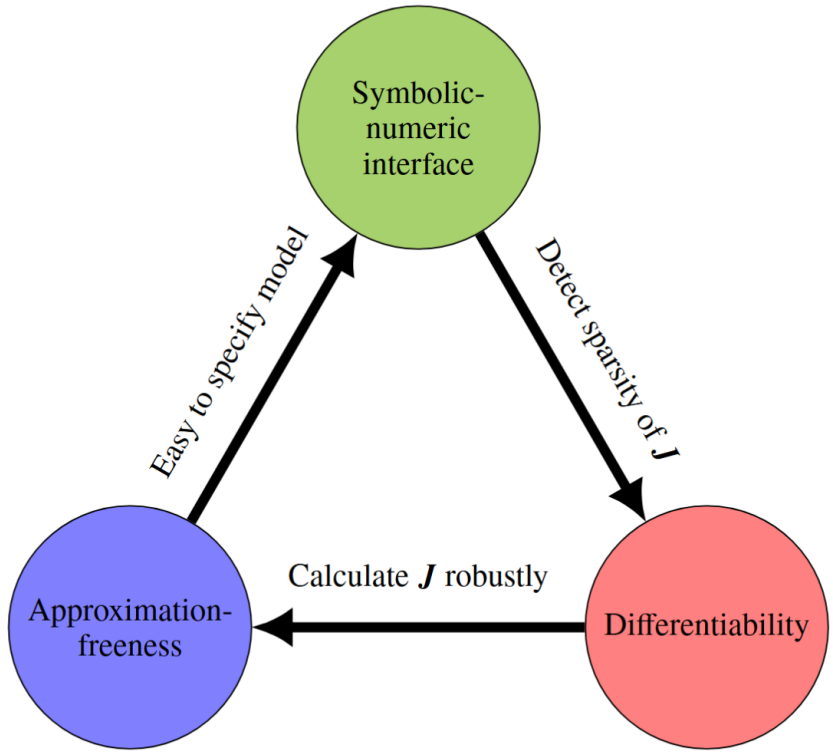
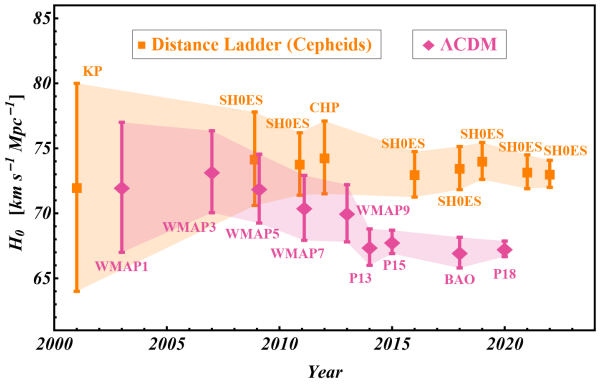
SymBoltz.jl: A symbolic-numeric, approximation-free and differentiable Einstein-Boltzmann code



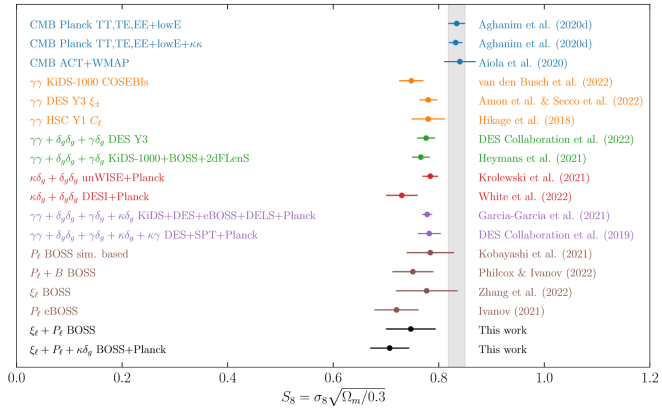
Herman Sletmoen (University of Oslo) / Prague 14.01.2026

Code & documentation & paper: github.com/hersle/SymBoltz.jl

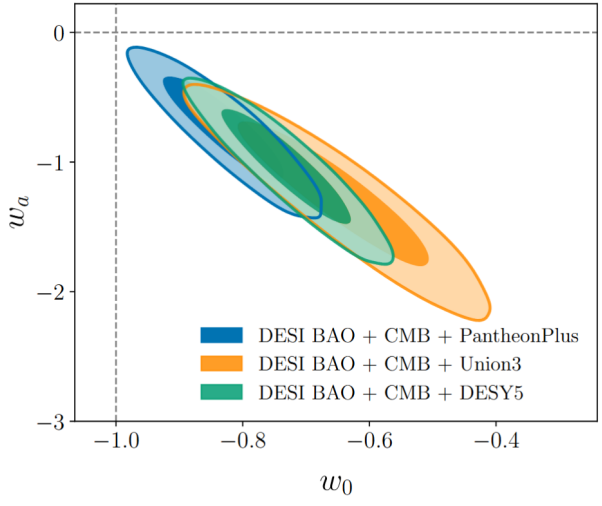
Tensions call for theoretical extensions to ΛCDM



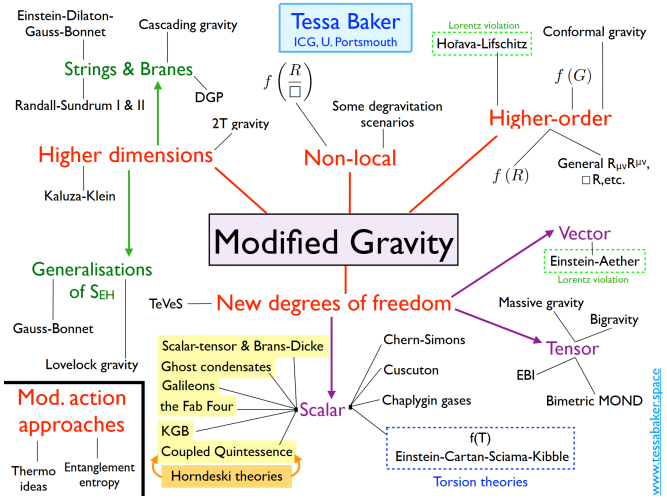
Hubble tension (2105.05208)



S_8 tension (2204.10392)



Dynamical DE? (2404.03002)

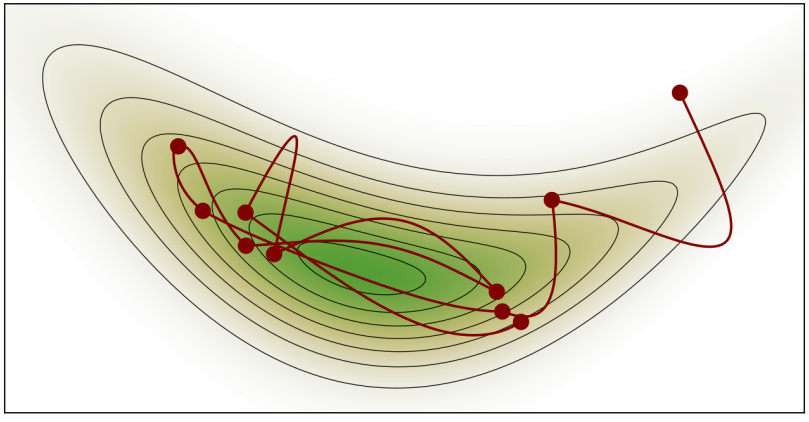


MG? (tessabaker.space/images/map_slide_v2.pdf)

- Want Boltzmann codes that are easy to extend

Next-generation surveys need differentiable predictions

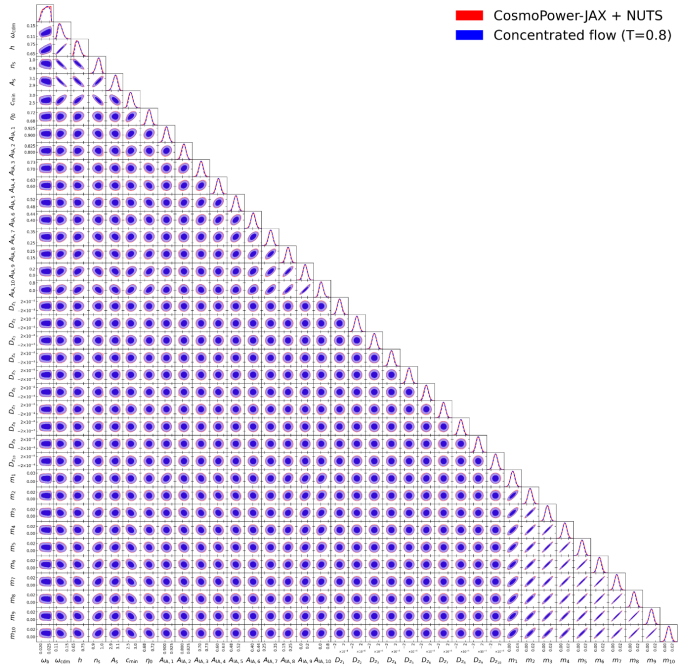
- More nuisance parameters
- Sample $O(100)$-dim. param. spaces
- Want gradient-based MCMC (e.g. HMC/NUTS $>$ MH)



Gradients push samplers in the right direction (like a ball rolling in a potential)

Two approaches:

- Emulators (of non-diff. codes)
- Differentiable codes



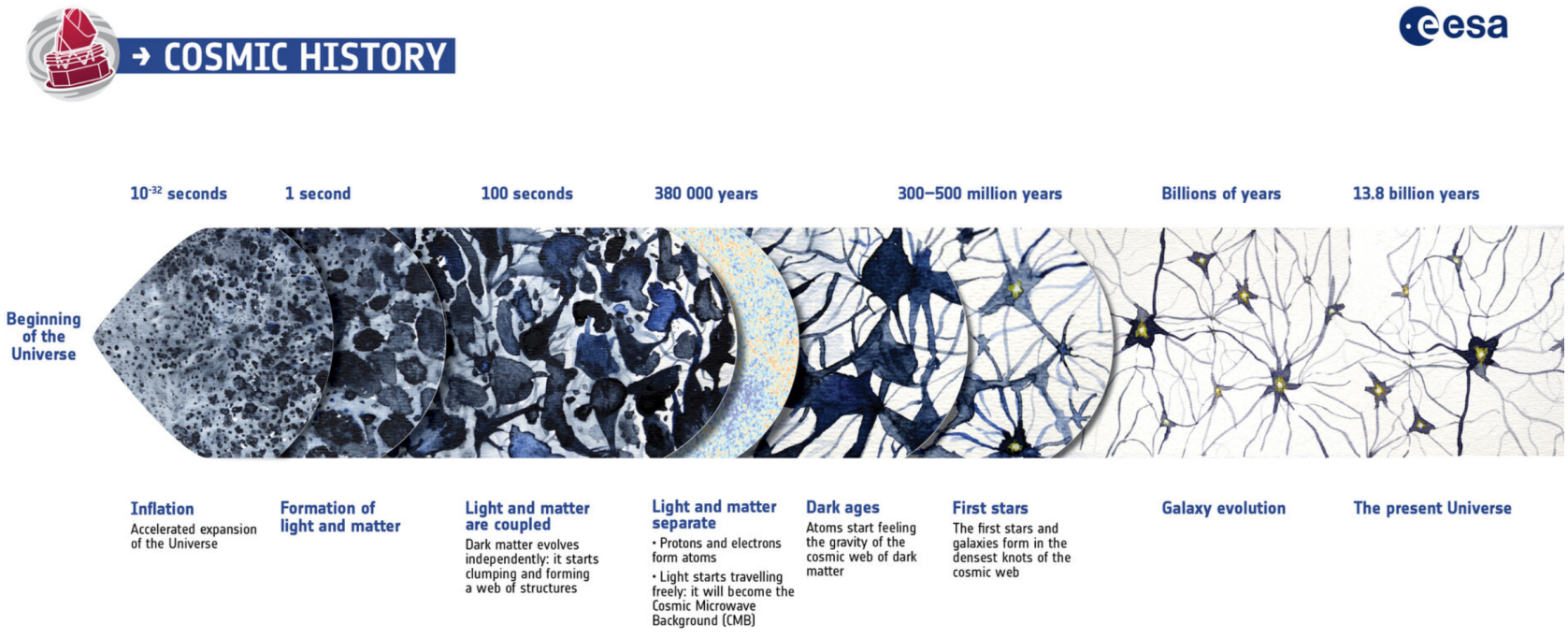
A 37-dim. param. space is now "small" (2405.12965)

