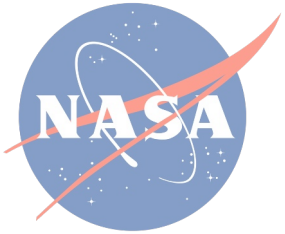


CEA2022: A Modernization of NASA Glenn's Software CEA (Chemical Equilibrium with Applications)

Mark Leader

Thomas Lavelle, Xiao-yen Wang, Kevin Dickens, Michael McTague

NASA Glenn Research Center

Jeffrey Hill

NASA Ames Research Center

Thermal & Fluids Analysis Workshop 2024
August 29, 2024

What is CEA?

Chemical Equilibrium with Applications

CEA2: current version of legacy code (released in 2002)

- robust chemical equilibrium solver with numerous applications
  - Turbojet engines, rocket engines, shock wave and detonation problems
  - Used widely in government, industry, and academia
- Culmination of 50+ years of development at NASA GRC

CEA2022: modernized CEA, releasing this year*

- backwards-compatible, consistent re-write

What can CEA do?

Chemical equilibrium solver

- Need two fixed states: temperature & pressure (tp), enthalpy & pressure (hp), entropy & pressure (sp), temperature & volume (tv), energy & volume (uv), entropy & volume (sv)
- Supports over 2000 possible products, and 62 additional reactants
- Assumes ideal gas behavior; supports condensed species, but assumes they take negligible volume
- Option to include ionic products
- Transport properties optionally computed
- Negative reactants*
- Inert reactants* (approximation for incomplete combustion)
- Derivatives for optimization*

Rocket solver

- Infinite-area combustor (IAC) or finite-area combustor (FAC)
- Solves properties in chamber, throat, and exit
- Exit conditions specified by pressure or area ratios
- Computes pressure ratio/area ratio, Mach, coefficient of thrust, specific impulse, and vacuum specific impulse

Shock solver

- Solves conditions in a shock tube for reflected and incident shocks
- Uses initial Mach number or incident shock velocity

Detonation solver

- Solves Chapman-Jouguet detonation parameters

*new feature

Why are we doing this update?

- Modernization: Rewrite CEA in a modern language using modern software development practices
- Enhancements: Add capability to the tool based on feedback solicited from the CEA user-community
- Legacy: Develop new NASA engineers as experts in CEA to retain extensive knowledge at GRC

Code Modernization

- Modern programming language (Fortran 2008, object-oriented)
- Version-control using git
- Software development using continuous testing
- Cross-platform build support (CMake)

Feature Enhancement

- Run cases in parallel (thread safe solves)
- Thermodynamic and transport database updates
- Allow for negative reactants (e.g. water precipitating from the flow)
- Analytic derivatives to enable gradient-based optimization

