Psi4's DFT and SAPT(DFT) Design Architecture: Successes, Failures, and Lessons

Daniel G. A. Smith

The Molecular Sciences Software Institute

MolSSI

SAPT(DFT) Requirements

Requirements:

- New DFT XC kernels
- DFT XC kernel 3rd-order derivatives
- Gradient-Regularized Asymptotic Correction
- SCF/DFT – Linear Response
- SCF/DFT – Time-dependent response

LibXC + Stuff

Orbital Hessians

"Make it faster"

Pickups:

- VV10
- CAM/DSD Functionals
- "Modern" Functionals (wB97M-V)
- DFT Hessians
- TDDFT
- Generalized solvers

DFT Overview

$$FC = SC\epsilon$$

HF: $F = H + 2J - K$

DFT: $F = H + 2J - \alpha K - \beta K^{\omega} + V$

V Permutations:

- X Functionals
- C Functionals
- VV10
- GRAC
- Range-Separated Parameter
- …

Other Effects:

- MP2
- D3/D4 Dispersion
- DAS Dispersion
- GCP Corrections
- BSIE Corrections
- …

"John Von Neumann would just look at this and sigh"

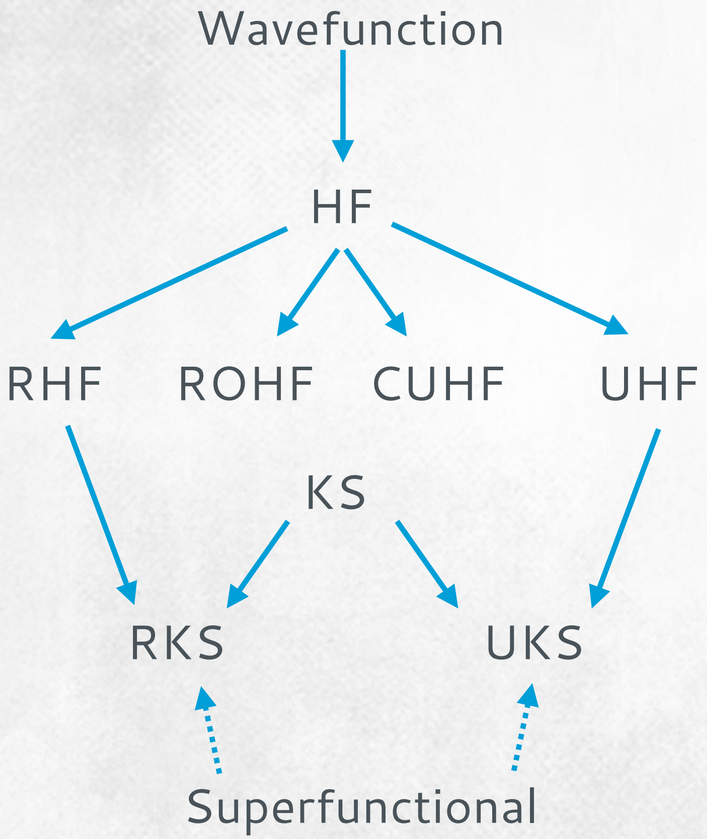
SCF Class Structure

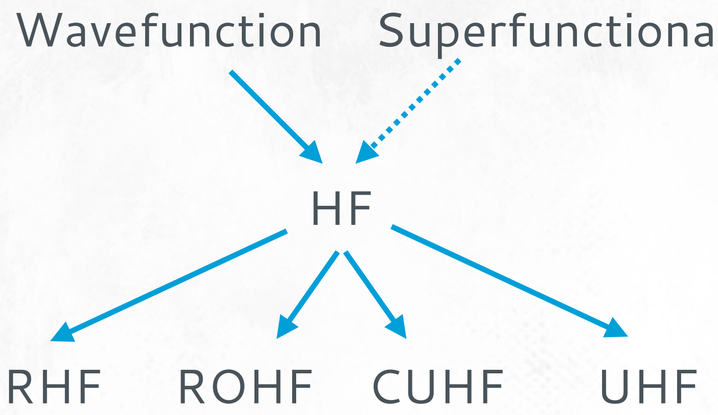
HF: $F = H + 2J - K$

DFT: $F = H + 2J - \alpha K - \beta K^{\omega} + V$

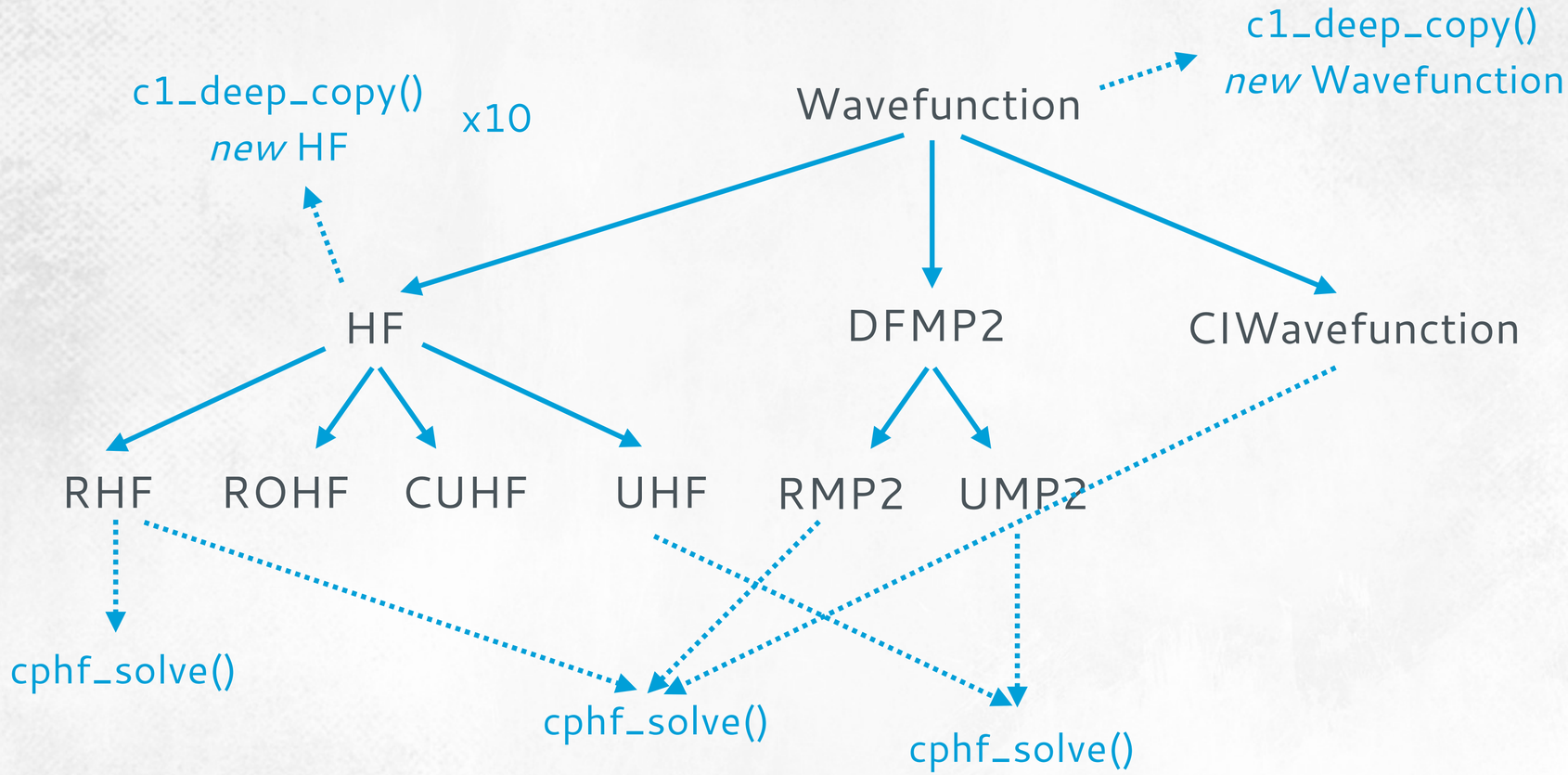
Superfunctional

- Parameters
- X/C Functionals
- Post-processing





Class Heavy Design



Consider: Functional Programming

rhf_cphf_solve(...)

uhf_cphf_solve(...)

```
def build_b2plyp_d3m(name, npoints, deriv, restricted):