History of Boltzmann codes

Year	Code	Lang.	New features
1995	COSMICS	Fortran	First proper treatment; seminal paper
1996	CMBFAST	Fortran	Line-of-sight integration (lower perturbations l_{max})
2000	CAMB 🏅	Fortran	Further development, non-flat models
2003	CMBEASY	C++	Object-oriented code structure
2011	CLASS 🏅	C	User-friendliness, flexibility, accuracy control, speed
2017	PyCosmo	Py/C++	Symbolic-numeric, code gen., approx.-free., sparsity
2021	Bolt	Julia	Differentiable, approx.-free
2024	DISCO-EB	Py/Jax	Differentiable, approx.-free, GPUs
2025	SymBoltz	Julia	Symbolic-numeric, approx.-free, differentiable

... and all forks thereof; e.g. HiCLASS, EFTCAMB, ...

What do Einstein-Boltzmann codes do? (less technical)



- Simulates a universe described by some cosmological model
- Perturbative around a homo. & iso. FLRW background
- Foundation for many cosmological CMB/LSS analyses

What do Einstein-Boltzmann codes do? (more technical)

1. Read input parameters Ω_{m0}, Ω_{b0}, $T_{\gamma 0}$, $N_{\rm eff}$, A_s, n_s, ...
2. Solve background and thermodynamics ODEs:

$$\left(\frac{\mathrm{d}a}{\mathrm{d}\tau}\right)^2 = \frac{8\pi}{3}\rho a^4, \quad \frac{\mathrm{d}\rho_s}{\mathrm{d}\tau} = -3\mathcal{H}(\rho_s + P_s), \quad \frac{\mathrm{d}x_H}{\mathrm{d}\tau} = aC\left[\beta(T_b)(1-x_H) - n_H\alpha^{(2)}(T_b)x_H^2\right], \quad ...$$

3. Solve perturbation ODEs (for several k):

$$\frac{\mathrm{d}\Phi}{\mathrm{d}\tau} = -\frac{4\pi}{3}a^2\mathcal{H}\delta\rho - \frac{k^2}{3\mathcal{H}}\Phi - \mathcal{H}\Psi, \quad \Phi - \Psi = \frac{12\pi a^2\Pi}{k^2},$$

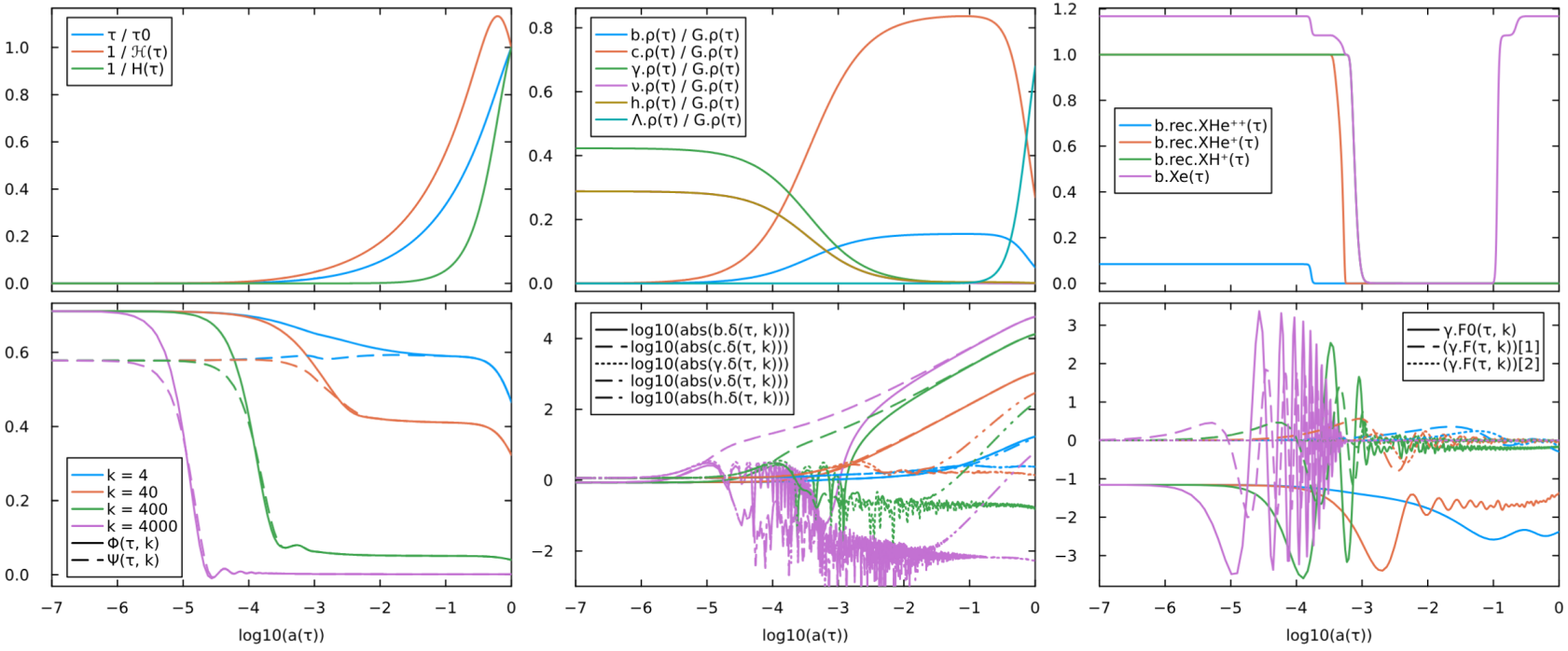
$$\frac{\mathrm{d}\delta_s}{\mathrm{d}\tau} = -\Big(1+w_s\Big)\Big(\theta_s - 3\frac{\mathrm{d}\Phi}{\mathrm{d}\tau}\Big) - 3\mathcal{H}(c_s^2 - w_s)\delta_s, \quad \frac{\mathrm{d}\theta_s}{\mathrm{d}\tau} = -\mathcal{H}(1-3w_s)\theta_s + \frac{c_s^2k^2\delta_s}{1+w_s} + k^2\Psi + ...$$

4. Solve line-of-sight integrals (for several k and l)

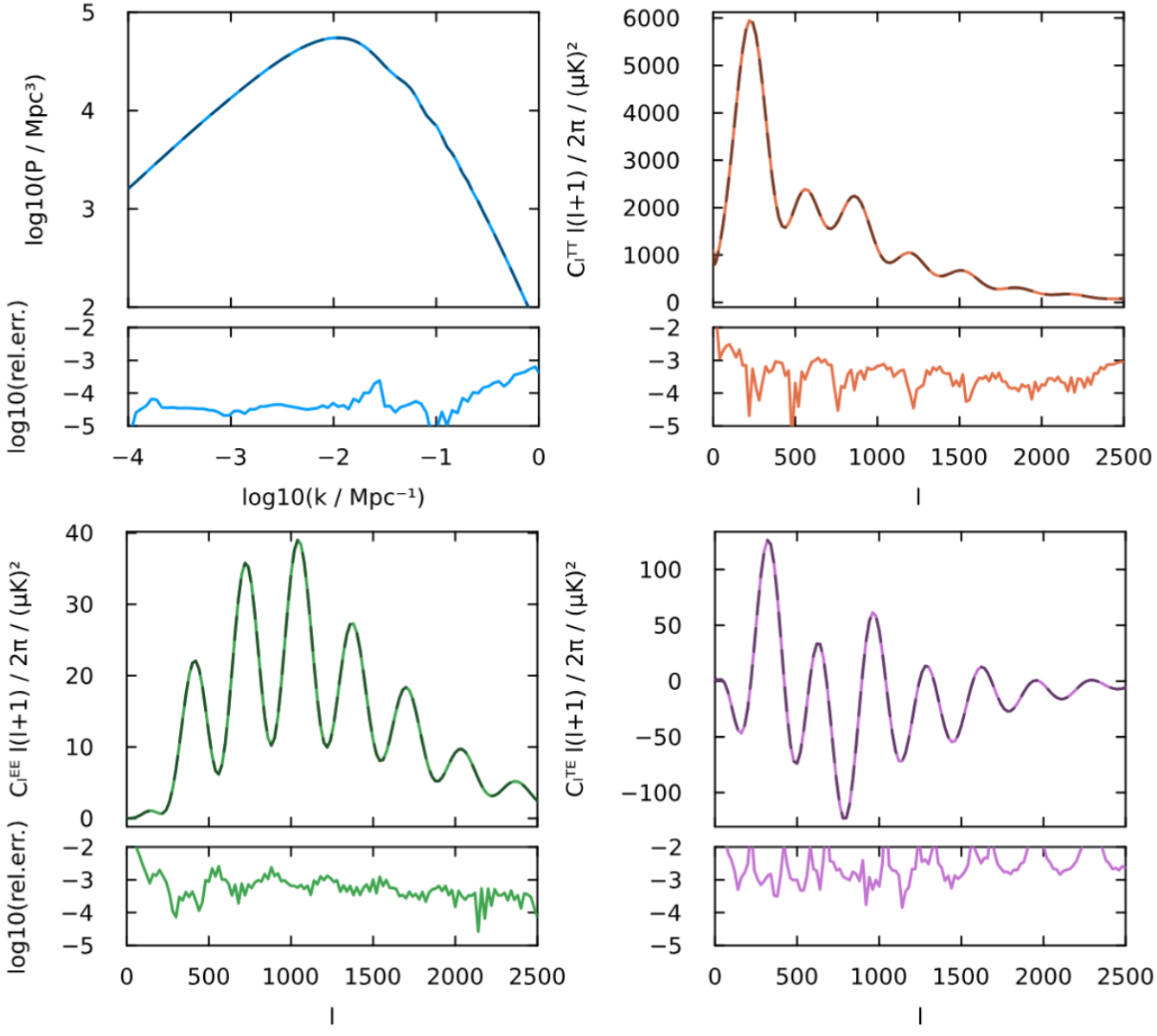
$$\frac{\mathrm{d}\Theta_l}{\mathrm{d}\tau} = \left[g\left(\Theta_0 + \Psi + \frac{\Pi}{4}\right) + \frac{gu_b}{k} + e^{-\tau}\frac{\mathrm{d}}{\mathrm{d}\tau}(\Psi - \Phi) + \frac{3}{4k^2}\frac{\mathrm{d}^2}{\mathrm{d}\tau^2}(g\Pi)\right] j_l((k(\tau_0 - \tau))$$

