IllinoisGRMHD Progress Update

Piecewise Polytropic Equation of State Support

Leonardo (Leo) Werneck

In collaboration with Zach Etienne

Funding acknowledgement

CAPES

TCAN-80NSSC18K1488

TCAN on Binary Neutron Stars Workshop, July 7, 2020

Overview: IllinoisGRMHD

- A 2015 rewrite of the original GRMHD code of the Illinois Numerical Relativity Group (https://arxiv.org/abs/1501.07276)
- Roundoff agreement with the original code, while being ~2x faster and containing ~23x less lines of code (from ~70k to <3k)
- Open-sourced and available as part of the Einstein Toolkit (https://www.einsteintoolkit.org/)
- GRMHD for dynamical spacetimes, including single neutron stars; binary neutron stars with and without magnetic fields; black hole accretion disks; and many more!
- Room for improvement

this talk

  - Good, but not great, documentation
  - Hybrid EOS support, but only some functions support piecewise polytrope-based
  - No neutrino physics or electron fraction

Overview: IllinoisGRMHD

- Main goal: extend equation of state support of **IllinoisGRMHD** and add neutrino physics
- But first: needed to learn how to navigate the code properly
- The code has been designed to be user-friendly, but was not a state-of-the-art example of “literate programming” (ala Knuth)
- Learned my way around the code by documenting it in pedagogical Jupyter notebooks, alongside NRPy+

IllinoisGRMHD: new documentation

Step 2.c: Computing U_r and U_l [Back to top]

We now compute U_r and U_l. Keep in mind that $U_{r,i} = U_{i+1/2}$, while $U_{l,i} = U_{i-1/2}$. The implemented equation follows eq. A1 in Duez *et al.* (2005), but with the standard PPM coefficient of $\frac{1}{6}$ (i.e. eq. A1 with $\frac{1}{8} \rightarrow \frac{1}{6}$). Keep in mind that we simplify the equation slightly before implementing it:

$$U_{r,i+0} = U_{i+0} + \frac{1}{2}(U_{i+1} - U_{i+0}) + \frac{1}{6}\left(\delta U_{i+0}^{\text{slope-lim}} - \delta U_{i+1}^{\text{slope-lim}}\right) \Rightarrow \boxed{U_{r,i+0} = \frac{1}{2}(U_{i+1} + U_{i+0}) + \frac{1}{6}\left(\delta U_{i+0}^{\text{slope-lim}} - \delta U_{i+1}^{\text{slope-lim}}\right)},$$

$$U_{l,i+0} = U_{i-1} + \frac{1}{2}(U_{i+0} - U_{i-1}) + \frac{1}{6}\left(\delta U_{i-1}^{\text{slope-lim}} - \delta U_{i+0}^{\text{slope-lim}}\right) \Rightarrow \boxed{U_{l,i+0} = \frac{1}{2}(U_{i+0} + U_{i-1}) + \frac{1}{6}\left(\delta U_{i-1}^{\text{slope-lim}} - \delta U_{i+0}^{\text{slope-lim}}\right)}.$$

After this step, the values of $U_{r,l,i+0}$ are stored as outputs.

[7]:
```
%%writefile -a $outdir/reconstruct_set_of_prims_PPM.C


      // Finally, compute face values Ur and Ul based on the PPM prescription
      //   (Eq. A1 in http://arxiv.org/pdf/astro-ph/0503420.pdf, but using standard 1/6=(1.0/6.0) coefficient)
      // Ur[PLUS0] represents U(i+1/2)
      // We applied a simplification to the following line: Ur=U+0.5*(U(i+1)-U) + ....= 0.5*(U(i+1)+U) + ....
      Ur[whichvar][PLUS0] = 0.5*(U[whichvar][PLUS1] + U[whichvar][PLUS0] ) + (1.0/6.0)*(slope_lim_dU[whichvar][PLUS0]  - slope_lim_dU[whichvar][PLUS1]);

      // Ul[PLUS0] represents U(i-1/2)
      // We applied a simplification to the following line: Ul=U(i-1)+0.5*(U-U(i-1)) + ....= 0.5*(U+U(i-1)) + ....
      Ul[whichvar][PLUS0] = 0.5*(U[whichvar][PLUS0] + U[whichvar][MINUS1]) + (1.0/6.0)*(slope_lim_dU[whichvar][MINUS1] - slope_lim_dU[whichvar][PLUS0]);

      /* *** LOOP 1c: WRITE OUTPUT *** */
      // Store right face values to {rho_br,Pr,vxr,vyr,vzr,Bxr,Byr,Bzr},
      //    and left face values to {rho_bl,Pl,vxl,vyl,vzl,Bxl,Byl,Bzl}
      OUT_PRIMS_R[whichvar].gf[index_arr[flux_dirn][PLUS0]] = Ur[whichvar][PLUS0];
      OUT_PRIMS_L[whichvar].gf[index_arr[flux_dirn][PLUS0]] = Ul[whichvar][PLUS0];
    }
```

```
Appending to ../src/reconstruct_set_of_prims_PPM.C
```