    # Call this first
    sup = core.SuperFunctional.blank()
    sup.set_max_points(npoints)
    sup.set_deriv(deriv)

    # => User-Customization <= #

    # Add member functionals
    becke = core.LibXCFunctional('XC_GGA_X_B88', restricted)
    becke.set_alpha(0.47)
    sup.add_x_functional(becke)
    lyp = core.LibXCFunctional('XC_GGA_C_LYP', restricted)
    lyp.set_alpha(0.73)
    sup.add_c_functional(lyp)

    # Set GKS up after adding functionals
    sup.set_x_alpha(0.53)
    sup.set_c_alpha(0.27)

    # => End User-Customization <= #

    # Call this last
    sup.allocate()
    return (sup, '-D3M')
```

```
psi4.energy("SCF", dft_functional=build_b2plyp_d3m)
```

Superfunc

- Full DFT customization
- Built completely in Python
- Can be passed into SCF
- The SCF "Functional" is no X/C kernels and 100% exact exchange

LibXC Kernels

- 250+ DFT Primitives
- First, second, and third derivatives
- Up-to-date:
  - HSE
  - ωB97M-V
  - SCAN
  - MN15
- ~10 contributors
- Implemented in 23+ codes
- Fortran and C++ interfaces
- Actively developed (some issues)

```
void xc_gga(const xc_func_type *p, int np,
            double *rho, double *sigma,
            double *exc, double *vrho,
            double *vsigma, double *v2rho2,
            double *v2rhosigma,
            double *v2sigma2);
```