5. Output some function of the ODEs/integrals, like $P(k)$ or C_l.

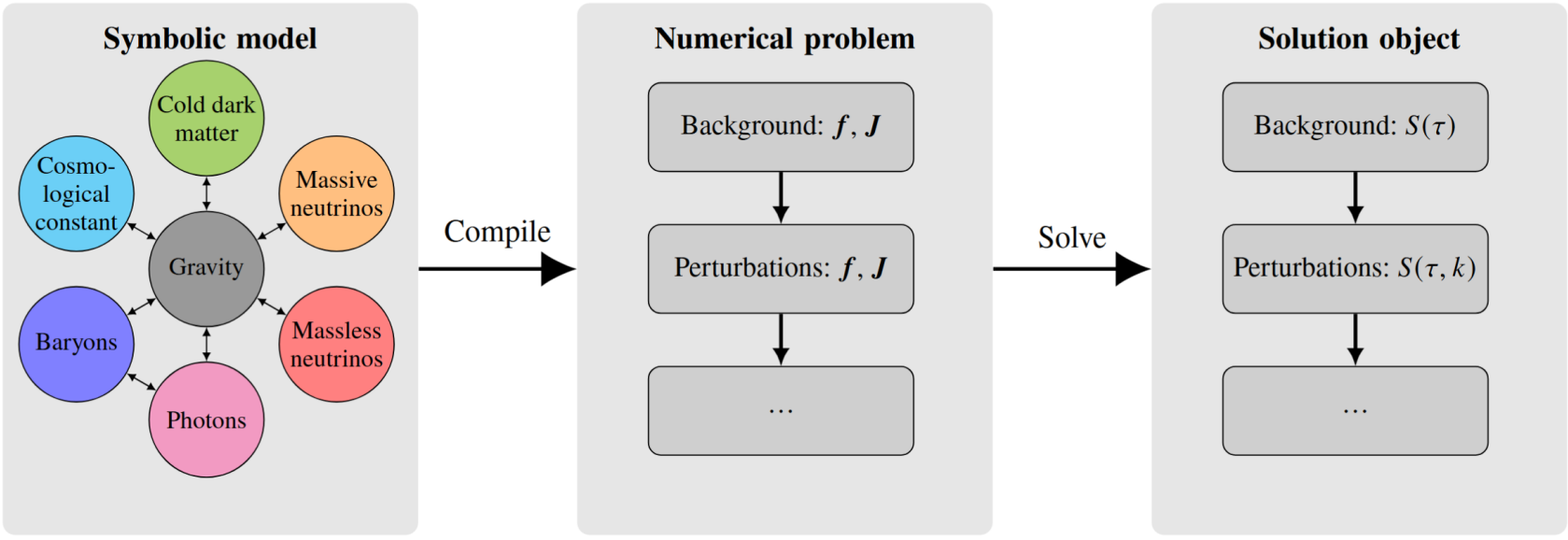
SymBoltz solves background and perturbations



Spectra agree with CLASS to around 0.1%



Feature 1: symbolic-numeric interface



- Programmatically analyze symbolic representation of all equations:

$$\frac{\mathrm{d}a(\tau)}{\mathrm{d}\tau} = (a(\tau))^2 \sqrt{\frac{8}{3}\rho(\tau)\pi}$$

$$\frac{\mathrm{d}}{\mathrm{d}\tau}\Phi(\tau,k) = \frac{-k^2\Phi(\tau,k)}{3\mathscr{H}(\tau)} + \frac{-\frac{4}{3}(a(\tau))^2\delta\rho(\tau,k)\pi}{\mathscr{H}(\tau)} - \mathscr{H}(\tau)\Psi(\tau,k)$$

$$k^2(\Phi(\tau,k) - \Psi(\tau,k)) = 12(a(\tau))^2\Pi(\tau,k)\pi$$

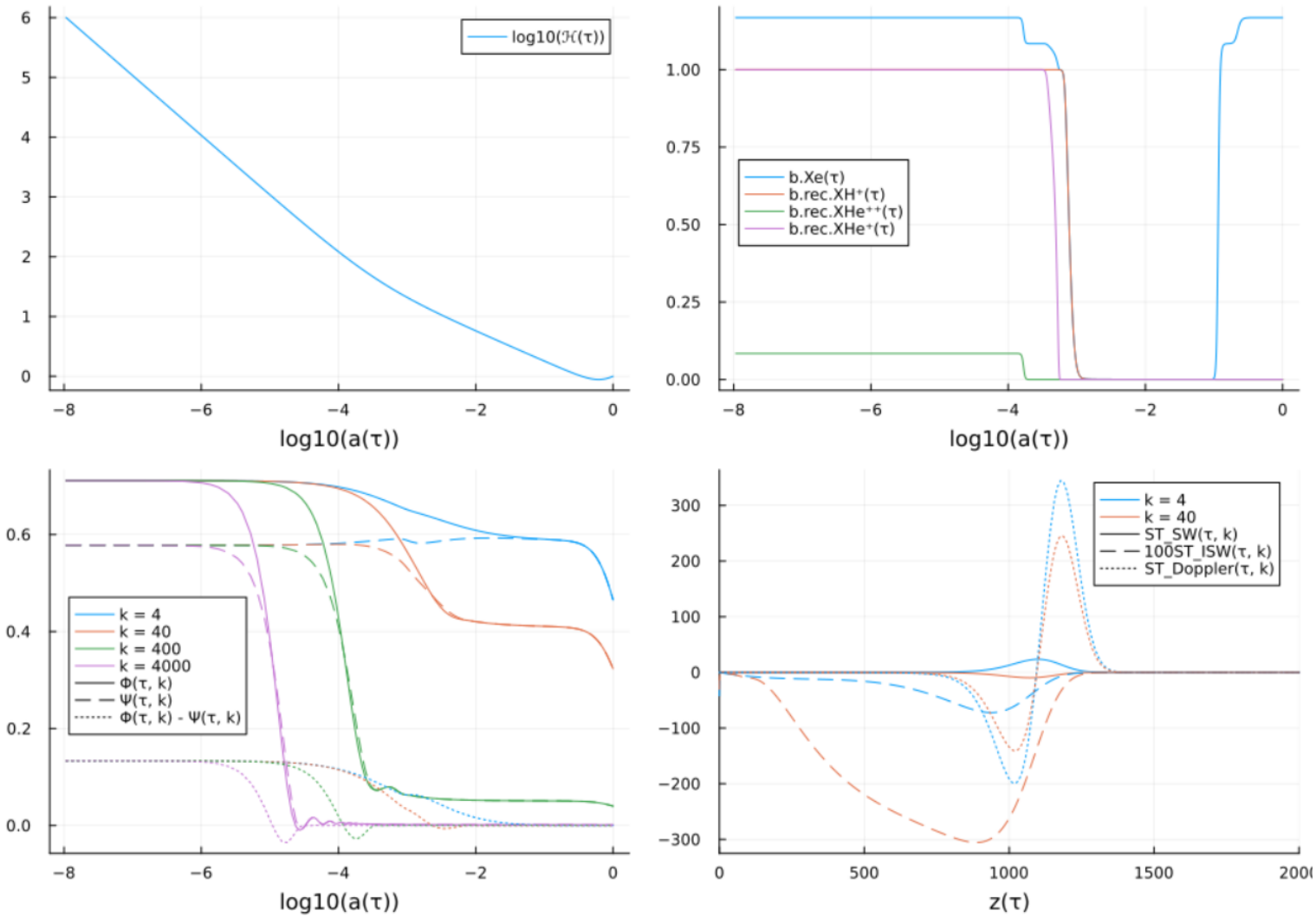
- Compile code for ODEs $\frac{\mathrm{d}\mathbf{u}}{\mathrm{d}\tau} = \mathbf{f}(\mathbf{u}, \mathbf{p}, \tau)$ and Jacobians $J_{ij} = \frac{\partial f_i}{\partial u_j}$
- Built on ModelingToolkit.jl (think SymPy, but more simulation-focused)
- **Goal:** maximize convenience/speed/stability with minimal user input

Interactive model-problem-solution workflow

```
# Load package
using SymBoltz

# 1) Define symbolic model
M = ΛCDM(lmax = 16)

# 2) Compile numerical problem
p = Dict(
 M.γ.T₀ ⇒ 2.7, M.b.Ω₀ ⇒ 0.05, M.b.YHe ⇒ 0.25,
 M.ν.Neff ⇒ 3.0, M.c.Ω₀ ⇒ 0.27, M.h.m_eV ⇒ 0.02,
 M.I.ln_As1e10 ⇒ 3.0, M.I.ns ⇒ 0.96, M.g.h ⇒ 0.7
)
prob = CosmologyProblem(M, p; jac = true, sparse = true)

# 3) Solve background and perturbations
ks = [4, 40, 400, 4000] # k / (H₀/c)
sol = solve(prob, ks)
```

Easily get/plot any expressions from the solution

```
p1 = plot(sol, log10(M.g.a), log10(M.g.ℋ))
p2 = plot(sol, log10(M.g.a), [M.b.Xe, M.b.rec.XH⁺, M.b.rec.XHe⁺⁺, M.b.rec.XHe⁺], legend_position = :left)
p3 = plot(sol, log10(M.g.a), [M.g.Φ, M.g.Ψ, M.g.Φ-M.g.Ψ], ks; legend_position = :left)
p4 = plot(sol, M.g.z, [M.ST_SW, 100*M.ST_ISW, M.ST_Doppler], ks[1:2]; xlims = (0, 2000), Nextra = 10)
plot(p1, p2, p3, p4, layout = (2, 2), size = (1000, 700))
```



Example: add $w_0 w_a$ dark energy

Implement equations from arXiv:1002.1311:

$$w = \frac{P}{\rho} = w_0 + w_a(1-a),$$

$$\frac{\mathrm{d}\rho}{\mathrm{d}\tau} = -3\mathcal{H}(\rho + P) \quad \rightarrow \quad \rho = \rho_0 a^{-3(1+w_0+w_a)} e^{-3w_a(1-a)},$$

$$c_a^2 = w - \frac{1}{3\mathcal{H}(1+w)}\frac{\mathrm{d}w}{\mathrm{d}\tau},$$

$$\frac{\mathrm{d}\delta}{\mathrm{d}\tau} = 3\mathcal{H}(w - c_s^2)\delta - \Big(1+w\Big)\Big(\Big(1+9\Big(\frac{\mathcal{H}}{k}\Big)^2(c_s^2 - c_a^2)\Big)\theta - 3\frac{\mathrm{d}\Phi}{\mathrm{d}\tau}\Big),$$

$$\frac{\mathrm{d}\theta}{\mathrm{d}\tau} = \mathcal{H}(3c_s^2 - 1)\theta + \frac{k^2 c_s^2 \delta}{1+w} + k^2\Psi,$$

$$\sigma = 0$$

Involved modification to CLASS I

Official advice: `grep -Rn _fld include/ source/ python/`

1. Read input parameters and handle parameter dependencies:

```
input.c:3177:  class_call(parser_read_double(pfc,"Omega_fld",&param2,&flag2,errmsg),
input.c:3186:             "'Omega_Lambda' or 'Omega_fld' must be left unspecified, except if 'Omega_scf' is set and
input.c:3189:             "You have entered 'Omega_scf' < 0 , so you have to specify both 'Omega_lambda' and 'Omega_
input.c:3215:    pba->Omega0_fld = param2;
input.c:3216:    Omega_tot += pba->Omega0_fld;
input.c:3232:    pba->Omega0_fld = 1. - pba->Omega0_k - Omega_tot;
input.c:3234:      printf(" -> matched budget equations by adjusting Omega_fld = %g\n",pba->Omega0_fld);
input.c:3248:  if (pba->Omega0_fld != 0.) {
input.c:3285:      class_read_double("w0_fld",pba->w0_fld);
input.c:3286:      class_read_double("wa_fld",pba->wa_fld);
input.c:3287:      class_read_double("cs2_fld",pba->cs2_fld);
input.c:3292:      class_read_double("w0_fld",pba->w0_fld);
input.c:3294:      class_read_double("cs2_fld",pba->cs2_fld);
```

