if(polarization){
    // ...
    // ...
  }
```

Step 7: Set right hand side of ODE system (Full system)

- Method Perturbations::rhs_full_ode(const double x, const double k) in Perturbations.cpp

  Fill the right hand side of the tight coupling ODE system.

Example of an indexing scheme for what component of Y contains what quantity (do it however you want)

Define the right hand side, i.e. set all the dydx[i]

```
//==============================================
// The right hand side of the full ODE
//==============================================

int Perturbations::rhs_full_ode(double x, double k, const double *y, double *dydx){

  //================================================================================
  // Compute where in the y / dydx array each component belongs
  // This is just an example of how to do it to make it easier
  // Feel free to organize the component any way you like
  //================================================================================

  // Index and number of the different quantities
  const int n_ell_theta         = Constants.n_ell_theta;
  const int n_ell_thetap        = Constants.n_ell_thetap;
  const int n_ell_neutrinos     = Constants.n_ell_neutrinos;
  const bool polarization       = Constants.polarization;
  const bool neutrinos          = Constants.neutrinos;

  // The different quantities in the y array
  const double &delta_cdm       =  y[Constants.ind_deltacdm];
  const double &delta_b         =  y[Constants.ind_deltab];
  const double &v_cdm           =  y[Constants.ind_vcdm];
  const double &v_b             =  y[Constants.ind_vb];
  const double &Phi             =  y[Constants.ind_Phi];
  const double *Theta           = &y[Constants.ind_start_theta];
  const double *Theta_p         = &y[Constants.ind_start_thetap];
  const double *Nu              = &y[Constants.ind_start_nu];

  // References to the quantities we are going to set in the dydx array
  double &ddelta_cdmdx    =  dydx[Constants.ind_deltacdm];
  double &ddelta_bdx      =  dydx[Constants.ind_deltab];
  double &dv_cdmdx        =  dydx[Constants.ind_vcdm];
  double &dv_bdx          =  dydx[Constants.ind_vb];
  double &dPhidx          =  dydx[Constants.ind_Phi];
  double *dThetadx        = &dydx[Constants.ind_start_theta];
  double *dTheta_pdx      = &dydx[Constants.ind_start_thetap];
  double *dNudx           = &dydx[Constants.ind_start_nu];

  // Cosmological parameters and variables
  // double Hp = cosmo->Hp_of_x(x);
  // ...

  // Recombination variables
  // ...

  //================================================================================
  // TODO: fill in the expressions for all the derivatives
  //================================================================================

  // SET: Scalar quantities (Phi, delta, v, ...)
  // ...
  // ...
  // ...

  // SET: Photon multipoles (Theta_ell)
  // ...
  // ...
  // ...

  // SET: Photon polarization multipoles (Theta_p_ell)
  if(polarization){
    // ...
```

Step 8: Collecting data and splining

- You will need to do a lot of book-keeping. I have added an indexing scheme if you want to use this, but if you feel you have more control using explicit numbers, e.g. [0] for deltaCDM, [1] for vCDM, etc. then use this. Just remember that Ytc and Y have different number of components so what is what might differ in the two regimes.

- You need to combine the data from tight coupling and the full system for each quantity in a 2D (x,k) vector and then spline it

- One way to do this is to:
  1) make an x-array from start till end.
  2) compute tight coupling time and figure out what index ‘i’ this corresponds to
  3) extract the 0->i components of the x-vector and pass that as your x-vector to the ODE solver.
  4) solve in TC, get the data and store it.
  5) Extract i-> N components and pass that to the full system.
  6) solve full system, get the data and store the data.

- Once you are done you need to make 2D splines

Example on how to store data in a 2D vector and making 2D spline

```
// Alternative way of making a spline: from a 2D vector z[ix][iy]
Vector2D zz_array(nx, Vector(ny));
for(int ix = 0; ix < nx; ix++)
  for(int iy = 0; iy < ny; iy++){
    zz_array[ix][iy] = function(x_array[ix], y_array[iy]);
  }
Spline2D  z_spline(x_array, y_array, zz_array, "Test 2D spline");
std::cout << std::setw(8) << z_spline(x,y) << " ";
```

Step 9: Compute source functions (needed for Milestone IV)

- Method Perturbations::compute_source_function() in Perturbations.cpp

  Source function(s) are functions needed in Milestone IV. This amounts to filling an array based on the perturbations you have computed and then splining it

- You are not required to do this now, but you will have to do it for Milestone IV so its a good idea to just to it now

Temperature

$$\tilde{S}(k,x) = \tilde{g}[\Theta_0 + \Psi + \frac{1}{4}\Pi] + e^{-\tau}[\Psi' - \Phi'] - \frac{1}{ck}\frac{d}{dx}(\mathcal{H}\tilde{g}v_b) + \frac{3}{4c^2k^2}\frac{d}{dx}[\mathcal{H}\frac{d}{dx}(\mathcal{H}\tilde{g}\Pi)]$$

Polarisation

$$\tilde{S}_E(k,x) = \frac{3\tilde{g}\Pi}{4k^2(\eta_0 - \eta)^2}, \quad \Pi = \Theta_2 + \Theta_0^P + \Theta_2^P$$