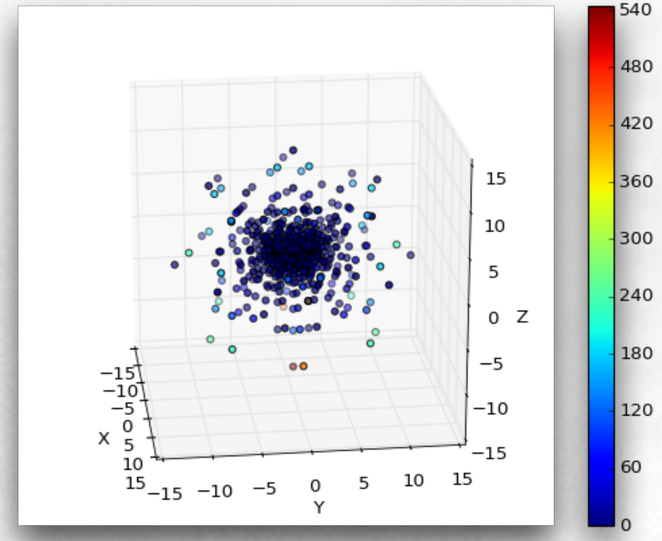
V Builds [grids]

$$F = H + 2J - \alpha K - \beta K^{\omega} + \boxed{V}$$

$$V = f^{xc}[\rho(\hat{r})] \qquad E_x^{LDA}[\rho] = -\frac{3}{4}\left(\frac{3}{\pi}\right)^{\frac{1}{3}} \int \rho(\hat{r})^{\frac{4}{3}} d\hat{r}$$

We also need the derivatives!

Quadrature Grid



Collocation

Density on a grid

$$\phi_{\mu}^{P} D_{\mu\nu} \phi_{\nu}^{P} \rightarrow \rho^{P}$$

Arbitrary density

V Integration

Cast AO density to grid

$$\mathcal{O}p\mu \rightarrow \phi_\mu^P D_{\mu\nu} \phi_\nu^P \rightarrow \rho^P$$

$$\mathcal{O}p\mu^2$$

Co-location is sparse and is broken into blocks

Perfect place to thread!

Evaluate all XC kernels and required derivatives

$$E_x^{GGA}, \delta\rho\,\delta\gamma^x, \delta\rho\,\delta\gamma^y, \ldots, \delta\gamma\,\delta\gamma \qquad \mathcal{O}p$$

Cast partial derivatives back to AO space

$$V_{\mu\nu} \leftarrow \phi_\mu^{Px} f_{\delta\gamma}^{xc}[\rho(\hat{r})]\phi_\nu^P$$