2. Add parameter hooks to Python wrapper, too:

```
cclassy.pxd:91:        double Omega0_fld
cclassy.pxd:92:        double w0_fld
cclassy.pxd:93:        double wa_fld
cclassy.pxd:94:        double cs2_fld
```

Involved modification to CLASS II

3. Declare background variables and indices

```
background.h:104:  double Omega0_fld;       /**< \f$ \Omega_{0 de} \f$: fluid */
background.h:110:  double w0_fld;   /**< \f$ w0_{DE} \f$: current fluid equation of state parameter */
background.h:111:  double wa_fld;   /**< \f$ wa_{DE} \f$: fluid equation of state parameter derivative */
background.h:112:  double cs2_fld;  /**< \f$ c^2_{s~DE} \f$: sound speed of the fluid in the frame comoving with the
background.h:169:  int index_bg_rho_fld;       /**< fluid density */
background.h:170:  int index_bg_w_fld;         /**< fluid equation of state */
background.h:257:  int index_bi_rho_fld; /**< {B} fluid density */
background.h:289:  short has_fld;       /**< presence of fluid with constant w and cs2? */
background.h:416:  int background_w_fld(
background.h:419:                       double * w_fld,
background.h:420:                       double * dw_over_da_fld,
background.h:421:                       double * integral_fld);
```

4. Compute background

```
background.c:398:  double w_fld, dw_over_da, integral_fld;
background.c:540:  if (pba->has_fld == _TRUE_) {
background.c:542:    /* get rho_fld from vector of integrated variables */
background.c:543:    pvecback[pba->index_bg_rho_fld] = pvecback_B[pba->index_bi_rho_fld];
background.c:545:    /* get w_fld from dedicated function */
background.c:546:    class_call(background_w_fld(pba,a,&w_fld,&dw_over_da,&integral_fld), pba->error_message, pba->e
background.c:547:    pvecback[pba->index_bg_w_fld] = w_fld;
background.c:550:    // pvecback[pba->index_bg_rho_fld] = pba->Omega0_fld * pow(pba->H0,2) / pow(a,3.*(1.+pba->w0_fl
1.));
background.c:551:    // But now everthing is integrated numerically for a given w_fld(a) defined in the function bac
background.c:553:    rho_tot += pvecback[pba->index_bg_rho_fld];
background.c:554:    p_tot += w_fld * pvecback[pba->index_bg_rho_fld];
background.c:555:    dp_dloga += (a*dw_over_da-3*(1+w_fld)*w_fld)*pvecback[pba->index_bg_rho_fld];
background.c:664:int background_w_fld(
background.c:667:                      double * w_fld,
```

Involved modification to CLASS III

```
background.c:668:                    double * dw_over_da_fld,
background.c:669:                    double * integral_fld
background.c:680:    *w_fld = pba->w0_fld + pba->wa_fld * (1. - a);
background.c:715:    *dw_over_da_fld = - pba->wa_fld;
background.c:738:    *integral_fld = 3.*((1.+pba->w0_fld+pba->wa_fld)*log(1./a) + pba->wa_fld*(a-1.));
background.c:985:  pba->has_fld = _FALSE_;
background.c:1012:  if (pba->Omega0_fld != 0.)
background.c:1013:    pba->has_fld = _TRUE_;
background.c:1080:  class_define_index(pba->index_bg_rho_fld,pba->has_fld,index_bg,1);
background.c:1081:  class_define_index(pba->index_bg_w_fld,pba->has_fld,index_bg,1);
background.c:1166:  class_define_index(pba->index_bi_rho_fld,pba->has_fld,index_bi,1);
background.c:1744:  double w_fld, dw_over_da, integral_fld;
background.c:1778:  if (pba->has_fld == _TRUE_) {
background.c:1780:    class_call(background_w_fld(pba,0.,&w_fld,&dw_over_da,&integral_fld), pba->error_message, pba-
background.c:1782:    class_test(w_fld >= 1./3.,
background.c:1785:               w_fld);
background.c:2150:  double rho_fld_today;
background.c:2151:  double w_fld,dw_over_da_fld,integral_fld;
background.c:2240:  if (pba->has_fld == _TRUE_) {
background.c:2242:    /* rho_fld today */
background.c:2243:    rho_fld_today = pba->Omega0_fld * pow(pba->H0,2);
background.c:2245:    /* integrate rho_fld(a) from a_ini to a_0, to get rho_fld(a_ini) given rho_fld(a0) */
background.c:2246:    class_call(background_w_fld(pba,a,&w_fld,&dw_over_da_fld,&integral_fld), pba->error_message, p
background.c:2254:    /* rho_fld at initial time */
background.c:2255:    pvecback_integration[pba->index_bi_rho_fld] = rho_fld_today * exp(integral_fld);
background.c:2453:  class_store_columntitle(titles,"(.)rho_fld",pba->has_fld);
background.c:2454:  class_store_columntitle(titles,"(.)w_fld",pba->has_fld);
background.c:2526:    class_store_double(dataptr,pvecback[pba->index_bg_rho_fld],pba->has_fld,storeidx);
background.c:2527:    class_store_double(dataptr,pvecback[pba->index_bg_w_fld],pba->has_fld,storeidx);
background.c:2652:  if (pba->has_fld == _TRUE_) {
background.c:2654:    dy[pba->index_bi_rho_fld] = -3.*(1.+pvecback[pba->index_bg_w_fld])*y[pba->index_bi_rho_fld];
```

Involved modification to CLASS IV

5. Declare perturbation variables

```
perturbations.h:247:  short has_source_delta_fld;  /**< do we need source for delta of dark energy? */
perturbations.h:261:  short has_source_theta_fld;  /**< do we need source for theta of dark energy? */
perturbations.h:294:  int index_tp_delta_fld;  /**< index value for delta of dark energy */
perturbations.h:310:  int index_tp_theta_fld;   /**< index value for theta of dark energy */
perturbations.h:478:  int index_pt_delta_fld;  /**< dark energy density in true fluid case */
perturbations.h:479:  int index_pt_theta_fld;  /**< dark energy velocity in true fluid case */
```

6. Compute perturbations

```
perturbations.c:472:          class_store_double(dataptr,tk[ppt->index_tp_delta_fld],ppt->has_source_delta_fld,store
perturbations.c:501:          class_store_double(dataptr,tk[ppt->index_tp_theta_fld],ppt->has_source_theta_fld,store
perturbations.c:560:      class_store_columntitle(titles,"d_fld",pba->has_fld);
perturbations.c:589:      class_store_columntitle(titles,"t_fld",pba->has_fld);
perturbations.c:712:  double w_fld_ini, w_fld_0,dw_over_da_fld,integral_fld;
perturbations.c:1187:  ppt->has_source_delta_fld = _FALSE_;
perturbations.c:1202:  ppt->has_source_theta_fld = _FALSE_;
perturbations.c:1294:        if (pba->has_fld == _TRUE_)
perturbations.c:1295:          ppt->has_source_delta_fld = _TRUE_;
perturbations.c:1325:        if (pba->has_fld == _TRUE_)
perturbations.c:1326:          ppt->has_source_theta_fld = _TRUE_;
perturbations.c:1401:      class_define_index(ppt->index_tp_delta_fld,  ppt->has_source_delta_fld, index_type,1);
perturbations.c:1415:      class_define_index(ppt->index_tp_theta_fld,  ppt->has_source_theta_fld, index_type,1);
perturbations.c:3360:      class_store_columntitle(ppt->scalar_titles, "delta_rho_fld", pba->has_fld);
perturbations.c:3361:      class_store_columntitle(ppt->scalar_titles, "rho_plus_p_theta_fld", pba->has_fld);
perturbations.c:3362:      class_store_columntitle(ppt->scalar_titles, "delta_p_fld", pba->has_fld);
perturbations.c:3941:      class_define_index(ppv->index_pt_delta_fld,pba->has_fld,index_pt,1); /* fluid density */
perturbations.c:3942:      class_define_index(ppv->index_pt_theta_fld,pba->has_fld,index_pt,1); /* fluid velocity */
perturbations.c:4402:      if (pba->has_fld == _TRUE_) {
perturbations.c:4405:          ppv->y[ppv->index_pt_delta_fld] =
perturbations.c:4406:            ppw->pv->y[ppw->pv->index_pt_delta_fld];
```

Involved modification to CLASS V

```
perturbations.c:4408:          ppv->y[ppv->index_pt_theta_fld] =
perturbations.c:4409:            ppw->pv->y[ppw->pv->index_pt_theta_fld];
perturbations.c:5281:  double w_fld,dw_over_da_fld,integral_fld;
perturbations.c:5458:      if (pba->has_fld == _TRUE_) {
perturbations.c:5460:        class_call(background_w_fld(pba,a,&w_fld,&dw_over_da_fld,&integral_fld), pba->error_mes
perturbations.c:5463:          ppw->pv->y[ppw->pv->index_pt_delta_fld] = - ktau_two/4.*(1.+w_fld)*(4.-
3.*pba->cs2_fld)/(4.-6.*w_fld+3.*pba->cs2_fld) * ppr->curvature_ini * s2_squared; /* from 1004.5509 */ //TBC: curvat
perturbations.c:5465:          ppw->pv->y[ppw->pv->index_pt_theta_fld] = - k*ktau_three/4.*pba->cs2_fld/(4.-
6.*w_fld+3.*pba->cs2_fld) * ppr->curvature_ini * s2_squared; /* from 1004.5509 */ //TBC:curvature
perturbations.c:5740:      if ((pba->has_fld == _TRUE_) && (pba->use_ppf == _FALSE_)) {
perturbations.c:5742:        class_call(background_w_fld(pba,a,&w_fld,&dw_over_da_fld,&integral_fld), pba->error_mes
perturbations.c:5744:        ppw->pv->y[ppw->pv->index_pt_delta_fld] -= 3*(1.+w_fld)*a_prime_over_a*alpha;
perturbations.c:5745:        ppw->pv->y[ppw->pv->index_pt_theta_fld] += k*k*alpha;
perturbations.c:6719:  double w_fld,dw_over_da_fld,integral_fld;
perturbations.c:6727:  double w_prime_fld, ca2_fld;
perturbations.c:6730:  double rho_fld, p_fld, rho_fld_prime, p_fld_prime;
perturbations.c:6732:  double Gamma_fld, S, S_prime, theta_t, theta_t_prime, rho_plus_p_theta_fld_prime;
perturbations.c:7109:    if (pba->has_fld == _TRUE_) {
perturbations.c:7111:      class_call(background_w_fld(pba,a,&w_fld,&dw_over_da_fld,&integral_fld), pba->error_messa
perturbations.c:7112:      w_prime_fld = dw_over_da_fld * a_prime_over_a * a;
perturbations.c:7115:        ppw->delta_rho_fld = ppw->pvecback[pba->index_bg_rho_fld]*y[ppw->pv->index_pt_delta_flo
perturbations.c:7116:        ppw->rho_plus_p_theta_fld = (1.+w_fld)*ppw->pvecback[pba->index_bg_rho_fld]*y[ppw->pv->
perturbations.c:7117:        ca2_fld = w_fld - w_prime_fld / 3. / (1.+w_fld) / a_prime_over_a;
perturbations.c:7119:        ppw->delta_p_fld = pba->cs2_fld * ppw->delta_rho_fld + (pba->cs2_fld-
ca2_fld)*(3*a_prime_over_a*ppw->rho_plus_p_theta_fld/k/k);
perturbations.c:7387:  double w_fld,dw_over_da_fld,integral_fld;
perturbations.c:7787:    /* delta_fld */
perturbations.c:7788:    if (ppt->has_source_delta_fld == _TRUE_) {
perturbations.c:7789:      _set_source_(ppt->index_tp_delta_fld) = ppw->delta_rho_fld/pvecback[pba->index_bg_rho_flo
perturbations.c:7790:        + 3.*a_prime_over_a*(1.+pvecback[pba->index_bg_w_fld])*theta_over_k2; // N-
body gauge correction
perturbations.c:7903:    /* theta_fld */
perturbations.c:7904:    if (ppt->has_source_theta_fld == _TRUE_) {
```