Available at: https://github.com/zachetienne/nrpytutorial/tree/master/IllinoisGRMHD

Hybrid equation of state

- IllinoisGRMHD currently supports hybrid equations of state of the form

$$P(\rho) = P_{\mathrm{cold}}(\rho) + P_{\mathrm{th}}(\rho)$$

where

$$P_{\mathrm{th}}(\rho) = \left(\Gamma_{\mathrm{th}} - 1\right) \rho \left(\epsilon - \epsilon_{\mathrm{cold}}\right)$$

and P_{cold} is described by using either:

✓ A simple polytropic equation of state

✓ A piecewise polytropic equation of state

Piecewise polytropic equation of state support

- Simple polytrope

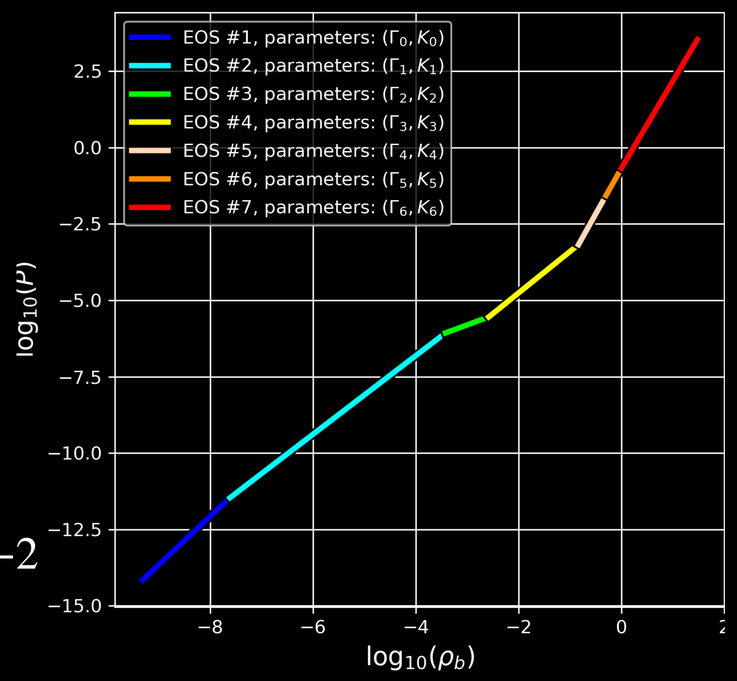
$$P_{\rm cold}(\rho) = K_{\rm cold}\rho^{\Gamma_{\rm cold}}$$

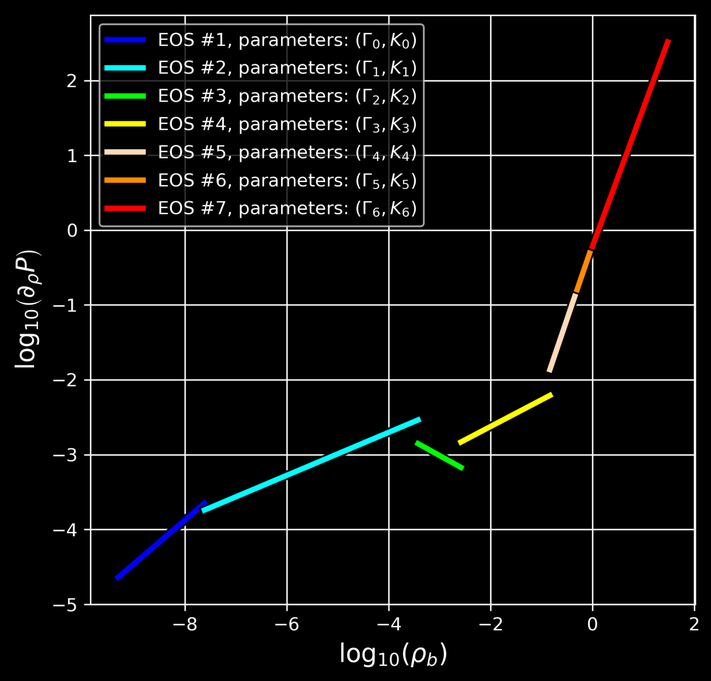
- Piecewise polytrope
(better approx. to realistic EOSs)

$$P_{\rm cold} = \begin{cases} K_0\rho_b^{\Gamma_0} & , \quad \rho_b \leq \rho_0 \\ K_1\rho_b^{\Gamma_1} & , \quad \rho_0 \leq \rho_b \leq \rho_1 \\ \vdots & \quad \vdots \\ K_j\rho_b^{\Gamma_j} & , \quad \rho_{j-1} \leq \rho_b \leq \rho_j \\ \vdots & \quad \vdots \\ K_{N-2}\rho_b^{\Gamma_{N-2}} & , \quad \rho_{N-3} \leq \rho_b \leq \rho_{N-2} \\ K_{N-1}\rho_b^{\Gamma_{N-1}} & , \quad \rho_b \geq \rho_{N-2} \end{cases}$$