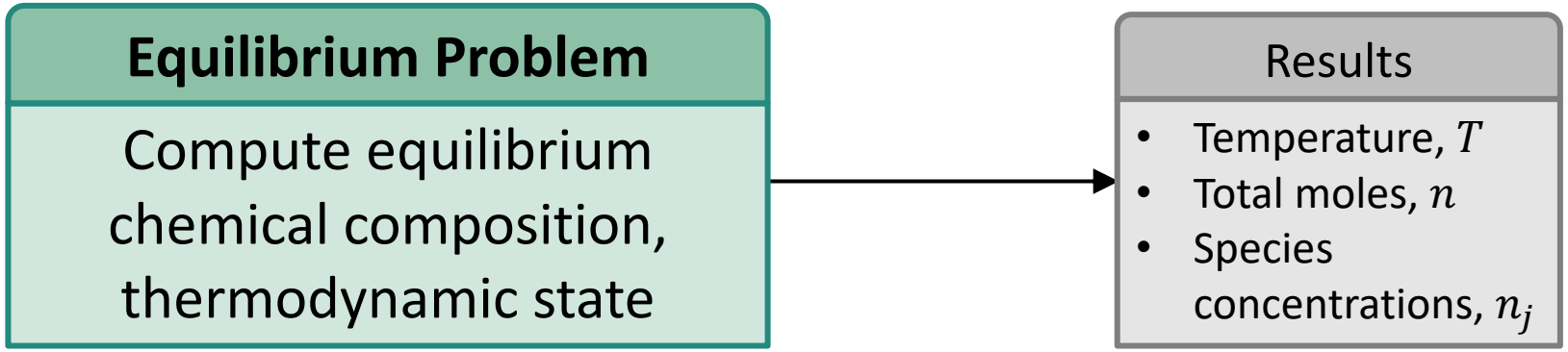
Interface Improvements

- Make CEA a reusable library: add APIs for multiple programming languages
- Python, Matlab, Excel, Fortran, C, C++
- Support flow-solver integration (CFD)

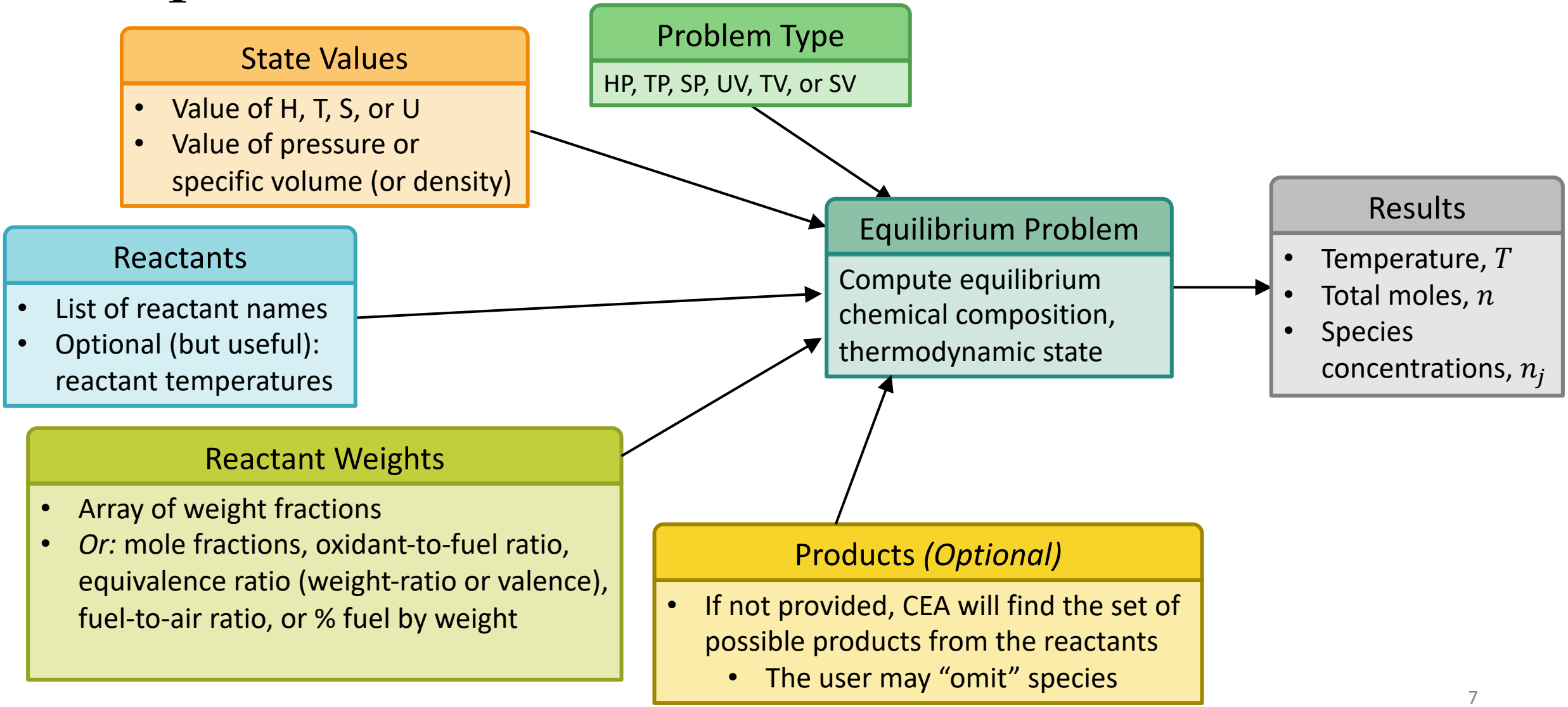
Outline

- Methodology: how CEA works, and how to use it well
- Interfaces: how to interact with CEA
- Validation examples: comparison of CEA2022 results against CEA2
- Conclusion