$$\mathcal{O}p\mu^2$$

Partials and chain rule can get out of hand →

```
// => GGA contribution <= //
if (ansatz >= 1) {
    double* gamma_aa = pworker->point_value("GAMMA_AA")->pointer();
    double* gamma_ab = pworker->point_value("GAMMA_AB")->pointer();
    double* gamma_bb = pworker->point_value("GAMMA_BB")->pointer();

    double* v_gamma_aa = vals["V_GAMMA_AA"]->pointer();
    double* v_gamma_ab = vals["V_GAMMA_AB"]->pointer();
    double* v_gamma_bb = vals["V_GAMMA_BB"]->pointer();

    double* v2_gamma_aa_gamma_aa = vals["V_GAMMA_AA_GAMMA_AA"]->pointer();
    double* v2_gamma_aa_gamma_ab = vals["V_GAMMA_AA_GAMMA_AB"]->pointer();
    double* v2_gamma_aa_gamma_bb = vals["V_GAMMA_AA_GAMMA_BB"]->pointer();
    double* v2_gamma_ab_gamma_ab = vals["V_GAMMA_AB_GAMMA_AB"]->pointer();
    double* v2_gamma_ab_gamma_bb = vals["V_GAMMA_AB_GAMMA_BB"]->pointer();
    double* v2_gamma_bb_gamma_bb = vals["V_GAMMA_BB_GAMMA_BB"]->pointer();

    double* v2_rho_a_gamma_aa = vals["V_RHO_A_GAMMA_AA"]->pointer();
    double* v2_rho_a_gamma_ab = vals["V_RHO_A_GAMMA_AB"]->pointer();
    double* v2_rho_a_gamma_bb = vals["V_RHO_A_GAMMA_BB"]->pointer();
    double* v2_rho_b_gamma_aa = vals["V_RHO_B_GAMMA_AA"]->pointer();
    double* v2_rho_b_gamma_ab = vals["V_RHO_B_GAMMA_AB"]->pointer();
    double* v2_rho_b_gamma_bb = vals["V_RHO_B_GAMMA_BB"]->pointer();

    double tmp_val = 0.0, v2_val_aa = 0.0, v2_val_ab = 0.0, v2_val_bb = 0.0;

    // This one is a doozy
    for (int P = 0; P < npoints; P++) {

        // V alpha contributions
        if (rho_a[P] > v2_rho_cutoff_){
            tmp_val = v2_rho_a_gamma_aa[P] * gamma_aak[P];
            tmp_val += v2_rho_a_gamma_ab[P] * gamma_abk[P];
            tmp_val += v2_rho_a_gamma_bb[P] * gamma_bbk[P];
            C_DAXPY(nlocal, (0.5 * w[P] * tmp_val), phi[P], 1, Tap[P], 1);
        }

        // V beta contributions
        if (rho_b[P] > v2_rho_cutoff_){
            tmp_val = v2_rho_b_gamma_aa[P] * gamma_aak[P];
            tmp_val += v2_rho_b_gamma_ab[P] * gamma_abk[P];
            tmp_val += v2_rho_b_gamma_bb[P] * gamma_bbk[P];
            C_DAXPY(nlocal, (0.5 * w[P] * tmp_val), phi[P], 1, Tbp[P], 1);
        }

        // => Alpha W terms <= //
        if ((rho_a[P] < v2_rho_cutoff_) || (rho_b[P] < v2_rho_cutoff_)) continue;

        // rho_ak
        v2_val_aa  = v2_rho_a_gamma_aa[P] * rho_ak[P];
        v2_val_ab  = v2_rho_a_gamma_ab[P] * rho_ak[P];

        // rho_bk
        v2_val_aa += v2_rho_b_gamma_aa[P] * rho_bk[P];
        v2_val_ab += v2_rho_b_gamma_ab[P] * rho_bk[P];

        // gamma_aak
        v2_val_aa += v2_gamma_aa_gamma_aa[P] * gamma_aak[P];
        v2_val_ab += v2_gamma_aa_gamma_ab[P] * gamma_aak[P];
        v2_val_aa += v2_gamma_aa_gamma_aa[P] * gamma_aak[P];
        v2_val_ab += v2_gamma_aa_gamma_ab[P] * gamma_aak[P];

        // gamma_abk
        v2_val_aa += v2_gamma_aa_gamma_ab[P] * gamma_abk[P];
        v2_val_ab += v2_gamma_ab_gamma_ab[P] * gamma_abk[P];

        // gamma_bbk
        v2_val_aa += v2_gamma_aa_gamma_bb[P] * gamma_bbk[P];
        v2_val_ab += v2_gamma_ab_gamma_bb[P] * gamma_bbk[P];

        // Wx
        tmp_val  = 2.0 * v_gamma_aa[P] * rho_ak_x[P];
        tmp_val += v_gamma_ab[P] * rho_bk_x[P];
        tmp_val += 2.0 * v2_val_aa * rho_ax[P];
        tmp_val += v2_val_ab * rho_bx[P];
        tmp_val *= w[P];

        C_DAXPY(nlocal, tmp_val, phi_x[P], 1, Tap[P], 1);