P is everywhere continuous, by construction

However, its first derivative is *not*





SLy EOS from Read *et al.*, PRD 79:124032,2009, arXiv:0812.2163

Piecewise polytropic equation of state support

- Thoroughly documented using Jupyter notebooks

Step 2: Continuity of $P_{\rm cold}$ [Back to top]

Consider a piecewise polytrope EOS of the form

$$P_{\rm cold} = \begin{cases} K_0 \rho_b^{\Gamma_0} & , \quad \rho_b \leq \rho_0 \\ K_1 \rho_b^{\Gamma_1} & , \quad \rho_0 \leq \rho_b \leq \rho_1 \\ \vdots & \quad \vdots \\ K_j \rho_b^{\Gamma_j} & , \quad \rho_{j-1} \leq \rho_b \leq \rho_j \\ \vdots & \quad \vdots \\ K_{N-2} \rho_b^{\Gamma_{N-2}} & , \quad \rho_{N-3} \leq \rho_b \leq \rho_{N-2} \\ K_{N-1} \rho_b^{\Gamma_{N-1}} & , \quad \rho_b \geq \rho_{N-2} \end{cases} .$$

The case of a single polytrope is given by the first EOS above, with no condition imposed on the value of ρ, i.e.

$$\boxed{P_{\rm cold} = K_0 \rho_b^{\Gamma_0} = K \rho_b^{\Gamma}} .$$

Notice that we have the following sets of variables:

$$\left\{ \underbrace{\rho_0, \rho_1, \ldots, \rho_{N-2}}_{N-1 \text{ values}} \right\} ; \left\{ \underbrace{K_0, K_1, \ldots, K_{N-1}}_{N \text{ values}} \right\} ; \left\{ \underbrace{\Gamma_0, \Gamma_1, \ldots, \Gamma_{N-1}}_{N \text{ values}} \right\} .$$

Also, notice that K_0 and the entire sets $\{\rho_0, \rho_1, \ldots, \rho_{N-1}\}$ and $\{\Gamma_0, \Gamma_1, \ldots, \Gamma_N\}$ must be specified by the user. The values of $\{K_1, \ldots, K_N\}$, on the other hand, are determined by imposing that $P_{\rm cold}$ be continuous, i.e.

$$P_{\rm cold}(\rho_0) = K_0 \rho_0^{\Gamma_0} = K_1 \rho_0^{\Gamma_1} \implies \boxed{K_1 = K_0 \rho_0^{\Gamma_0 - \Gamma_1}} .$$

- Stringent validation tests

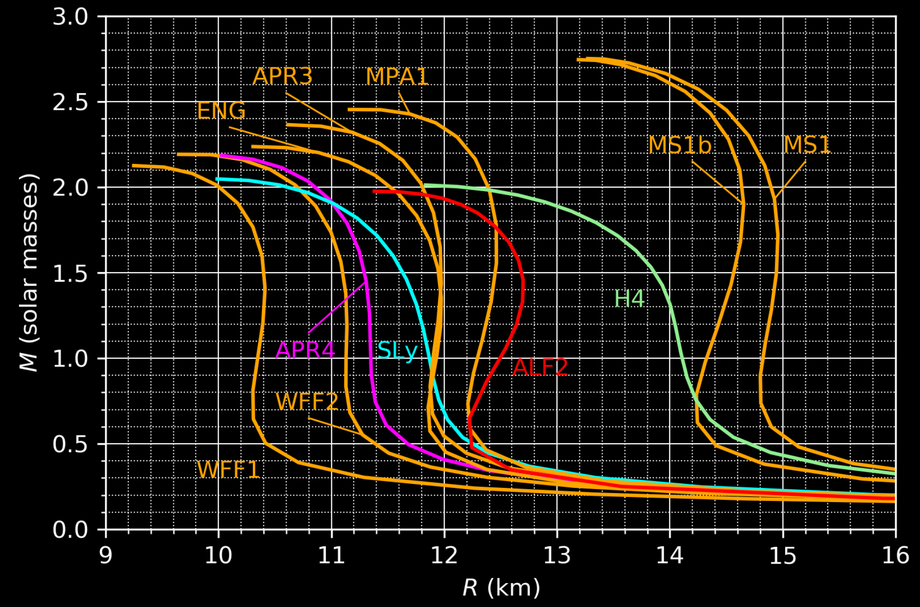
Step 11.d: Mass vs. Radius relations [Back to top]

We now subject our functions to a very stringent test, where we aim to determine the relationship between the mass and radius of Neutron stars subject to different piecewise polytropic EOS parameters.

A similar, yet more complete, plot to the one we are generating is Figure 3 in Demorest *et al.* (2010) (notice that the figure can be accessed by going into the "Figure" tab on the right).

Another plot which is identical to ours can be found by looking at the results of Joonas Nättilä's TOV solver.

Let us start by defining a simple loop that computes a set of pairs $\{M, R\}$ for different values of $\rho_{\rm central}$.