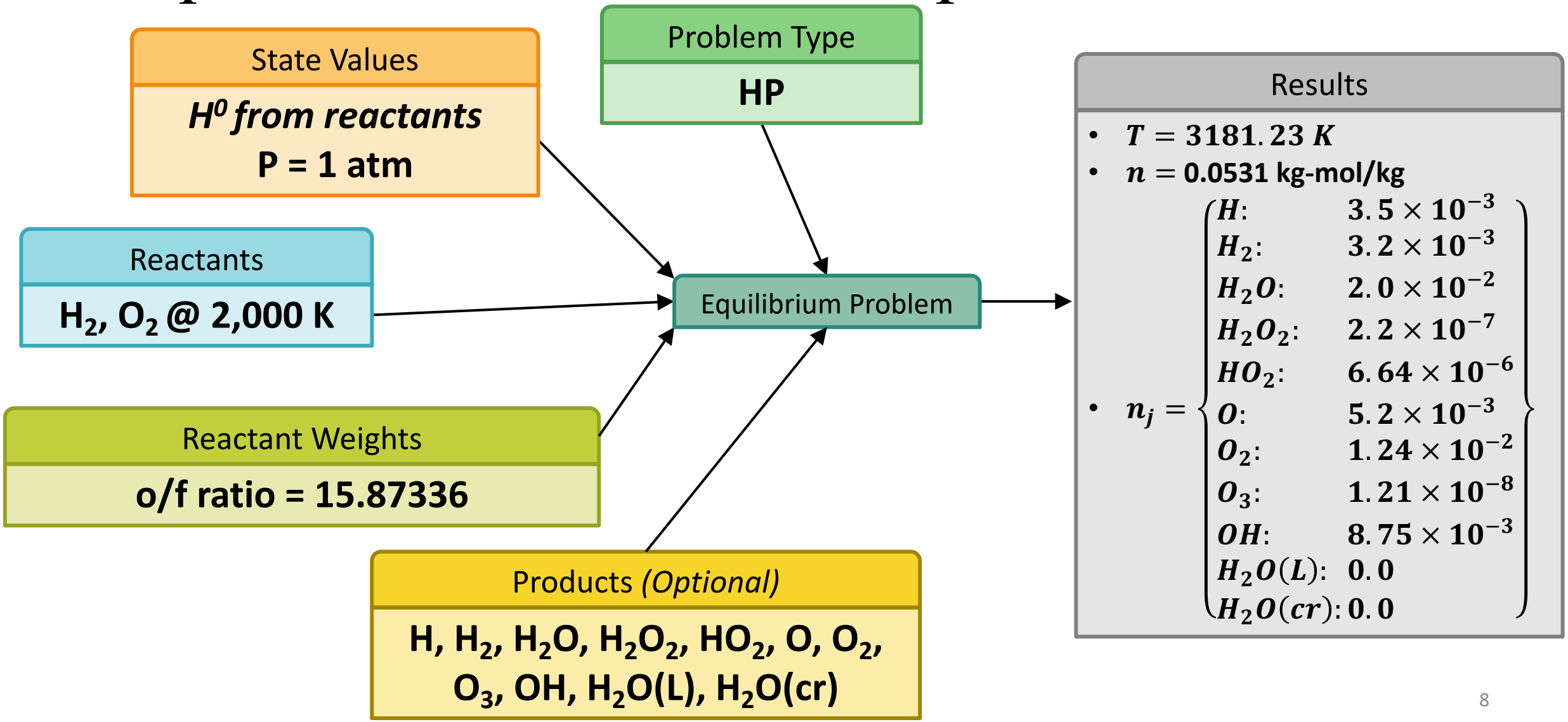
Equilibrium Problem



Equilibrium Problem



Equilibrium Problem Example



How do we solve the equilibrium problem?