Involved modification to CLASS VI

```
perturbations.c:7906:      class_call(background_w_fld(pba,a,&w_fld,&dw_over_da_fld,&integral_fld), pba->error_messa
perturbations.c:7908:      _set_source_(ppt->index_tp_theta_fld) = ppw->rho_plus_p_theta_fld/(1.+w_fld)/pvecback[pba
perturbations.c:8472:    class_store_double(dataptr, ppw->delta_rho_fld, pba->has_fld, storeidx);
perturbations.c:8473:    class_store_double(dataptr, ppw->rho_plus_p_theta_fld, pba->has_fld, storeidx);
perturbations.c:8474:    class_store_double(dataptr, ppw->delta_p_fld, pba->has_fld, storeidx);
perturbations.c:8683:  double w_fld,dw_over_da_fld,w_prime_fld,integral_fld;
perturbations.c:9269:    if (pba->has_fld == _TRUE_) {
perturbations.c:9276:        class_call(background_w_fld(pba,a,&w_fld,&dw_over_da_fld,&integral_fld), pba->error_mes
perturbations.c:9277:        w_prime_fld = dw_over_da_fld * a_prime_over_a * a;
perturbations.c:9279:        ca2 = w_fld - w_prime_fld / 3. / (1.+w_fld) / a_prime_over_a;
perturbations.c:9280:        cs2 = pba->cs2_fld;
perturbations.c:9284:        dy[pv->index_pt_delta_fld] =
perturbations.c:9285:          -(1+w_fld)*(y[pv->index_pt_theta_fld]+metric_continuity)
perturbations.c:9286:          -3.*(cs2-w_fld)*a_prime_over_a*y[pv->index_pt_delta_fld]
perturbations.c:9287:          -9.*(1+w_fld)*(cs2-ca2)*a_prime_over_a*a_prime_over_a*y[pv->index_pt_theta_fld]/k2;
perturbations.c:9291:        dy[pv->index_pt_theta_fld] = /* fluid velocity */
perturbations.c:9292:          -(1.-3.*cs2)*a_prime_over_a*y[pv->index_pt_theta_fld]
perturbations.c:9293:          +cs2*k2/(1.+w_fld)*y[pv->index_pt_delta_fld]
```

- Things related to one species scattered in many places
- A lot of boilerplate code for mechanical tasks $\rightarrow$ unnecessary
- In CLASS' defence, it handles more general $w(a)$ models

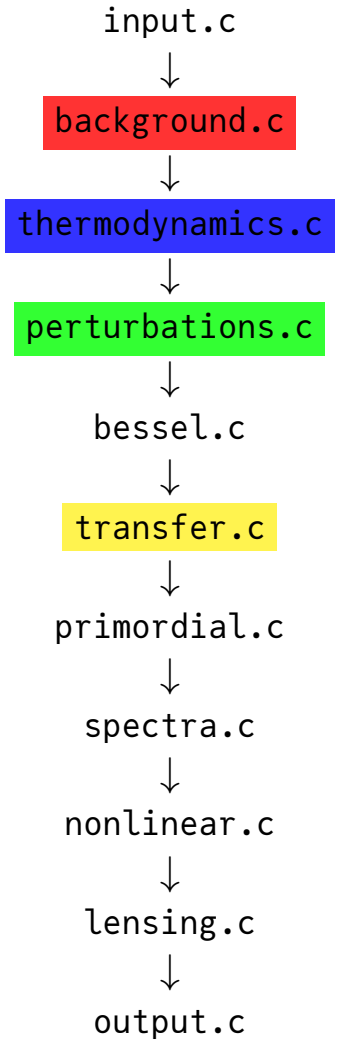
Short modification to SymBoltz

```
# 1) Create w0wa species "X"
g, τ, k = M.g, M.τ, M.k
a, ℋ, Φ, Ψ = g.a, g.ℋ, g.Φ, g.Ψ
D = Differential(τ)
@parameters w₀ wₐ cₛ² Ω₀ ρ₀
@variables ρ(τ) P(τ) w(τ) cₐ²(τ) δ(τ, k) θ(τ, k) σ(τ, k)
eqs = [
  w ~ w₀ + wₐ*(1-a)
  ρ₀ ~ 3*Ω₀ / (8*Num(π))
  ρ ~ ρ₀ * a^(-3(1+w₀+wₐ)) * exp(-3wₐ*(1-a))
  P ~ w * ρ
  cₐ² ~ w - 1/(3ℋ) * D(w)/(1+w)
  D(δ) ~ 3ℋ*(w-cₛ²)*δ - (1+w)*((1+9(ℋ/k)^2*(cₛ²-cₐ²))*θ - 3*D(Φ))
  D(θ) ~ (3cₛ²-1)*ℋ*θ + k^2*cₛ²*δ/(1+w) + k^2*Ψ
  σ ~ 0
]
initialization_eqs = [
  δ ~ -3//2 * (1+w) * Ψ
  θ ~ 1//2 * (k^2*τ) * Ψ
]
X = System(eqs, τ; initialization_eqs, name = :X)

# 2) Create extended model and problem
M = ΛCDM(Λ = X, lmax = 16, name = :w₀wₐCDM)
push!(p, X.w₀ => -0.9, X.wₐ => 0.2, X.cₛ² => 1.0)
prob = CosmologyProblem(M, p; jac=true, sparse=true)
```

Symbolic engine does mechanical tasks:

- parameter hooks (e.g. w_0, w_a, c_s^2, Ω_0),
- move (τ)-functions to background,
- move (τ, k)-functions to perturbations,
- expand $\mathrm{D}(w) = \mathrm{d}w/\mathrm{d}\tau$ and $\mathrm{D}(\Phi) = \mathrm{d}\Phi/\mathrm{d}\tau$,
- look up background in perturbations,
- source gravity with energy-momentum,
- spline $\rho(\tau)$ in the perturbations (if integrating $\mathrm{d}\rho/\mathrm{d}\tau$),
- set $\Omega_{X0} = 1 - \sum_{s \neq X} \Omega_{s0}$ (if GR),
- eliminate common subexpressions like $x = 1 + w$ and $y = 1/x$,
- generate ODE code/indices for δ' and θ',
- generate sparse anal. J_{ij} for δ' and θ',
- output any variable, both unknowns (i.e. δ, θ) and observeds (e.g. w, c_a^2).
- lower higher-order derivs. to 1 (e.g. $\frac{\mathrm{d}^2\phi}{\mathrm{d}\tau^2}$),
- everything related to one species in one place,
- compact and readable unicode code.

Other codes are structured by computational stages



CLASS structure

$$\mathbf{J} = [\partial u_i'/\partial u_j] \sim$$

	a	$X_{\rm H}^+$	$X_{\rm He}^+$	T_b	κ	Φ	δ_c	θ_c	δ_b	θ_b	$F_{\gamma 0}$	$F_{\gamma 1}$	$F_{\gamma 2}$	$F_{\gamma 3}$	$F_{\gamma 4}$	$F_{\gamma 5}$	$\Theta_{\gamma l}$
a'	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$X_{\rm H}^{+\prime}$	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$X_{\rm He}^{+\prime}$	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
T_b'	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
κ'	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Φ'	1	0	0	0	0	1	1	0	1	0	1	0	1	0	0	0	0
δ_c'	1	0	0	0	0	1	1	1	1	0	1	0	1	0	0	0	0
θ_c'	1	0	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0
δ_b'	1	1	1	1	0	1	1	0	1	1	1	0	1	0	0	0	0
θ_b'	1	1	1	1	0	1	0	0	1	1	0	1	1	0	0	0	0
$F_{\gamma 0}'$	1	0	0	0	0	1	1	0	1	0	1	1	1	0	0	0	0
$F_{\gamma 1}'$	1	1	1	1	0	1	0	0	0	1	1	1	1	0	0	0	0
$F_{\gamma 2}'$	1	1	1	1	0	0	0	0	0	0	0	1	1	1	0	0	0
$F_{\gamma 3}'$	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0
$F_{\gamma 4}'$	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	0
$F_{\gamma 5}'$	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	0
$\Theta_{\gamma l}'$	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0

Reflects variable dependencies in the full system

- Spline prev. stages
- Scatters one species across many files

SymBoltz is structured by physical components

```
eqs = [
    # metric equations
    z ~ 1/a - 1
    ℋ ~ D(a) / a
    H ~ ℋ / a

    # gravity equations
    D(a) ~ √(8*Num(π)/3 * ρ) * a^2 # 1st Friedmann equation
    D(Φ) ~ -4*Num(π)/3*a^2/ℋ*δρ - k^2/(3*ℋ)*Φ - ℋ*Ψ
    k^2 * (Φ - Ψ) ~ 12*Num(π) * a^2 * Π
    ρ ~ ρc + ρb + ργ + ρν + ρh + ρΛ
    P ~ Pγ + Pν + Ph + PΛ
    δρ ~ δc*ρc + δb*ρb + δγ*ργ + δν*ρν + δh*ρh
    Π ~ (1+wγ)*ργ*σγ + (1+wν)*ρν*σν + (1+wh)*ρh*σh
```

- All (background/perturbations) equations written in one system
- Background/perturbations are automatically split internally
- Everything related to one component located in one place
- Perturbations can freely refer to the background
- **Goal:** work with all equations as if part of one big system

Feature 2: approximation-freeness

Einstein-Boltzmann equations are stiff due to different (inverse) time scales, e.g.:

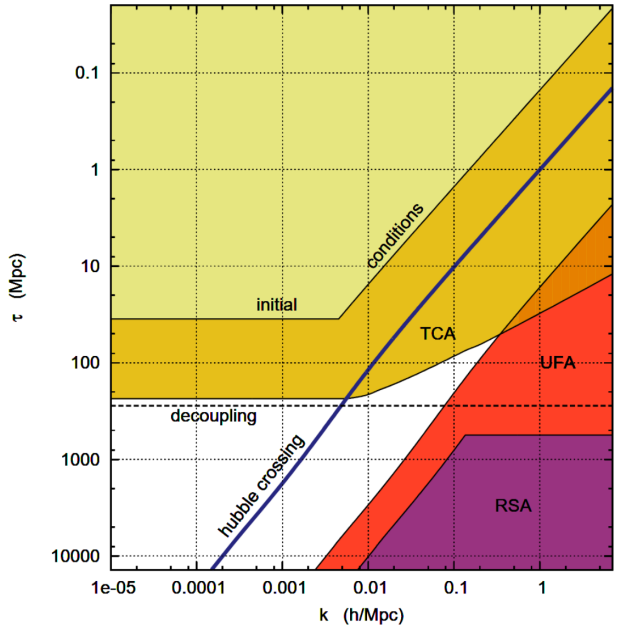
$$\frac{\mathrm{d}\theta_b}{\mathrm{d}\tau} = -\mathcal{H}\theta_b + k^2 c_s^2 \delta_b - \frac{4\rho_\gamma}{3\rho_\beta}\dot{\kappa}(\theta_\gamma - \theta_b)$$

Standard explicit ODE integrators (e.g. RK4) take tiny steps and crash!