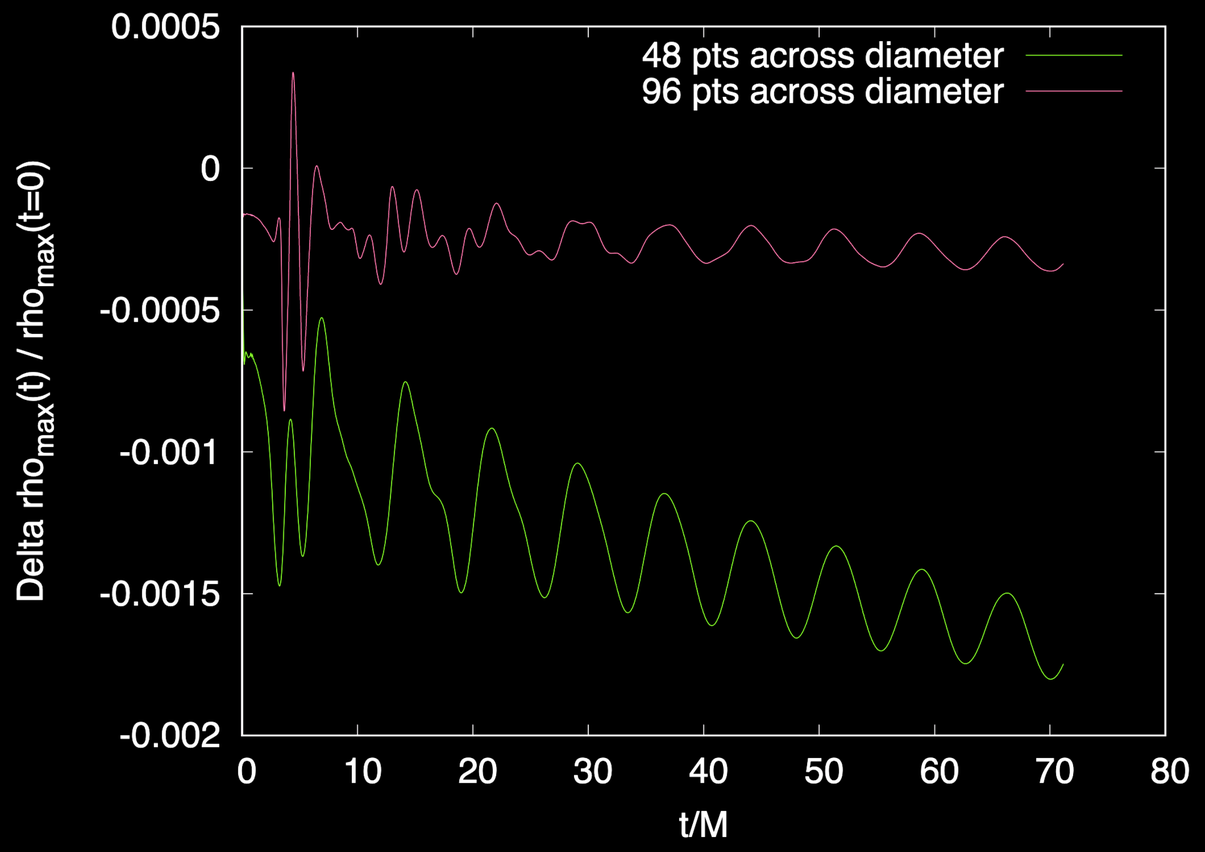


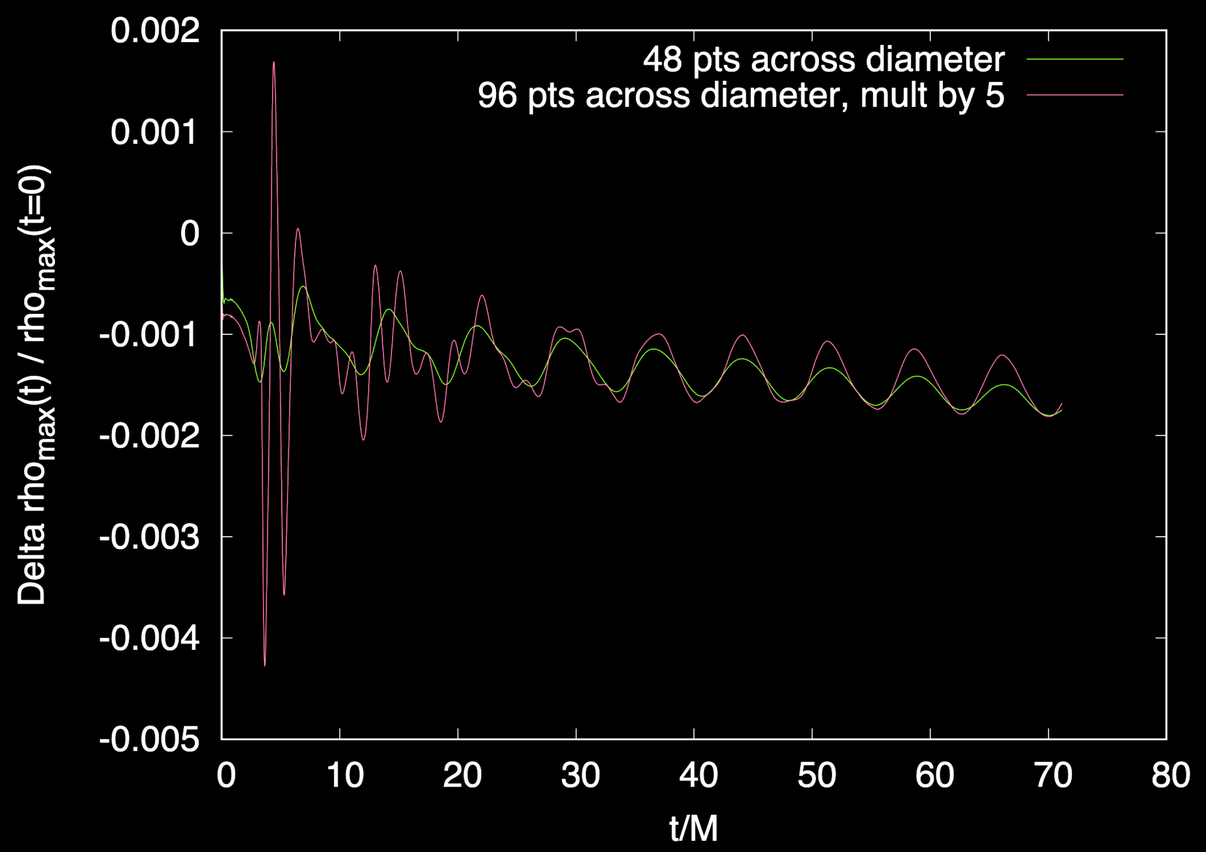
EOS parameters from Read *et al.*, PRD 79:124032,2009, arXiv:0812.2163