        // Wy
        tmp_val  = 2.0 * v_gamma_aa[P] * rho_ak_y[P];
        tmp_val += v_gamma_ab[P] * rho_bk_y[P];
        tmp_val += 2.0 * v2_val_aa * rho_ay[P];
        tmp_val += v2_val_ab * rho_by[P];
        tmp_val *= w[P];

        C_DAXPY(nlocal, tmp_val, phi_y[P], 1, Tap[P], 1);

        // Wz
        tmp_val  = 2.0 * v_gamma_aa[P] * rho_ak_z[P];
        tmp_val += v_gamma_ab[P] * rho_bk_z[P];
        tmp_val += 2.0 * v2_val_aa * rho_az[P];
        tmp_val += v2_val_ab * rho_bz[P];
        tmp_val *= w[P];

        C_DAXPY(nlocal, tmp_val, phi_z[P], 1, Tap[P], 1);

        // => Beta W terms <= //

        // rho_ak
        v2_val_bb  = v2_rho_a_gamma_bb[P] * rho_ak[P];
        v2_val_ab  = v2_rho_a_gamma_ab[P] * rho_ak[P];

        // rho_bk
        v2_val_bb += v2_rho_b_gamma_bb[P] * rho_bk[P];
        v2_val_ab += v2_rho_b_gamma_ab[P] * rho_bk[P];

        // gamma_bbk
        v2_val_bb += v2_gamma_bb_gamma_bb[P] * gamma_bbk[P];
        v2_val_ab += v2_gamma_ab_gamma_bb[P] * gamma_bbk[P];

        // gamma_abk
        v2_val_bb += v2_gamma_ab_gamma_bb[P] * gamma_abk[P];
        v2_val_ab += v2_gamma_ab_gamma_ab[P] * gamma_abk[P];

        // gamma_aak
        v2_val_bb += v2_gamma_aa_gamma_bb[P] * gamma_aak[P];
        v2_val_ab += v2_gamma_aa_gamma_ab[P] * gamma_aak[P];

        // Wx
        tmp_val  = 2.0 * v_gamma_bb[P] * rho_bk_x[P];
        tmp_val += v_gamma_ab[P] * rho_ak_x[P];
        tmp_val += 2.0 * v2_val_bb * rho_bx[P];
        tmp_val += v2_val_ab * rho_ax[P];
        tmp_val *= w[P];

        C_DAXPY(nlocal, tmp_val, phi_x[P], 1, Tbp[P], 1);

        // Wy
        tmp_val  = 2.0 * v_gamma_bb[P] * rho_bk_y[P];
        tmp_val += v_gamma_ab[P] * rho_ak_y[P];
        tmp_val  = 2.0 * v_gamma_bb[P] * rho_bk_y[P];
        tmp_val += v_gamma_ab[P] * rho_ak_y[P];
        tmp_val += 2.0 * v2_val_bb * rho_by[P];
        tmp_val += v2_val_ab * rho_ay[P];
        tmp_val *= w[P];

        C_DAXPY(nlocal, tmp_val, phi_y[P], 1, Tbp[P], 1);

        // Wz
        tmp_val  = 2.0 * v_gamma_bb[P] * rho_bk_z[P];
        tmp_val += v_gamma_ab[P] * rho_ak_z[P];
        tmp_val += 2.0 * v2_val_bb * rho_bz[P];
        tmp_val += v2_val_ab * rho_az[P];
        tmp_val *= w[P];

        C_DAXPY(nlocal, tmp_val, phi_z[P], 1, Tbp[P], 1);

    }
}

// Put it all together
C_DGEMM('T', 'N', nlocal, nlocal, npoints, 1.0, phi[0], max_functions, Tap[0],
        max_functions, 0.0, Vax_localp[0], max_functions);
C_DGEMM('T', 'N', nlocal, nlocal, npoints, 1.0, phi[0], max_functions, Tbp[0],
        max_functions, 0.0, Vbx_localp[0], max_functions);

// Symmetrization (V is *always* Hermitian)
for (int m = 0; m < nlocal; m++) {
    for (int n = 0; n <= m; n++) {
        Vax_localp[m][n] = Vax_localp[n][m] = Vax_localp[m][n] + Vax_localp[n][m];
        Vbx_localp[m][n] = Vbx_localp[n][m] = Vbx_localp[m][n] + Vbx_localp[n][m];
    }
}
// R_Vax_local[rank]->print();
// R_Vbx_local[rank]->print();


// => Unpacking <= //
double** Vaxp = Vax_AO[dindex]->pointer();
double** Vbxp = Vbx_AO[dindex]->pointer();
for (int ml = 0; ml < nlocal; ml++) {
    int mg = function_map[ml];
    for (int nl = 0; nl < ml; nl++) {
        int ng = function_map[nl];

        #pragma omp atomic update
        Vaxp[mg][ng] += Vax_localp[ml][nl];
        #pragma omp atomic update
        Vaxp[ng][mg] += Vax_localp[ml][nl];
```