Solution 1: remove stiffness with approximations

Remove stiffness (and speed up) with approximation schemes:

- tight coupling (TCA)
- radiation streaming (RSA)
- ultra-relativistic fluid (UFA)
- non-cold dark matter fluid (NCDMFA)
- Saha approximation



CLASS (1104.2933)

- Complicates equations and code (also harder to differentiate)
- Switching criteria must be tuned (repeat for modifications)
- Approximations must be validated (repeat for modifications)
- Extensions reintroduce stiffness and need new approximations
- Biases extensions to less complicated sectors?

Approximations complicate the code

```
9115        /** - ---> photon temperature higher momenta and photon polarization (depend on tight-coupling approximation) */
9116
9117        if (ppw->approx[ppw->index_ap_rsa] == (int)rsa_off) {
9118
9119          /** - ----> if photon tight-coupling is off */
9120          if (ppw->approx[ppw->index_ap_tca] == (int)tca_off) {
9121
9122            /** - -----> define \f$ \Pi = G_{\gamma 0} + G_{\gamma 2} + F_{\gamma 2} \f$ */
9123            P0 = (y[pv->index_pt_pol0_g] + y[pv->index_pt_pol2_g] + 2.*s_l[2]*y[pv->index_pt_shear_g])/8.;
9124
9125            /** - -----> photon temperature velocity */
9126
9127            dy[pv->index_pt_theta_g] =
9128              k2*(delta_g/4.-s2_squared*y[pv->index_pt_shear_g])
9129              + metric_euler
9130              + pvecthermo[pth->index_th_dkappa]*(theta_b-theta_g);
9131
9132            if (pth->has_idm_g == _TRUE_) {
9133              dy[pv->index_pt_theta_g] += dmu_idm_g * (theta_idm - theta_g);
9134            }
9135
9136            /** - -----> photon temperature shear */
9137            dy[pv->index_pt_shear_g] =
9138              0.5*(8./15.*(theta_g+metric_shear)
9139                   -3./5.*k*s_l[3]/s_l[2]*y[pv->index_pt_l3_g]
9140                   -photon_scattering_rate*(2.*y[pv->index_pt_shear_g]-4./5./s_l[2]*P0));
```

- Several versions of equations
- Nested `if-else` branching

Solution 2: integrate full stiff equations with implicit solvers

- Designed to solve stiff systems
- **Challenge:** every step solves $N \sim O(100)$ implicit eqs., e.g.:

$$\text{Explicit Euler method: } \mathbf{u}_{n+1} = \mathbf{u}_n + h\,\mathbf{f}(\mathbf{u}_n, t_n)$$
$$\text{Implicit Euler method: } \mathbf{u}_{n+1} = \mathbf{u}_n + h\,\mathbf{f}(\mathbf{u}_{n+1}, t_n)$$

- Nonlinear solver (e.g. Newton's method) $\rightarrow$ needs ODE $\mathbf{J}$
  - Symbolic equations $\rightarrow$ analytical $\mathbf{J}$
- Dense LU-fact. is $O(N^3)$ $\rightarrow$ want $O(\mathrm{nnz})$ sparse matrices
  - Symbolic equations $\rightarrow$ exact sparsity pattern of $\mathbf{J}$
  - Approximation-free $\rightarrow$ fixed sparsity pattern of $\mathbf{J}$
- Handled by Julia's powerful OrdinaryDiffEq.jl library
- **Reward:** long steps, one simple equation set, no switching

Symbolic codes compute the Jacobian analytically

By definition, the perturbation ODEs $f_i = \sum_j A_{ij} u_j$ are linear.

$$\implies J_{ij} = \partial f_i / \partial u_j = A_{ij}$$
$$\implies f_i = \sum_j J_{ij} u_j$$
$$\implies \text{sparse } \mathbf{J} \text{ requires fewer operations than } \mathbf{f}$$

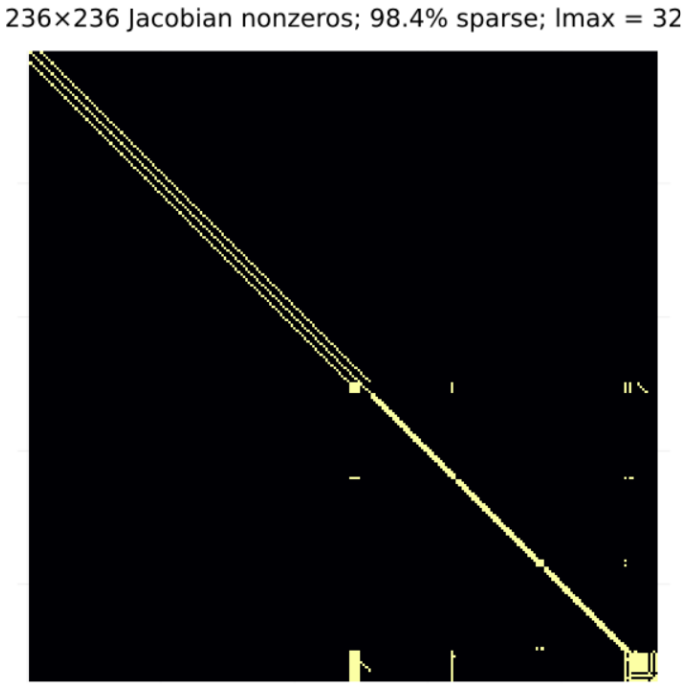
CLASS (😕):

- Knows only $\mathbf{f}$
- Computes approximate $\mathbf{J}$ with $O(N)$ finite-difference evaluations of $\mathbf{f}$.

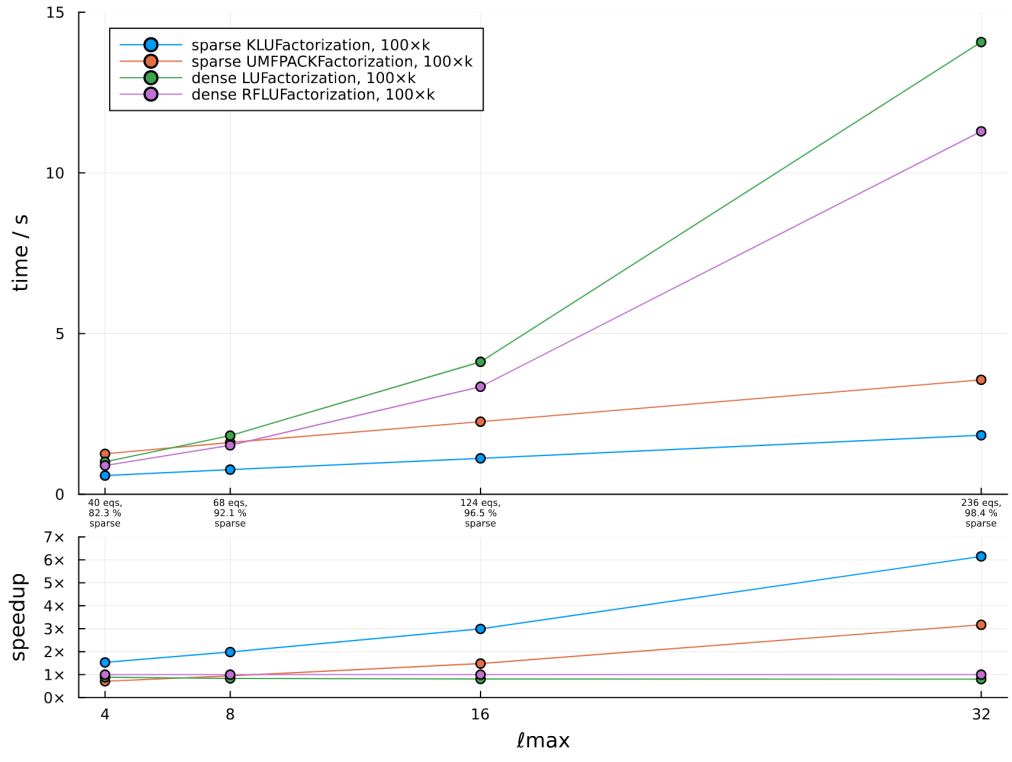
SymBoltz (🙂):

- Knows both $\mathbf{f}$ and $\mathbf{J}$
- Computes exact $\mathbf{J}$ from 1 analytical evaluation with *fewer* operations than $\mathbf{f}$.