Available at http://nrpyplus.net: Tutorial-TOV-Piecewise_Polytrope_EOSs.ipynb

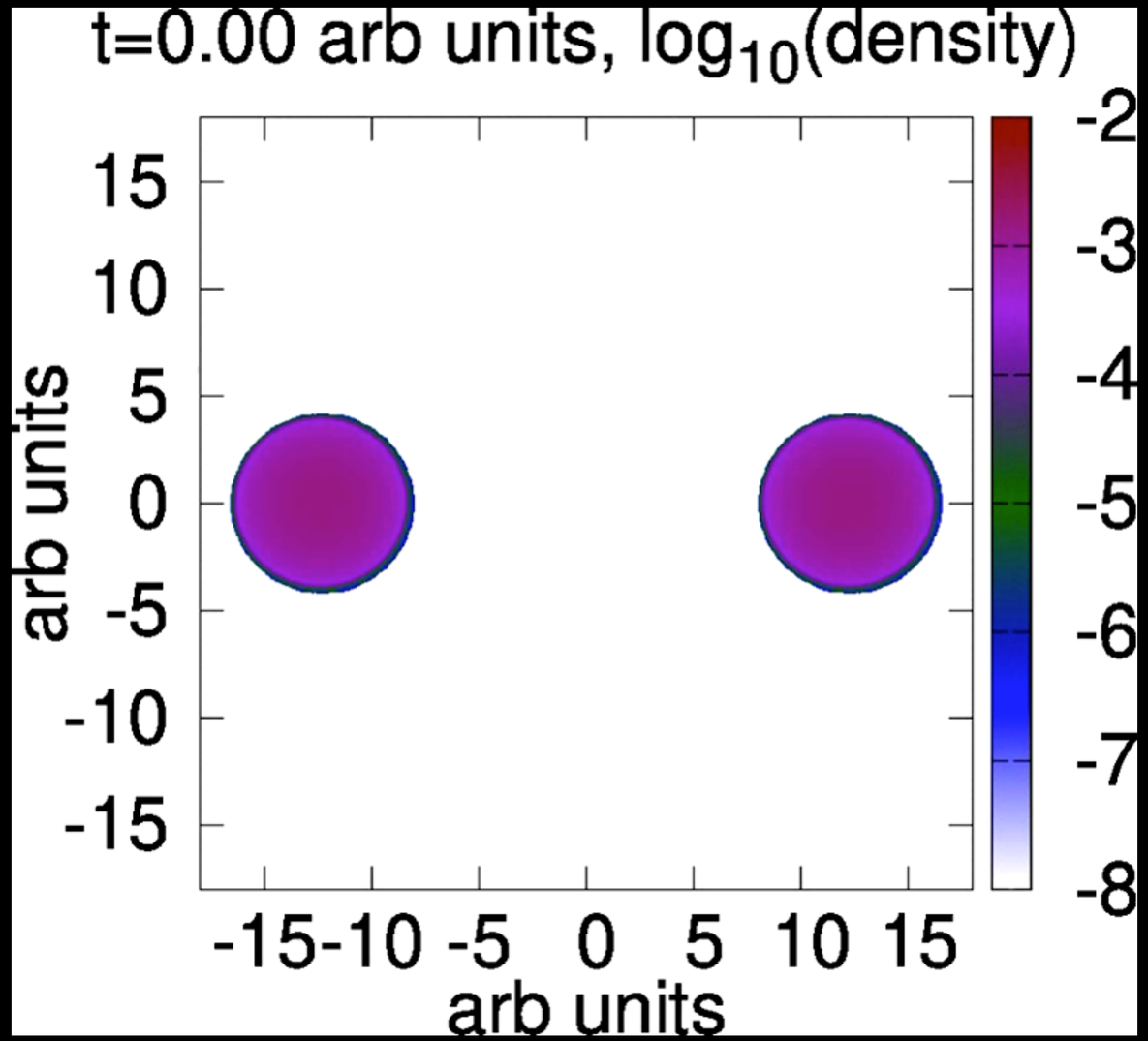
Piecewise polytropic equation of state support