Basic assumption: $PV = n_g RT$

Equilibrium is based on minimization of Gibbs' energy: $g = \sum_{j}^{NS} \mu_j n_j$

But, the equilibrium solution has to end with the same amount of each element that we stated with:

$$\sum_{j}^{NS} a_{ij} n_j - b_i^{\circ} = 0, \; i = 1, \ldots, NE$$

How do we solve the equilibrium problem?

Constrained minimization problem → form a Lagrangian:

$$\mathcal{L} = g + \sum_{i}^{NS} \lambda_i \left(b_i - b_i^{\circ} \right)$$

$$\delta \mathcal{L} = 0$$

Expand the previous equation to form a nonlinear matrix system of equations:

$$\mathbf{R}(\mathbf{x}, \mathbf{u}) = \mathbf{A}(\mathbf{x}, \mathbf{u})\mathbf{u} - \mathbf{f}(\mathbf{x}, \mathbf{u}) = 0$$

Use a Newton-method solver with a damped update to converge the solution

How do we solve the equilibrium problem?

Solution variables:

- Species concentrations: $\mathbf{n} = [n_1, \ldots, n_j]^T$
- Total mixture moles: n
- Mixture temperature: T

An initial guess is required the the iterative solution procedure:

$$\mathbf{n} = \left[\frac{0.1}{NS}, \ldots, \frac{0.1}{NS}\right]^T \text{ kg-mol/kg}$$

$$n = 0.1 \text{ kg-mol/kg}$$

$$T = 3800 \text{ K}$$

Iteration Procedure

Use a Newton-method solver with a damped update to converge the solution

Solution update variables:

1) Change in log of gas species concentrations $\Delta \ln(n_j), \quad j = 1, \ldots, NG$

2) Change in condensed species concentrations $\Delta n_j, \quad j = NG+1, \ldots, NS$

3) Change in log-moles $\Delta \ln(n)$

4) Change in log-temperature $\Delta \ln(T)$

Example update:

Damped update factor

$$\ln(n_j)^{k+1} = \ln(n_j)^k + \lambda(\Delta \ln(n_j)), \quad j = 1, \ldots, NG$$

Typically: Damped update factor λ is initially < 1 for early iterations, and equal to 1 when the solution is close to convergence

Iteration Procedure

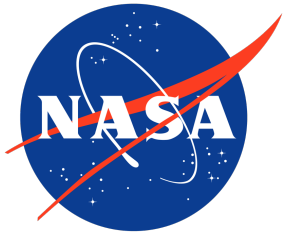
Note: internally, $\ln(n_j)$ and n_j are stored separately

- Seems redundant, but:
- Threshold is applied for species with $\ln(n_j) < \overbrace{\ln(10^{-8})}^{default} \rightarrow n_j = 0$
- $\ln(n_j)$ is continually updated, so that n_j can come in and out of the solution without loss of information from previous
- Default threshold of $\ln(10^{-8}) = -18.420681$ can be controlled using the ”trace” variable

Convergence Criteria

Converge is based on the relative size of the update variables

- If updates are sufficiently small for all variables, we have reached an equilibrium point, and the residual equation is satisfied
- Test for each of:
  - Gas species concentrations: $n_j, \quad j = 1, \ldots, NG$
  - Condensed species concentrations: $n_j, \quad j = NG + 1, \ldots, NS$
  - Total moles: n
  - Temperature: T
  - Element balance: $b_i, \quad i = 1, \ldots, NE$
  - Entropy: s
  - Modified Lagrange multipliers: $\pi_i, \quad i, \ldots, NE$



Next: special cases

- Condensed species
- Ionized species
- Negative reactant amounts
- Inert species

Condensed Species

Inclusion of condensed species violates the ideal gas assumption

* This assumption is okay as long as condensed species only take up a negligible volume

Solution procedure:

1. Start with no condensed species in the initial guess
2. Compute equilibrium with only gas phase species
3. After convergence, test if adding any condensed species lowers the Gibbs energy
4. If any meet the criteria, add the species that lowers the Gibbs energy the most, and converge the system again
5. Repeat 3 and 4 until final convergence

+ Additional considerations for handling phase change, or multiple phases of the same species

Ionized Species

CEA supports the use of ionized species

This adds an additional constraint to the problem, such that the charge-balance equation is satisfied:

$$\sum_{j=1}^{NG} a_{ej} n_j = 0$$

Charge-balance for species *j*

Negative Reactant Amounts

CEA2022 allows reactants to be specified with negative amounts

Example: water precipitating from a flow

This affects:

1. The element-balance constraint
2. The fixed-value of $H°$, $S°$, or $U°$ (in cases where the reactant mixture is used to compute those values)

* This is a non-equilibrium analysis, so be extra careful interpreting results when using this feature

Inert Species

CEA2022 allows reactants to be specified as inert

- This allows the user to specify a fixed-quantity of a species to persist in the product mixture
- Example: retain some amount of fuel to model incomplete combustion
- Only supported for certain hydrocarbons

InertC10H8,naph	InertCH4
InertO2	InertH2(L)
InertJP-5	InertJet-A(g)
InertC2H4	InertH
InertAir	InertJP-10(L)
InertJet-A(L)	InertO2(L)
InertH2	InertRP-1
InertJP-10(g)	InertO
InertJP-4	

* This is a non-equilibrium analysis, so be extra careful interpreting results when using this feature

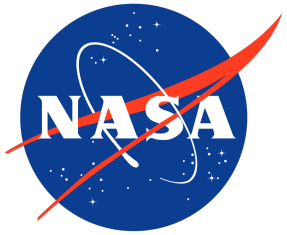
Thermodynamic Properties

Compute $\frac{C_p}{R}$, $\frac{H}{RT}$, and $\frac{S}{R}$ using piecewise polynomial curve fit data

Example: $$\frac{C_p}{R} = a_1 T^{-2} + a_2 T^{-1} + a_3 + a_4 T + a_5 T^2 + a_6 T^3 + a_7 T^4$$

- 2,012 product species currently supported, with an additional 62 reactants
- For gaseous species, curve fits are done in the intervals:
  - [200, 600] K
  - [600, 1000] K
  - [1000, 6000] K
  - [6000, 20000] K
- Condensed species are unique to each species

Thermodynamic Properties



Green propellants

Sustainable Aviation Fuel (SAF)

New species added for CEA2022:

- Ammonium DiNitramide (ADN)
- LMP-103S
- HydroxylAmmonium Nitrate (HAN)
- AF-M315E, a.k.a., ASCENT
- n-Butanol
- Biodiesel

Thermodynamic Properties

$$c_p := \left(\frac{\partial h}{\partial T}\right)_P$$

$$h = \sum n_j h_j$$

$$\left(\frac{\partial h}{\partial T}\right)_P = \sum \left(n_j \frac{\partial h_j}{\partial T} + \frac{\partial n_j}{\partial T} h_j \right)$$

$$= n_j \, c_{p,j}$$

This comes from CEA curve fit data

”frozen” heat capacity

”reaction” heat capacity

This term is included in the Jacobian matrix

Transport Properties

CEA can optionally compute mixture transport properties

Uses curve fits with least-squares coefficients to compute:

- Viscosity, η
- Thermal conductivity, λ

$$\begin{cases} \ln \eta \\ \ln \lambda \end{cases} = A \ln T + \frac{B}{T} + \frac{C}{T^2} + D$$

- Transport properties are only computed for gaseous species
- A binary interaction parameter η_{ij} is included for some pairs of species
- Prandtl number is also computed

Frozen vs. Equilibrium Transport Properties

Thermal conductivity, specific heat, and Prandtl number each have “frozen” and “reaction” contributions

The frozen (“fr”) and reaction (“re”) terms sum to equilibrium (“eq”) value:

$$\lambda_{\mathrm{eq}} = \lambda_{\mathrm{fr}} + \lambda_{\mathrm{re}}$$

Rocket Analysis

CEA can analyze rocket performance
- Can assume either an infinite-area combustor (IAC) or a finite-area combustor (FAC)