Big speedup from sparse matrix methods

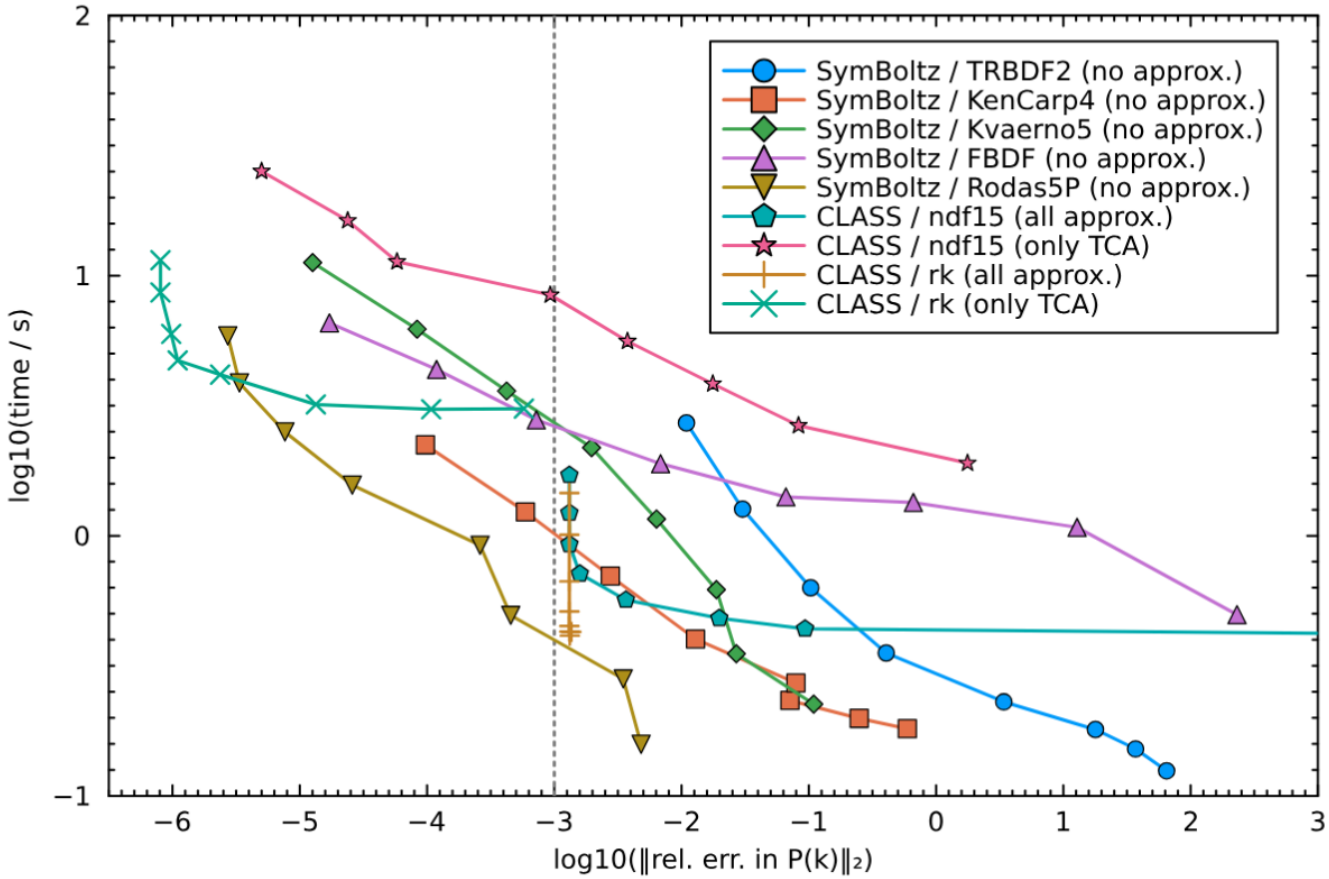


Perturbations are very sparse!



Performance is good: work vs. precision diagram

Choice of implicit solver matters!



L_2 error of $P(k)$. Tolerances 10^{-9}-10^{-2}. Reference tolerance 10^{-10}. Same k-modes. Same l_{max} cutoff. Default massive neutrino q-sampling.

(Line-of-sight integration for C_l not yet as fast as CLASS)

Comparison of approximation/performance characteristics

	CAMB	CLASS	PyCosmo	DISCO-EB	Bolt	SymBoltz
Best implicit solver	-	`ndf15`	`BDF2`	`Kvaerno5`	`KenCarp4`	`Rodas5P`
Approximation-free	No	Almost[1]	Yes	Yes	Yes	Yes
Solver order (stable)	-	1-5 (2)[2]	2 (2)	5 (5)	4 (4)	5 (5)
Newton iterations	-	Yes	No[3]	Yes	Yes	No[4]
Jacobian method	-	$O(N)$ fin.diff.	$O(1)$ anal.	$O(N)$ auto.diff.	$O(N)$ auto.diff.	$O(1)$ anal.
Jacobian sparsity	-	Dynamic[5]	Fixed	Not supported	Not supported	Fixed

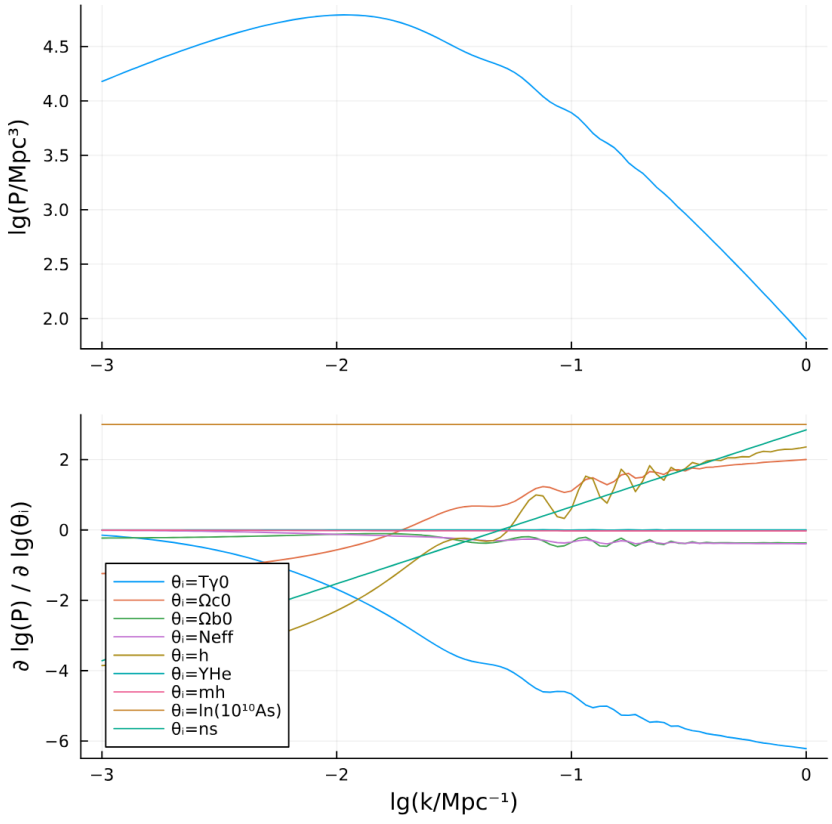
- [1] Tight-coupling approximation is mandatory, but others can be disabled.
- [2] BDF methods are "A-stable" and "L-stable" only to 2nd order (e.g. doi.org/10.1137/S1064827594276424).
- [3] Custom `BDF2` method specializing on linearity $\mathbf{f} = \mathbf{J}\mathbf{u}$ (see 1708.05177 eq. (21)).
- [4] Rosenbrock methods linearize $\mathbf{f} = \mathbf{J}\mathbf{u}$ (e.g. doi.org/10.1007/s10543-023-00967-x).
- [5] Sparsity found numerically and changes with approximations.

Feature 3: differentiability

Derivatives are important in e.g.:

- Gradient-baed MCMCs: HMC, NUTS
- Training machine learning emulators (minimize loss)
- Fisher forecasting
- Sensitivity analysis: $\partial(\text{output})/\partial(\text{input})$

Autodiff propagates exact gradients through the code:



Derivatives of $P(k)$ wrt. parameters

What is automatic differentiation?

Any computer program is one (big) composite function

$$\mathbf{f} = \mathbf{f}_N \circ \mathbf{f}_{N-1} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1 = \mathbf{f}_N(\mathbf{f}_{N-1}(\cdots \mathbf{f}_2(\mathbf{f}_1)))$$

Finite differences: approximate $J_{ij} \approx \frac{\mathbf{f}(\mathbf{x}+\mathbf{e}_j\frac{\epsilon}{2})-\mathbf{f}(\mathbf{x}-\mathbf{e}_j\frac{\epsilon}{2})}{\epsilon} \cdot \mathbf{e}_i$.

Automatic differentiation: any method computing chain rule

$$\mathbf{J} = \mathbf{J}_N \cdot \mathbf{J}_{N-1} \cdots \mathbf{J}_2 \cdot \mathbf{J}_1 = \frac{\partial \mathbf{f}_N}{\partial \mathbf{f}_{N-1}} \cdot \frac{\partial \mathbf{f}_{N-1}}{\partial \mathbf{f}_{N-2}} \cdots \frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2} \cdot \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}.$$

- Source code transformation ($\mathbf{f} \to$ compiler $\to \mathbf{J}$)
- Dual numbers (number type with $(\mathbf{f}, \mathbf{f}')$)

Unlike $\mathbf{f}$, the product for $\mathbf{J}$ can be evaluated in any order:

- Forward-mode seeds $\mathbf{J}_1 = \mathbf{1}$; fastest when $\mathbf{f}$ has more outputs
- Reverse-mode seeds $\mathbf{J}_N = \mathbf{1}$; fastest when $\mathbf{f}$ has more inputs

Example: finite differences vs. automatic differentiation

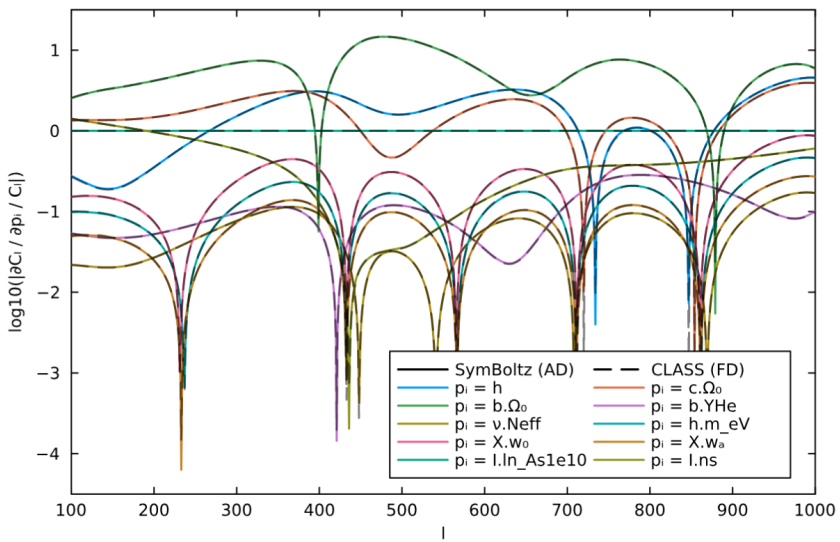
```julia
function f(x)
    println("typeof(x) = ", typeof(x))
    return x^2
end

f(3.0)
# typeof(x) = Float64
# 9.0

# 1) Finite differences
ϵ = 1e-10
(f(3.0+ϵ/2) - f(3.0-ϵ/2)) / ϵ
# typeof(x) = Float64
# typeof(x) = Float64
# 6.000000496442226

# 2) Automatic differentiation (forward-mode dual numbers)
using ForwardDiff
ForwardDiff.derivative(f, 3.0)
# typeof(x) = ForwardDiff.Dual{ForwardDiff.Tag{typeof(f), Float64}, Float64, 1}
# 6.0
```