- Single TOV star, hybrid EOS, with APR4 cold PPEOS convergence test





Piecewise polytropic equation of state support



SLy hybrid EOS from Read *et al.,* PRD 79:124032,2009, arXiv:0812.2163

Initial data for two equal mass NSs (1.5 solar masses), obtained using LORENE

Movie by Zach Etienne

IllinoisGRMHD + HARM3D: the hand-off

- The TCAN collaboration is currently working on a hand-off approach
- BNS inspiral and merger → **IllinoisGRMHD**
- Post merger/BH accretion disk → **HARM3D**
- **HARM3D** has a head start with tabulated EOS support & neutrino leakage
- Goal: implement the same tabulated EOS & neutrino leakage infrastructure used by **HARM3D** in **IllinoisGRMHD**, ensuring that the data is consistently handled by both codes

Tabulated equation of state support

- Most realistic neutron star EOSs available
  - ★ Particularly interested in the EOSs available at https://stellarcollapse.org/
  - ★ Tune in for Luke Robert's talk tomorrow!
- Introduces new primitive variables: the temperature and the electron fraction
  - ★ Requires updating of the conservatives-to-primitives routines (based on D. Siegel et al. algorithms)
  - ★ Currently working with Scott and Ari (fellow TCANers!) to leverage their recent work in implementing tabulated EOS and neutrino leakage in **HARM3D**
  - ★ Tune in for Scott and Ari's talks tomorrow!
- Current status in IllinoisGRMHD:
  - ★ TOV solver already fully supports tabulated EOSs
  - ★ IGM functions have been adapted to support tabulated EOSs and code compiles
  - ★ Work is ongoing!!

Neutrino physics and leakage scheme

$$
\begin{aligned}
\partial_t \rho_* &+ \partial_j \left(\rho_* v^j\right) &&= 0 \\
\partial_t \tilde{\tau} &+ \partial_j \left(\alpha^2 \sqrt{\gamma} T^{0j} - \rho_* v^j\right) &&= s + \alpha\sqrt{\gamma} Q u^0 \\
\partial_t \tilde{S}_i &+ \partial_j \left(\alpha \sqrt{\gamma} T^j{}_i\right) &&= \tfrac{1}{2}\alpha\sqrt{\gamma} T^{\mu\nu} \partial_i g_{\mu\nu} + \alpha\sqrt{\gamma} Q u^i \\
\partial_t \tilde{B}^i &+ \partial_j \left(v^j \tilde{B}^i - v^i \tilde{B}^j\right) &&= 0 \\
\partial_t Y_e^* &+ \partial_i \left(Y_e^* v^i\right) &&= + \alpha\sqrt{\gamma} R
\end{aligned}
$$

$$
s = \alpha\sqrt{\gamma}\left[\left(T^{00}\beta^i\beta^j + 2T^{0i}\beta^j + T^{ij}\right) K_{ij} - \left(T^{00}\beta^i + T^{0i}\right)\partial_i \alpha\right]
$$

$$
T^{\mu\nu} = T^{\mu\nu}_{\mathrm{GRHD}} + T^{\mu\nu}_{\mathrm{EM}} = \left(\rho_0 h + b^2\right) u^\mu u^\nu + \left(P + \frac{b^2}{2}\right) g^{\mu\nu} - b^\mu b^\nu
$$

Computed using routines from **HARM3D**

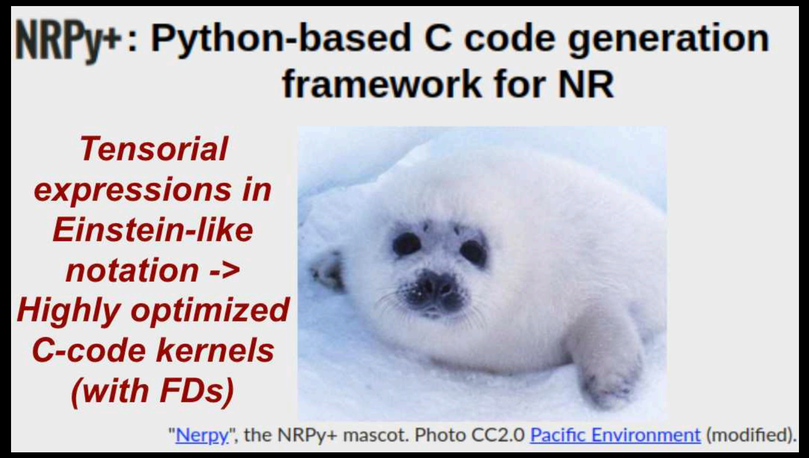
NRPy+ helps us extend these equations while minimizing human error!!!