```cpp
//====================================================
// After integrsating the perturbation compute the
// source function(s)
//====================================================
void Perturbations::compute_source_functions(){
  Utils::StartTiming("source");

  //============================================================================
  // TODO: Make the x and k arrays to evaluate over and use to make the splines
  //============================================================================
  // ...
  // ...
  Vector k_array;
  Vector x_array;

  // Make storage for the source functions (in 1D array to be able to pass it to the spline)
  Vector ST_array(k_array.size() * x_array.size());
  Vector SE_array(k_array.size() * x_array.size());

  // Compute source functions
  for(auto ix = 0; ix < x_array.size(); ix++){
    const double x = x_array[ix];
    for(auto ik = 0; ik < k_array.size(); ik++){
      const double k = k_array[ik];

      // NB: This is the format the data needs to be stored
      // in a 1D array for the 2D spline routine source(ix,ik) -> S_array[ix + nx * ik]
      const int index = ix + n_x * ik;

      //============================================================================
      // TODO: Compute the source functions
      //============================================================================
      // Fetch all the things we need...
      // const double Hp       = cosmo->Hp_of_x(x);
      // const double tau      = rec->tau_of_x(x);
      // ...
      // ...

      // Temperatur source
      ST_array[index] = 0.0;

      // Polarization source
      if(Constants.polarization){
        SE_array[index] = 0.0;
      }
    }
  }

  // Spline the source functions
  ST_spline.create (x_array, k_array, ST_array, "Source_Temp_x_k");
  if(Constants.polarization){
    SE_spline.create (x_array, k_array, SE_array, "Source_Pol_x_k");
  }

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

Temperature

Polarisation

Step 10: Output and profit

- Method Perturbations::output(const double k, const std::string filename) in Perturbations.cpp

  Output data and make plots

- **One tip:** Don't try to do everything at once, but do it step by step.

- For example focus on a single k-value (which you can change from small to large when testing) and try to solve the tight coupling ODE. When this works (output, plot and compare) then solve the full system and when this works (output, plot and compare) you can add the loop over k and get everything.

- Compile often when coding!

```
//====================================================
// Output some results to file for a given value of k
//====================================================

void Perturbations::output(const double k, const std::string filename) const{
  std::ofstream fp(filename.c_str());
  const int npts = 5000;
  auto x_array = Utils::linspace(x_start, x_end, npts);
  auto print_data = [&] (const double x) {
    fp << x                 << " ";
    fp << get_Theta(x,k,0)  << " ";
    fp << get_Theta(x,k,1)  << " ";
    fp << get_Theta(x,k,2)  << " ";
    fp << get_Phi(x,k)      << " ";
    fp << get_Psi(x,k)      << " ";
    fp << get_Pi(x,k)       << " ";


    fp << "\n";
  };
  std::for_each(x_array.begin(), x_array.end(), print_data);
}
```

Good luck

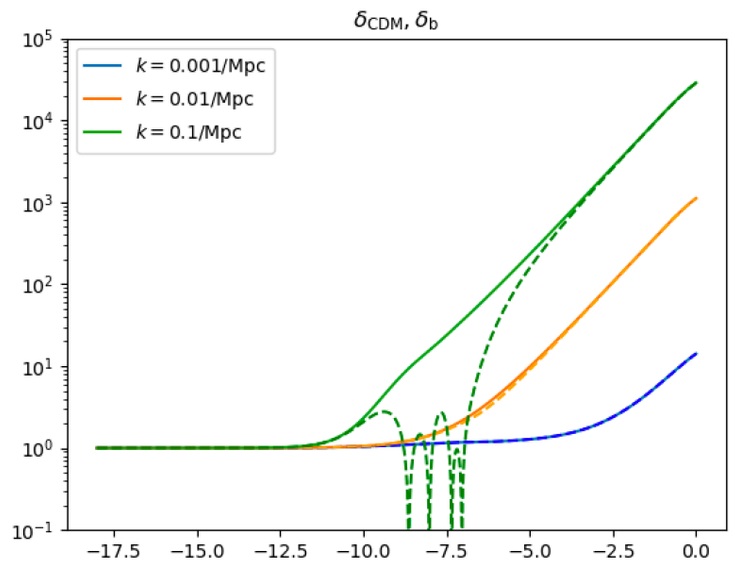
- What you are to do is straight forward on paper… but its still a bit of a mess and it's easy to do small mistakes which can be hard to figure out.

- A lot of equations that has to be set correctly and two different regimes so there are many things can go wrong when coding this up.