Forward-mode AD works in SymBoltz

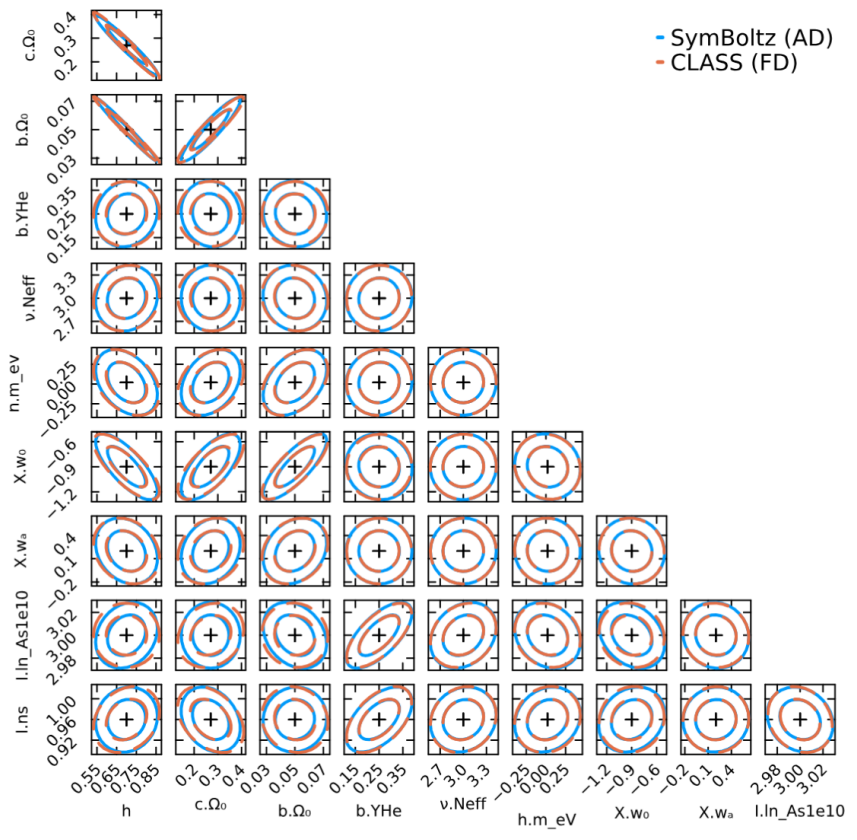
Can do differentiable Fisher forecasting, e.g. for cosmic variance-dominated CMB survey:

$$\log L(\mathbf{p}) = -\frac{1}{2}\sum_l \left(\frac{C_l(\mathbf{p}) - \bar{C}_l}{\sigma_l^2}\right)^2, \quad \sigma_l = \sqrt{\frac{2}{2l+1}}\bar{C}_l,$$

$$F_{ij} = -\frac{1}{2}\frac{\partial^2 \log L}{\partial p_i \partial p_j} = \sum_l \frac{\partial C_l}{\partial p_i}\frac{1}{\sigma_l^2}\frac{\partial C_l}{\partial p_j}$$

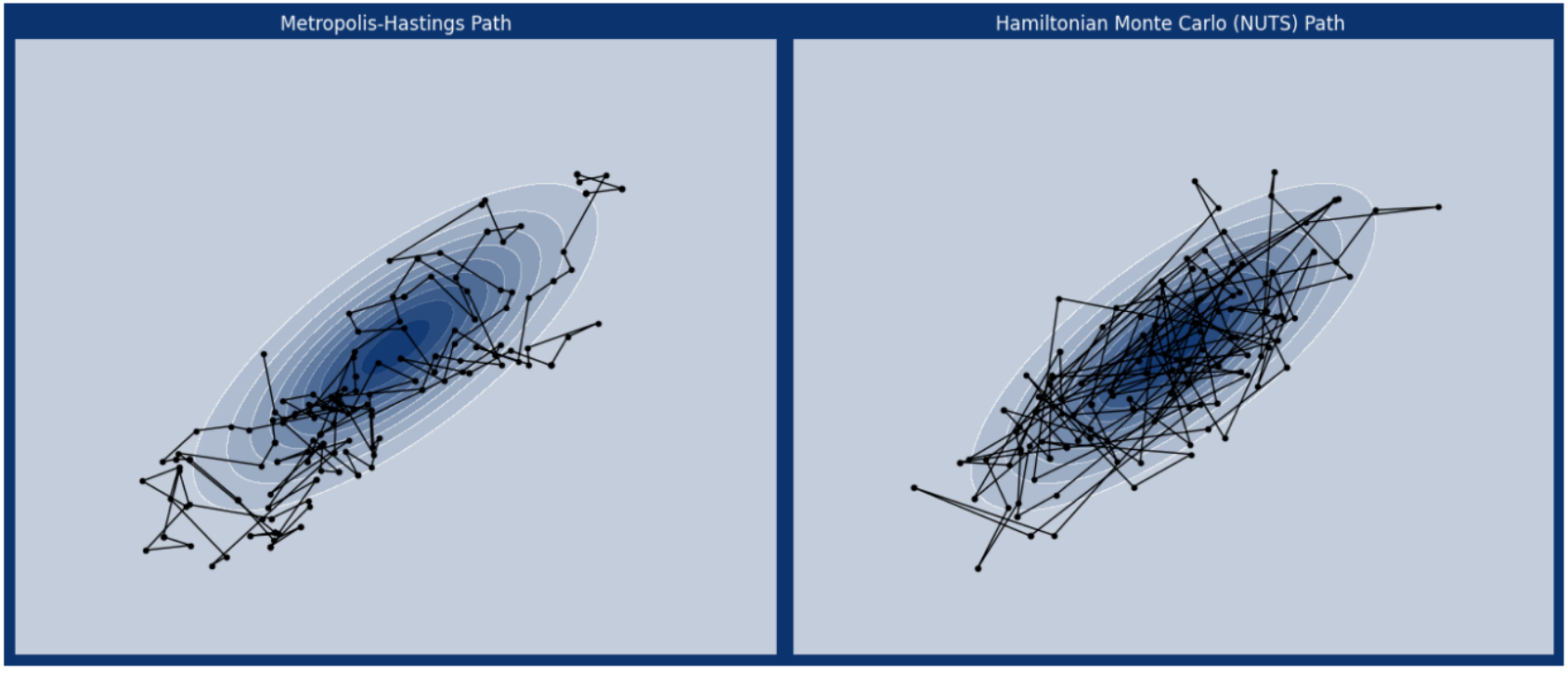


- CLASS: needs step size tuning
- SymBoltz: no step size tuning
- SymBoltz currently slower for differentiable runs



Reverse-mode AD does not work yet in SymBoltz

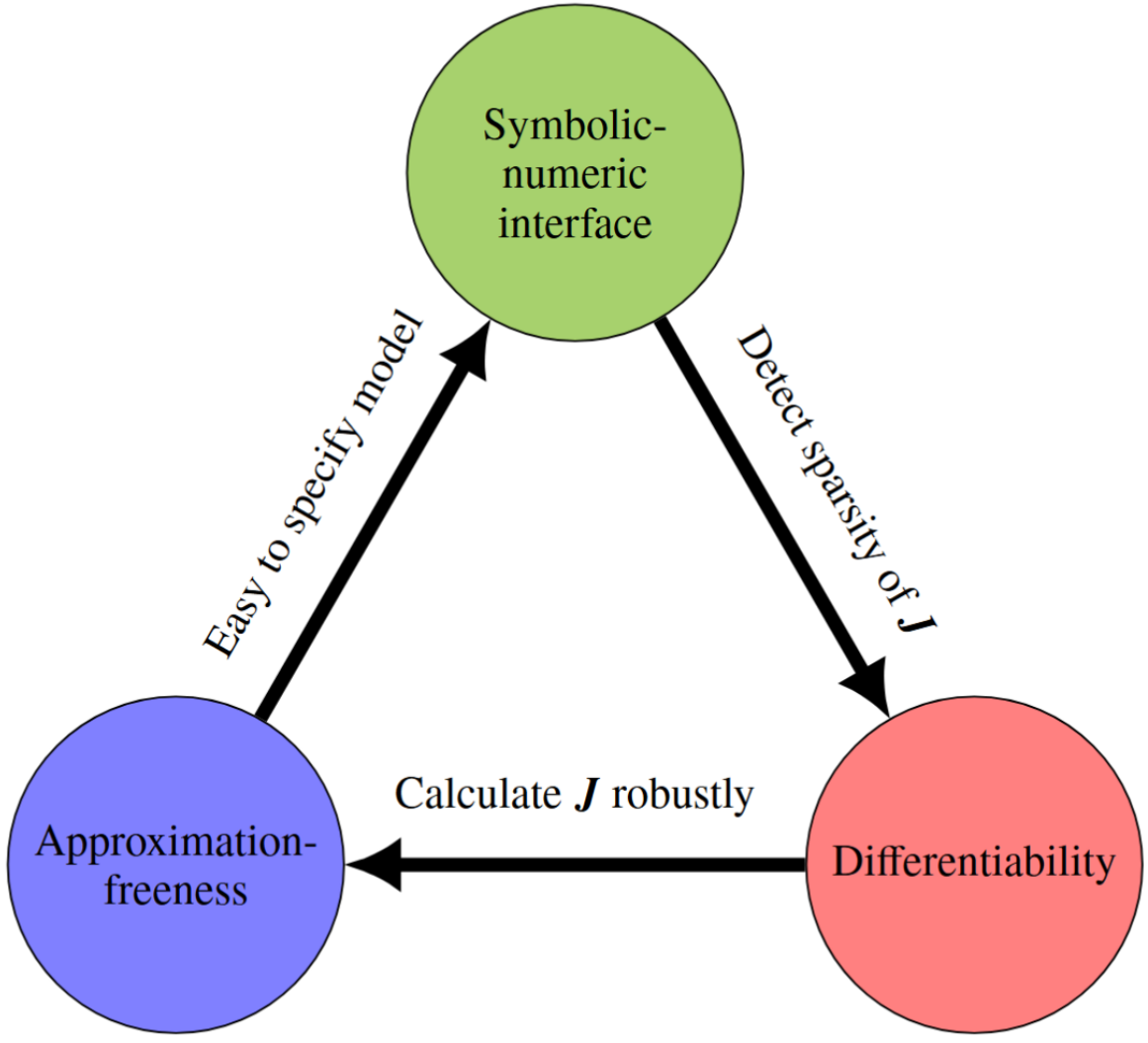
- Most attractive for MCMCs, since $L(\mathbf{p})$ is vector-to-scalar
- Want to enable use of gradient-based MCMC methods:



HMC explores likelihood more efficiently (2512.09724)

- **Goal:** $T(L, \nabla L) \cdot N(\text{NUTS}) < T(L) \cdot N(\text{MH})$ (2406.04725)

Main features reinforce each other



"Stronger together" 💪

Current feature status and future work

- 🟩 Baryons (RECFAST)
- 🟩 Photons
- 🟩 Cold dark matter
- 🟩 Massless neutrinos
- 🟩 Massive neutrinos
- 🟩 Cosmological constant
- 🟩 $w_0 w_a$ dark energy
- 🟩 General Relativity
- 🟨 Brans-Dicke gravity
- 🟨 Curved geometry
- 🟩 Scalar perturbations
- 🟥 Vector perturbations
- 🟥 Tensor perturbations

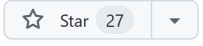
- 🟩 Matter power spectrum
- 🟩 CMB power spectrum
- 🟨 CMB lensing
- 🟩 Symbolic interface
- 🟩 Auto stage separation
- 🟩 Auto spline ODE vars
- 🟩 Auto compute alg vars
- 🟩 Auto (τ, k)-interpolate
- 🟩 Auto parameter hooks
- 🟨 Auto gauge transf
- 🟨 Auto initial conditions
- 🟥 Auto unit conversion
- 🟩 Auto plot recipes

- 🟩 Shooting method
- 🟩 Approximation-free
- 🟩 Performance
- 🟩 Forward-mode autodiff
- 🟥 Reverse-mode autodiff
- 🟨 Performance (autodiff)
- 🟩 Documentation
- 🟩 Testing
- 🟩 CLASS comparison
- 🟨 Non-linear boosting
- 🟥 GPU support
- 🟩 Paper (arXiv)
- ⬛ **Ideas welcome!**

The end

Available github.com/hersle/SymBoltz.jl

- Install with `using Pkg; Pkg.add("SymBoltz")` in Julia
- Link to documentation and paper 

- Star Github if you find this interesting :) 

- Looking for people to try it and give feedback!
- Feel free to open issues on Github!

Thank you!