See e.g. Duez *et al.*, Phys.Rev. D72 (2005) 024028, arXiv:astro-ph/0503420

See Ruffert *et al.*, Astron.Astrophys.311:532-566,1996, arXiv:astro-ph/9509006

NRPy+

- NRPy+: "Python-based code generation for numerical relativity... and beyond!”



Available at: http://nrpyplus.net/

- Similar to Kranc, but no Mathematica/Maple license required

- Completely open-sourced, permissively licensed

Overview: NRPy+

- Input: Einstein notation + simple Python data structures (lists)
- Output: Highly efficient C code with CSE, SIMD, and finite differences

Single-Instruction, Multiple Data (SIMD) compiler intrinsics

Common Subexpression Elimination (CSE)

```
[1]: from outputC import outputC
     import sympy as sp

     # Declare some variables, using SymPy's symbols() function
     a,b,c = sp.symbols("a b c")

     # Set x = b*sin(2*a) + c/sin(2*a)
     x = b*sp.sin(2*a) + c/( sp.sin(2*a) )

     outputC(x,"x")
```

```
/*
 *  Original SymPy expression:
 *  "x = b*sin(2*a) + c/sin(2*a)"
 */
{
   const double tmp_0 = sin(2*a);
   x = b*tmp_0 + c/tmp_0;
}
```

```
[2]: # Initialize code Python/NRPy+ modules
     import sympy as sp
     import NRPy_param_funcs as par
     from outputC import outputC

     # Declare some variables, using SymPy's symbols function
     a,b,c = sp.symbols("a b c")

     # Set x = b*sin(2*a) + c/sin(2*a)
     x = b*sp.sin(2*a) + c/( sp.sin(2*a) )

     outputC(x,"x",params="SIMD_enable=True")
```

```
/*
 *  Original SymPy expression:
 *  "x = b*sin(2*a) + c/sin(2*a)"
 */
{
   const double tmp_Integer_1 = 1.0;
   const REAL_SIMD_ARRAY _Integer_1 = ConstSIMD(tmp_Integer_1);

   const double tmp_Integer_2 = 2.0;
   const REAL_SIMD_ARRAY _Integer_2 = ConstSIMD(tmp_Integer_2);

   const double tmp_NegativeOne_ = -1.0;
   const REAL_SIMD_ARRAY _NegativeOne_ = ConstSIMD(tmp_NegativeOne_);

   const REAL_SIMD_ARRAY tmp_0 = SinSIMD(MulSIMD(_Integer_2, a));
   x = FusedMulAddSIMD(b, tmp_0, DivSIMD(c, tmp_0));
}
```

Overview: NRPy+

```
[3]: import sympy as sp
     from outputC import lhrh
     import grid as gri
     import indexedexp as ixp
     import NRPy_param_funcs as par
     import finite_difference as fin

     # Set the spatial dimension to 1
     par.set_parval_from_str("grid::DIM",1)

     # Set the finite difference accuracy to second order
     par.set_parval_from_str("finite_difference::FD_CENTDERIVS_ORDER",2)

     # Register the input gridfunction "phi" and the gridfunction to which date is output, "output"
     phi, output = gri.register_gridfunctions("AUX",["phi","output"])

     # Declare phi_dDD: a rank-2 indexed expression: phi_dDD[i][j] := \partial_{i}\partial_{j}\phi
     phi_dDD = ixp.declarerank2("phi_dDD","sym01")

     # Set output to \partial_{0}^{2}\phi
     output = phi_dDD[0][0]

     # Output to the screen the core C code for evaluating the finite difference derivative
     fin.FD_outputC("stdout",lhrh(lhs=gri.gfaccess("out_gf","output"),rhs=output))
```

Computing $\partial_x^2\phi$ using NRPy+

```
{
   /*
    * NRPy+ Finite Difference Code Generation, Step 1 of 2: Read from main memory and compute finite difference stencils:
    */
   /*
    *  Original SymPy expression:
    *  "const double phi_dDD00 = invdx0**2*(-2*phi + phi_i0m1 + phi_i0p1)"
    */
   const double phi_i0m1 = aux_gfs[IDX2(PHIGF, i0-1)];
   const double phi = aux_gfs[IDX2(PHIGF, i0)];
   const double phi_i0p1 = aux_gfs[IDX2(PHIGF, i0+1)];
   const double FDPart1_Integer_2 = 2.0;
   const double phi_dDD00 = ((invdx0)*(invdx0))*(-FDPart1_Integer_2*phi + phi_i0m1 + phi_i0p1);
   /*
    * NRPy+ Finite Difference Code Generation, Step 2 of 2: Evaluate SymPy expressions and write to main memory:
    */
   /*
    *  Original SymPy expression:
    *  "aux_gfs[IDX2(OUTPUTGF, i0)] = phi_dDD00"
    */
   aux_gfs[IDX2(OUTPUTGF, i0)] = phi_dDD00;
}
```