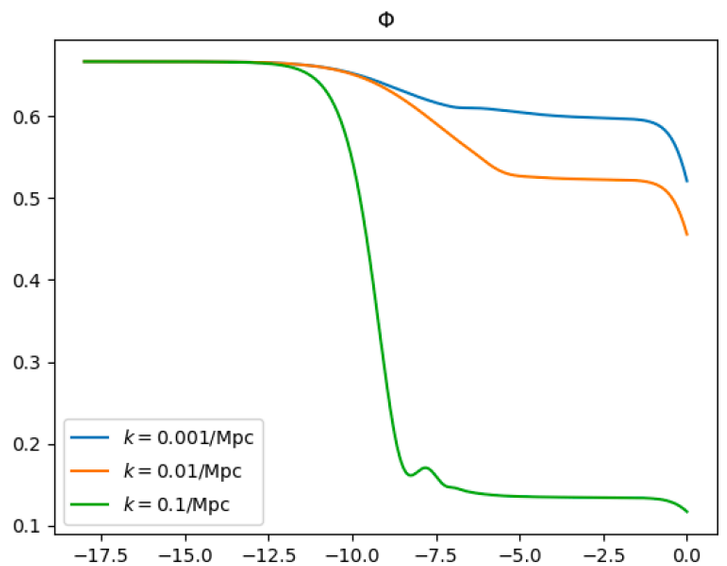
- Therefore **start early!** This milestone is going to to take some time to get correct so you don't want to have to deal with this the last week.

- In this milestone the physics enters much more than in the previous ones. You are expected to be able to describe and explain the results you got based on the physics we have been (and are going to go) through. Why do the perturbation evolve the way they do on different scales?

- That is what we will focus on in the lectures over the next month!



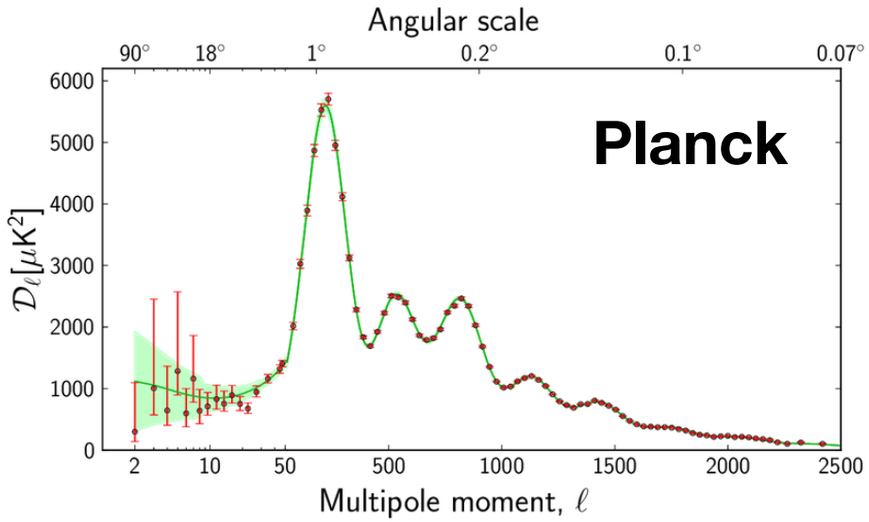
For example: baryon and CDM density.
Why do we see oscillations here?

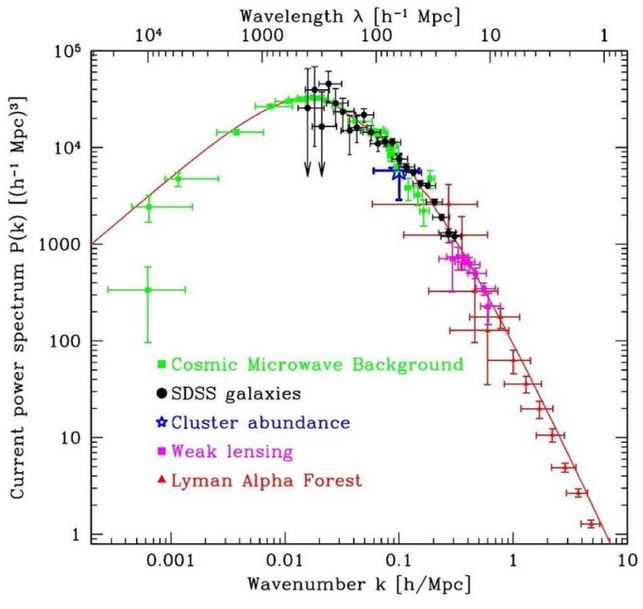


For example: metric potential.
Why does it decay at a certain time?
What determines this?

The next and final milestone

$$\vec{Y} = \begin{pmatrix} \Theta_0 \\ \Theta_1 \\ \Theta_2 \\ \cdots \\ \Theta_{\ell_{\max}} \\ \Phi \\ \delta_b \\ v_b \\ \cdots \end{pmatrix}$$





Galaxy surveys

With the perturbations in hand we are basically an 'integration' or two away from computing theory predictions that can be compared to observations!