DFT Update Benefits

- Hundreds of new functionals (more than Q-Chem/Gaussian/NWChem/GAMESS)
  - GRAC, VV10, CAM, DSD

- V builds are threaded and optimized, up to 12x speedup on 6 cores
- Overall DFT can be up to 10x faster for meta functionals

- Time dependent/independent Hessian-vector elements are available
- DFT linear response implemented
  - TDDFT just a step away
- Second-order KS convergence (SOUKS)
  - Stability analysis could be done easily

```
"XC_HYB_GGA_XC_CAM_B3LYP",        # 433  CAM version of B3LYP
"XC_HYB_GGA_XC_TUNED_CAM_B3LYP", # 434  CAM version of B3LYP tuned for excitations
"XC_HYB_GGA_XC_BHANDH",           # 435  Becke half-and-half
"XC_HYB_GGA_XC_BHANDHLYP",        # 436  Becke half-and-half with B88 exchange
"XC_HYB_GGA_XC_MB3LYP_RC04",      # 437  B3LYP with RC04 LDA
"XC_HYB_GGA_XC_MPWLYP1M",         # 453  MPW with 1 par. for metals/LYP
"XC_HYB_GGA_XC_REVB3LYP",         # 454  Revised B3LYP
"XC_HYB_GGA_XC_CAMY_BLYP",        # 455  BLYP with yukawa screening
"XC_HYB_GGA_XC_PBE0_13",          # 456  PBE0-1/3
"XC_HYB_GGA_XC_B3LYPs",           # 459  B3LYP* functional
"XC_HYB_GGA_XC_WB97",             # 463  Chai and Head-Gordon
"XC_HYB_GGA_XC_WB97X",            # 464  Chai and Head-Gordon
"XC_HYB_GGA_XC_LRC_WPBEH",        # 465  Long-range corrected functional by Rorhdanz et al
"XC_HYB_GGA_XC_WB97X_V",          # 466  Mardirossian and Head-Gordon
"XC_HYB_GGA_XC_LCY_PBE",          # 467  PBE with yukawa screening
"XC_HYB_GGA_XC_LCY_BLYP",         # 468  BLYP with yukawa screening
"XC_HYB_GGA_XC_LC_VV10",          # 469  Vydrov and Van Voorhis
"XC_HYB_GGA_XC_CAMY_B3LYP",       # 470  B3LYP with Yukawa screening
"XC_HYB_GGA_XC_WB97X_D",          # 471  Chai and Head-Gordon
"XC_HYB_GGA_XC_HPBEINT",          # 472  hPBEint
"XC_HYB_GGA_XC_LRC_WPBE",         # 473  Long-range corrected functional by Rorhdanz et al
"XC_HYB_GGA_XC_B3LYP5",           # 475  B3LYP with VWN functional 5 instead of RPA
"XC_HYB_GGA_XC_EDF2",             # 476  Empirical functional from Lin, George and Gill
"XC_HYB_GGA_XC_CAP0",             # 477  Correct Asymptotic Potential hybrid
"XC_MGGA_XC_ZLP",                 #  42  Zhao, Levy & Parr, Eq. (21)
"XC_MGGA_XC_OTPSS_D",             #  64  oTPSS_D functional of Goerigk and Grimme
"XC_MGGA_XC_TPSSLYP1W",           # 242  Functionals fitted for water
"XC_MGGA_XC_B97M_V",              # 254  Mardirossian and Head-Gordon
"XC_HYB_MGGA_XC_M05",             # 438  M05 functional from Minnesota
"XC_HYB_MGGA_XC_M05_2X",          # 439  M05-2X functional from Minnesota
"XC_HYB_MGGA_XC_B88B95",          # 440  Mixture of B88 with BC95 (B1B95)
"XC_HYB_MGGA_XC_B86B95",          # 441  Mixture of B86 with BC95
"XC_HYB_MGGA_XC_PW86B95",         # 442  Mixture of PW86 with BC95
"XC_HYB_MGGA_XC_BB1K",            # 443  Mixture of B88 with BC95 from Zhao and Truhlar
"XC_HYB_MGGA_XC_M06_HF",          # 444  M06-HF functional from Minnesota
"XC_HYB_MGGA_XC_MPW1B95",         # 445  Mixture of mPW91 with BC95 from Zhao and Truhlar
"XC_HYB_MGGA_XC_MPWB1K",          # 446  Mixture of mPW91 with BC95 for kinetics
"XC_HYB_MGGA_XC_X1B95",           # 447  Mixture of X with BC95
"XC_HYB_MGGA_XC_XB1K",            # 448  Mixture of X with BC95 for kinetics
"XC_HYB_MGGA_XC_M06",             # 449  M06 functional from Minnesota
"XC_HYB_MGGA_XC_M06_2X",          # 450  M06-2X functional from Minnesota
"XC_HYB_MGGA_XC_PW6B95",          # 451  Mixture of PW91 with BC95 from Zhao and Truhlar
"XC_HYB_MGGA_XC_PWB6K",           # 452  Mixture of PW91 with BC95 from Zhao and Truhlar for
"XC_HYB_MGGA_XC_TPSSH",           # 457  TPSS hybrid
"XC_HYB_MGGA_XC_REVTPSSH",        # 458  revTPSS hybrid
"XC_HYB_MGGA_XC_M08_HX",          # 460  M08-HX functional from Minnesota
"XC_HYB_MGGA_XC_M08_SO",          # 461  M08-SO functional from Minnesota
"XC_HYB_MGGA_XC_M11",             # 462  M11    functional from Minnesota
"XC_HYB_MGGA_XC_WB97M_V",         # 531  Mardirossian and Head-Gordon
```

DFT Update Lessons

Successes:

- XC Kernel generation is offloaded
- New object-level threading model for DFT
- LibXCFunctional derived from Functional

Failures:

- CP-SCF is implemented on HF derived Wavefunctions
- Missing SG-0/SG-1/SG-2 grids (I think)
- Optimized before profiling (collocation matrix rate limiting)
- An enormous amount of potential currently wasted

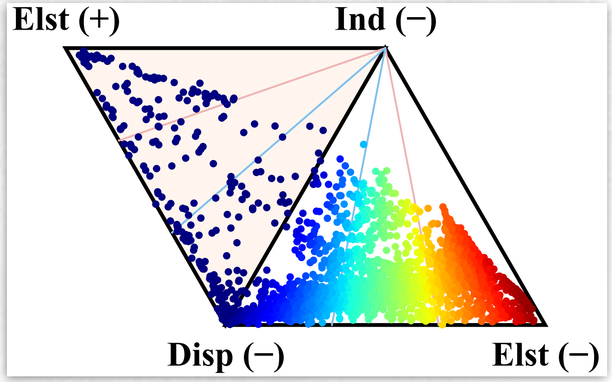
Lessons:

- Large vertical class hierarchy gets messy and restrictive
  - What happens when a method is only implemented on Wavefunction?
- Pre-factors matter more than scaling in our normal size regime

$$\frac{4}{3}\frac{\pi 3^{\frac{2}{3}}}{4} \approx 0.529\ldots$$

SAPT Introduction

- Obtains interaction energies by an intermolecular perturbation
- Four fundamental forces:
  - Electrostatics – Charge-charge interaction
  - Exchange – Pauli repulsion
  - Induction – Relaxation of a monomer in the field of the other
  - Dispersion – Correlation between monomers
- SAPT Hamiltonian:

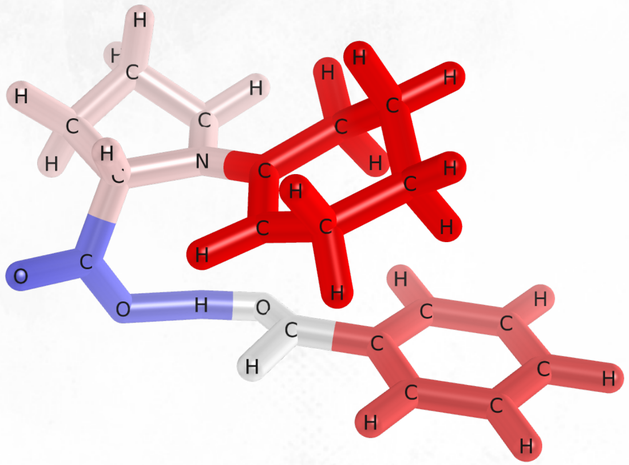


Intramonomer Correlation

$$H = F_A + F_B + \zeta V + \xi W_A + \eta W_B$$

Fock Operators

Intermonomer Correlation



Thanks to Brandon Bakr for F-SAPT image

SAPT Terms

What does it really take?

$$H = F_A + F_B + \zeta V + \xi W_A + \eta W_B$$

Difficult, tedious, etc

$$E^1_{\text{elst}} = D^A_{\mu\nu}(\mu\nu|\lambda\sigma)D^B_{\lambda\sigma} + D^A_{\mu\nu}V^B_{\mu\nu} + D^B_{\mu\nu}V^A_{\mu\nu} + E^{\text{int}}_{\text{nuc}}$$

e^- / p^+ attraction

e^- / e^- repulsion

p^+ / p^+ repulsion

$$E^1_{\text{exch}}(S^2) = \text{tr}\left(D^A S D^B S P^A \omega_A + D^B S D^A S P^B \omega_B + P^A S D^B \cdot K[D^A S P^B]\right)$$

More complex, but still densities

SAPT Terms

SCF Wavefunctions

Cache the Wavefunctions and densities

```
scf_eA, scf_wfnA = psi4.energy("B3LYP/cc-pVDZ", return_wfn=True, molecule=monomerA)
scf_eB, scf_wfnB = psi4.energy("B3LYP/cc-pVDZ", return_wfn=True, molecule=monomerB)

cache = {}
cache["D_A"] = scf_wfnA.Da()
cache["Cocc_A"] = scf_wfnA.Ca_subset("AO", "OCC")

cache["D_B"] = scf_wfnB.Da()
…
```

Cache any integrals

```
mintsA = psi4.core.MintsHelper(scf_wfnA.basisset())
mintsB = psi4.core.MintsHelper(scf_wfnB.basisset())

cache["V_A"] = mintsA.ao_potential()
cache["V_B"] = mintsB.ao_potenital()
```

SAPT Terms

Computing Electrostatics

$$E^1_{\text{elst}} = D^A_{\mu\nu}(\mu\nu|\lambda\sigma)D^B_{\lambda\sigma} + D^A_{\mu\nu}V^B_{\mu\nu} + D^B_{\mu\nu}V^A_{\mu\nu} + E^{\text{int}}_{\text{nuc}}$$

```
# Coulomb contraction
jk.clear()
jk.add_C_left(cache["Cocc_A"])
jk.compute()

J_A = jk.J()[0]
```

$$J_{\mu\nu}[D_{\lambda\sigma}] = (\mu\nu|\lambda\sigma)C^{\text{l}}_{\lambda i}C^{\text{r}}_{\sigma i}$$
$$K_{\mu\nu}[D_{\lambda\sigma}] = (\mu\lambda|\nu\sigma)C^{\text{l}}_{\lambda i}C^{\text{r}}_{\sigma i}$$

```
# Contract terms
Elst10  = cache["D_B"].vector_dot(J_A)
Elst10 += cache["D_A"].vector_dot(cache["V_B"])
Elst10 += cache["D_B"].vector_dot(cache["V_A"])
Elst10 += cache["nuclear_repulsion_energy"]
```

SAPT Induction

Psi4 Wavefunction Object

- Induction amplitudes are recovered from coupled-perturbed (CP) quantities. (N^4)
- Reference can be:
  - RHF, UHF, RKS, UKS
  - Fractions of exact exchange
  - Range-separated functionals
  - Asymptotic corrections

$$\mathbf{H}^{(2)}\mathbf{x}^{\mathbf{A}} = \boldsymbol{\omega}^{\mathbf{B}}$$

Electrostatic Potential

$$E_{\text{ind}}^{(2)} = -\mathbf{x}^{A}\boldsymbol{\omega}^{B} - \mathbf{x}^{B}\boldsymbol{\omega}^{A}$$

CP Response

```
scf_e, scf_wfn = psi4.energy("CAM-B3LYP/cc-pVDZ", return_wfn=True)

X_A = scf_wfn.cphf_solve(cache["W_B"], **options)

print(-1.0 * X_A.vector_dot(cache["W_B"])
```

SAPT(DFT) Update Lessons

Successes:

- Elst and Exch based on functional programming
  - Valid for any SCF theory
- Reusing some Disp and Exch-Disp from F-SAPT through Python and functional-like programming

Failures:

- **Third (!)** SAPT0 solver implemented.
  - All still active, but tested to get matching results.
- Documentationless at the moment