“NRPy-fication” of GRMHD flux and source terms

Recall from above that

$$T^{\mu\nu}_{\rm EM} = b^2 u^\mu u^\nu + \frac{1}{2} b^2 g^{\mu\nu} - b^\mu b^\nu.$$

Also

$$T^{\mu}_{{\rm EM}\nu} = T^{\mu\delta}_{\rm EM} g_{\delta\nu}$$

[6]:
```
# Step 3.a: Define T_{EM}^{mu nu} (a 4-dimensional tensor)
def compute_TEM4UU(gammaDD,betaU,alpha, smallb4U, smallbsquared,u4U):
    global TEM4UU

    # Then define g^{mu nu} in terms of the ADM quantities:
    import BSSN.ADMBSSN_tofrom_4metric as AB4m
    AB4m.g4UU_ito_BSSN_or_ADM("ADM",gammaDD,betaU,alpha)

    # Finally compute T^{mu nu}
    TEM4UU = ixp.zerorank2(DIM=4)
    for mu in range(4):
        for nu in range(4):
            TEM4UU[mu][nu] = smallbsquared*u4U[mu]*u4U[nu] \
                             + sp.Rational(1,2)*smallbsquared*AB4m.g4UU[mu][nu] \
                             - smallb4U[mu]*smallb4U[nu]

# Step 3.b: Define T^{mu}_{nu} (a 4-dimensional tensor)
def compute_TEM4UD(gammaDD,betaU,alpha, TEM4UU):
    global TEM4UD
    # Next compute T^mu_nu = T^{mu delta} g_{delta nu}, needed for S_tilde flux.
    # First we'll need g_{alpha nu} in terms of ADM quantities:
    import BSSN.ADMBSSN_tofrom_4metric as AB4m
    AB4m.g4DD_ito_BSSN_or_ADM("ADM",gammaDD,betaU,alpha)
    TEM4UD = ixp.zerorank2(DIM=4)
    for mu in range(4):
        for nu in range(4):
            for delta in range(4):
                TEM4UD[mu][nu] += TEM4UU[mu][delta]*AB4m.g4DD[delta][nu]
```

Greatly expedites implementation of complicated expressions, while generating highly optimized C code

Human friendly coding!

Fantastic extensibility!

Recall from above that

$$T^{\mu\nu} = \rho_0 h u^\mu u^\nu + P g^{\mu\nu},$$

where $h = 1 + \epsilon + \frac{P}{\rho_0}$. Also

$$T^{\mu}{}_{\nu} = T^{\mu\delta} g_{\delta\nu}$$

[2]:
```
# Step 2.a: First define h, the enthalpy:
def compute_enthalpy(rho_b,P,epsilon):
    global h
    h = 1 + epsilon + P/rho_b

# Step 2.b: Define T^{mu nu} (a 4-dimensional tensor)
def compute_T4UU(gammaDD,betaU,alpha, rho_b,P,epsilon,u4U):
    global T4UU

    compute_enthalpy(rho_b,P,epsilon)
    # Then define g^{mu nu} in terms of the ADM quantities:
    import BSSN.ADMBSSN_tofrom_4metric as AB4m
    AB4m.g4UU_ito_BSSN_or_ADM("ADM",gammaDD,betaU,alpha)

    # Finally compute T^{mu nu}
    T4UU = ixp.zerorank2(DIM=4)
    for mu in range(4):
        for nu in range(4):
            T4UU[mu][nu] = rho_b * h * u4U[mu]*u4U[nu] + P*AB4m.g4UU[mu][nu]

# Step 2.c: Define T^{mu}_{nu} (a 4-dimensional tensor)
def compute_T4UD(gammaDD,betaU,alpha, T4UU):
    global T4UD
    # Next compute T^mu_nu = T^{mu delta} g_{delta nu}, needed for S_tilde flux.
    # First we'll need g_{alpha nu} in terms of ADM quantities:
    import BSSN.ADMBSSN_tofrom_4metric as AB4m
    AB4m.g4DD_ito_BSSN_or_ADM("ADM",gammaDD,betaU,alpha)
    T4UD = ixp.zerorank2(DIM=4)
    for mu in range(4):
        for nu in range(4):
            for delta in range(4):
                T4UD[mu][nu] += T4UU[mu][delta]*AB4m.g4DD[delta][nu]
```


Special thanks to Patrick Nelson & Zach Etienne

IllinoisGRMHD: update summary

- IllinoisGRMHD updates
  - ★ Public IGM is available at: https://github.com/zachetienne/nrpytutorial/tree/master/IllinoisGRMHD
    - ✳ Documented and generated using pedagogical Jupyter notebooks
    - ✳ Contains latest PP/Hybrid EOS
      - ✓ Validated with single TOV star tests, and full BNS merger simulations
  - ★ Ongoing work (stay tuned!)
    - ✳ Used NRPy+ to convert complex expressions into highly optimized C code
    - ✳ Tabulated EOS support under development; temperature and electron fraction added; major conservatives-to-primitives update
    - ✳ Neutrino leakage scheme support under development, working with Scott & Ari who have implemented the same infrastructure to **HARM3D**

Thank you!