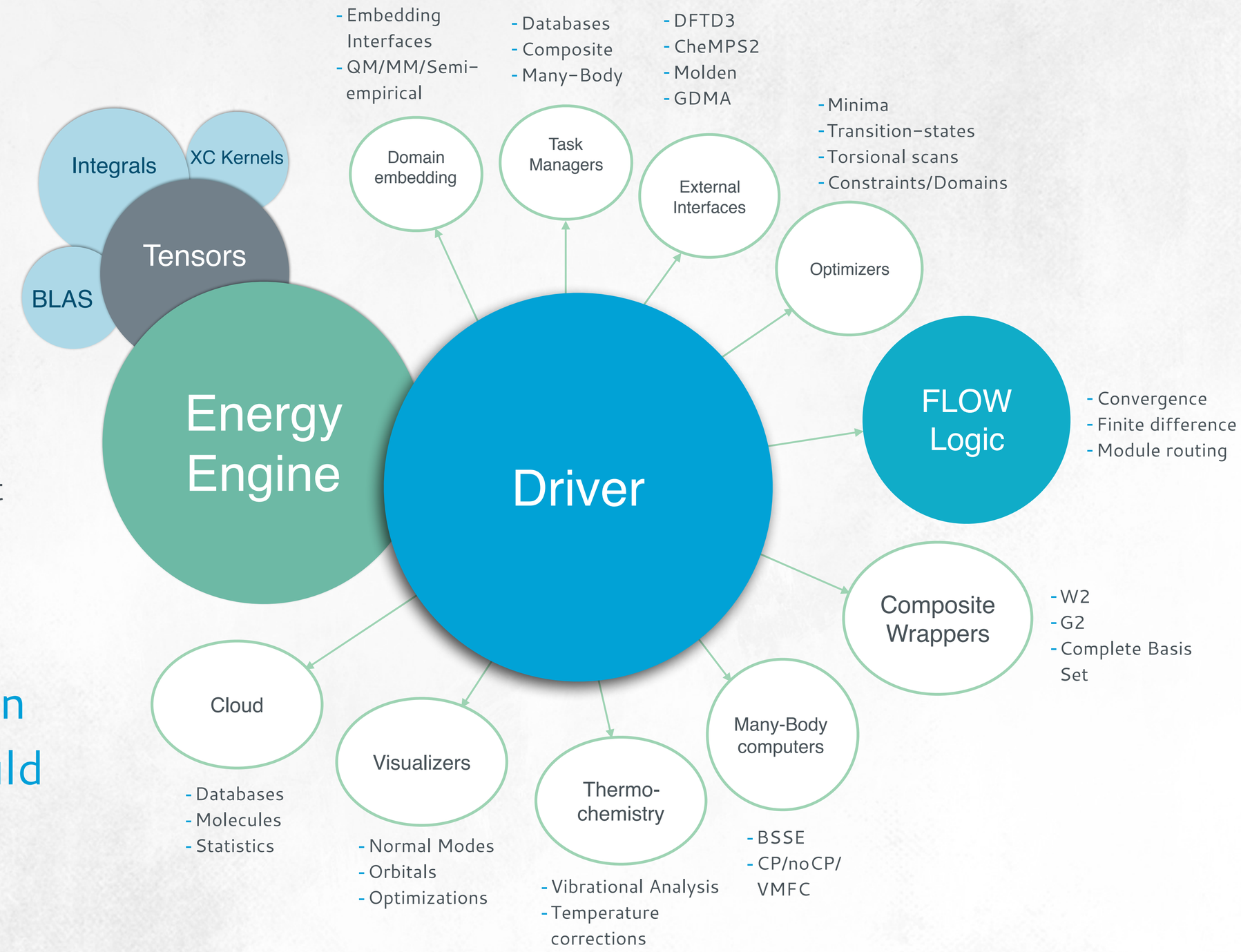
Lessons:

- A discussion about deprecation and switching backends needs to be held
- Start mixing in functional-style programming
- If it can be written in Python without performance degradation, do that

a QM program

- ***A*** diagram for modularity
- Each area is of an interest to specific groups

More modules than any one group would *want* to work on



Free as in a puppy

- We are hindered by technical debt
  - Software will continuously evolve
  - Hardware will continuously evolve
- We are *used to* many warts
  - Aging disk I/O
  - Little tensor support
  - SCF convergence issues
  - Large duplication of effort
- We are *missing* some easy tricks
  - ~~DFT~~ MP2 Hessians
  - TDDFT
  - MM (Hessians, opts, etc)
  - Visualization access
- Development has become *spotty* again
  - 2–3 people focusing on Psi
  - Large oscillations in effort (trackable on git, happened since the 80's)

- This is ***ok*** if Psi wants to maintain the status quo
- This is ***not ok*** if Psi wants to take over the world

Psi4 Versioning

[Sorry Lori]

- Psi 1, 2, 3, 4 Major Version means a language change
  - Psi4 beta -> C++ program with some Python bindings
  - Psi4 1.0 -> Python API is a bit of an afterthought
  - Psi4 1.1 -> Inverted, recent Python API
- Python API is constrained by our fully C++ past
  - Psithon is invaluable
  - Increasingly large population using Psi4 as a library
- If we want a really good API, we probably need to break it, and we need to think about it very hard
- Psi5 -> A Python program with a C++ backend for heavy lifting


I WANT YOU
FOR PSI4
[GITHUB.COM/PSI4]