- Converge each station in order of: combustor, throat, and exit
- Equilibrium module is called at each station by default, unless “frozen” mode is used
- Frozen analysis: compute equilibrium mixture up until the frozen station; after that, the composition is fixed

Inputs:
- Reactants
- Chamber pressure
- Exit parameters:
  - Chamber pressure to exit pressure ratio, P_c/P_e
  - Or, exit area to throat area ratio, A_e/A_t
- FAC:
  - Contraction ratio, A_c/A_t
  - Or, mass flow rate per chamber area, $\dot{m}/A_c$

Shock Problems

CEA can solve shock tube problems for both incident and reflected shocks

Inputs:

- Reactants
- Un-shocked pressure and temperature
- Incident shock velocity, *or* Mach number

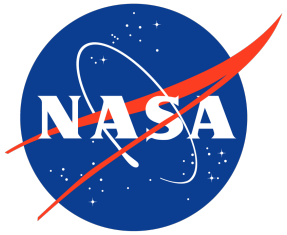
Detonation Problems

CEA can compute parameters for Chapman-Jouguet detonation problems

Inputs:

- Reactants
- Initial temperature and pressure
- Detonation velocity, *or* Mach number

Software Development Summary



Version control:

- Using git distributed version control system
- NASA GitHub for automated testing, collaboration, issue tracking, release management

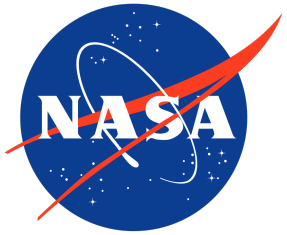
Language: Fortran 2008 (compatible subset)

- Performance: native execution, performant multi-dimensional arrays, highly optimized compilers
- Legacy CEA Compatibility: simplifies migration, test, modernization of legacy algorithms
- Software Architecture: object-oriented programming, polymorphism
- Widely available: GNU v5+ (2015), Intel 16.0+ (2015), PGI/NVIDIA 19.4+ (2019)

Testing:

- Concurrent development of unit tests exercising software at subroutine level
  - Run every time the code is compiled to make sure no bugs or unexpected behavior is introduced
  - Enables rapid verification of builds on new platform/compilers; rapid code refactoring

Software Development Summary



Thread-safe solves:

- All solve functions are thread-safe to allow running problems in parallel

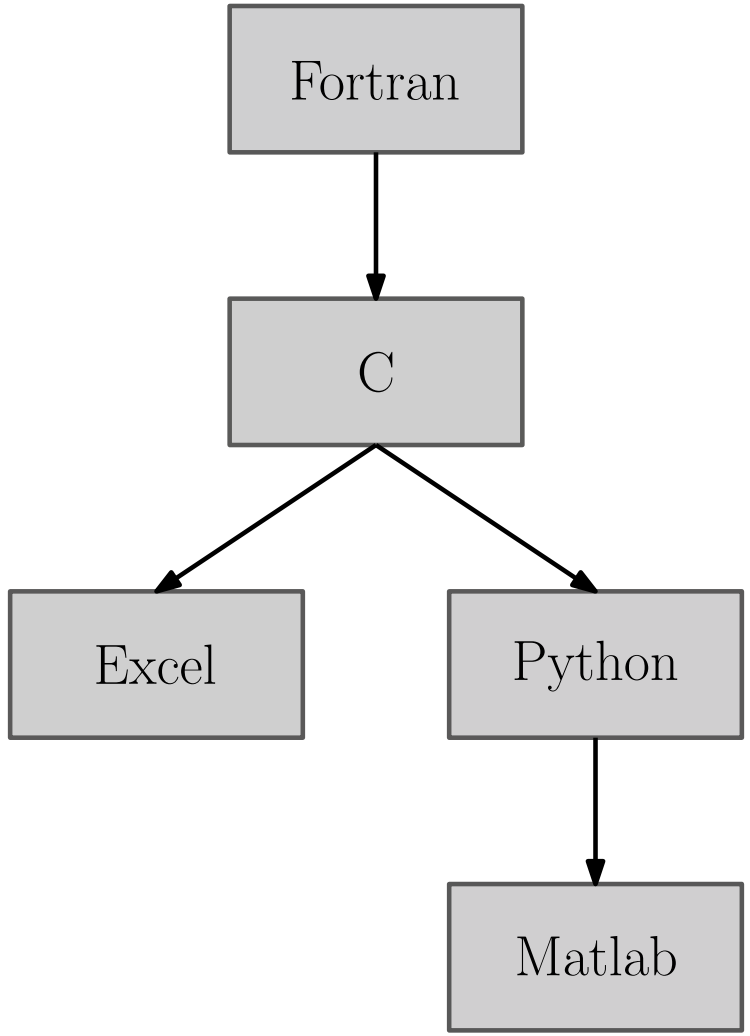
Subroutine interface:

- CEA2022 emphasizes the use of a subroutine interface, whereas CEA2 was restricted to text file input/output
- Allows CEA2022 to be called more easily by other programs, for example, integration within a CFD solver
- Serves as the base API to allow interfaces in other programming languages

Software Interface Approach

- Support calling CEA from other software via “subroutine interface” or “application programming interface” (API)
  - Want this support across a range of low-level (Fortran/C/C++) and interpreted (Python, MatLab) languages
- Strategy: “Hour Glass” design pattern, where standard C is used to define an API accessible in multiple languages.
  - Fortran 2003/2008 provides facilities for portable C interop
  - All target languages have ability to load/call C libraries

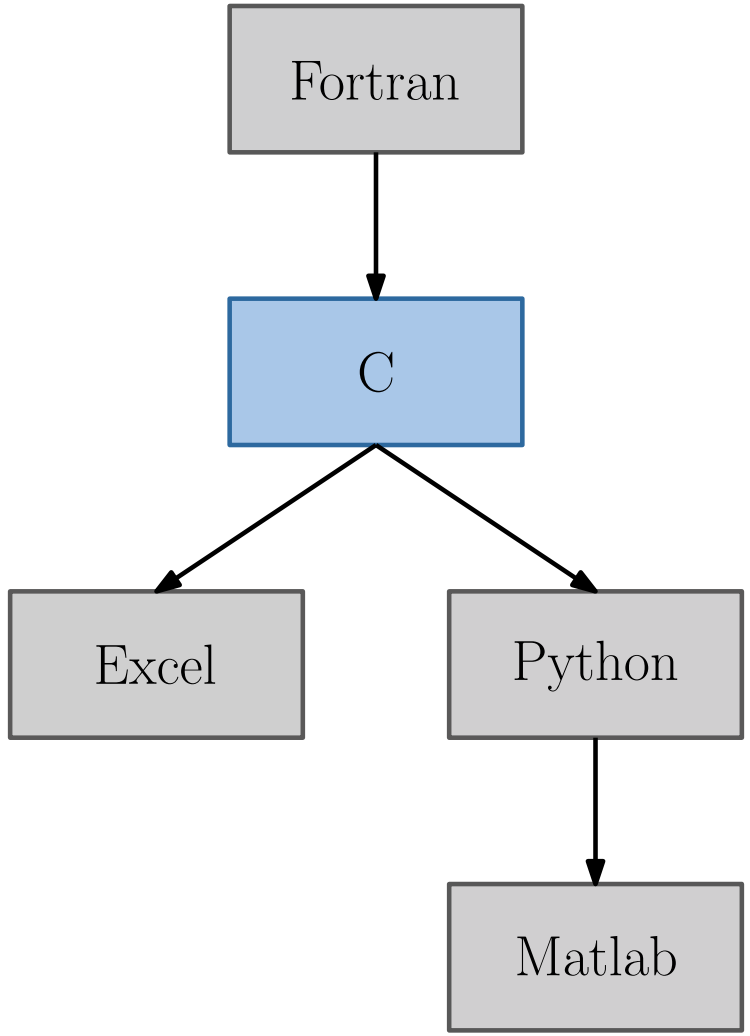
The legacy “.inp” is still an option!



C Interface

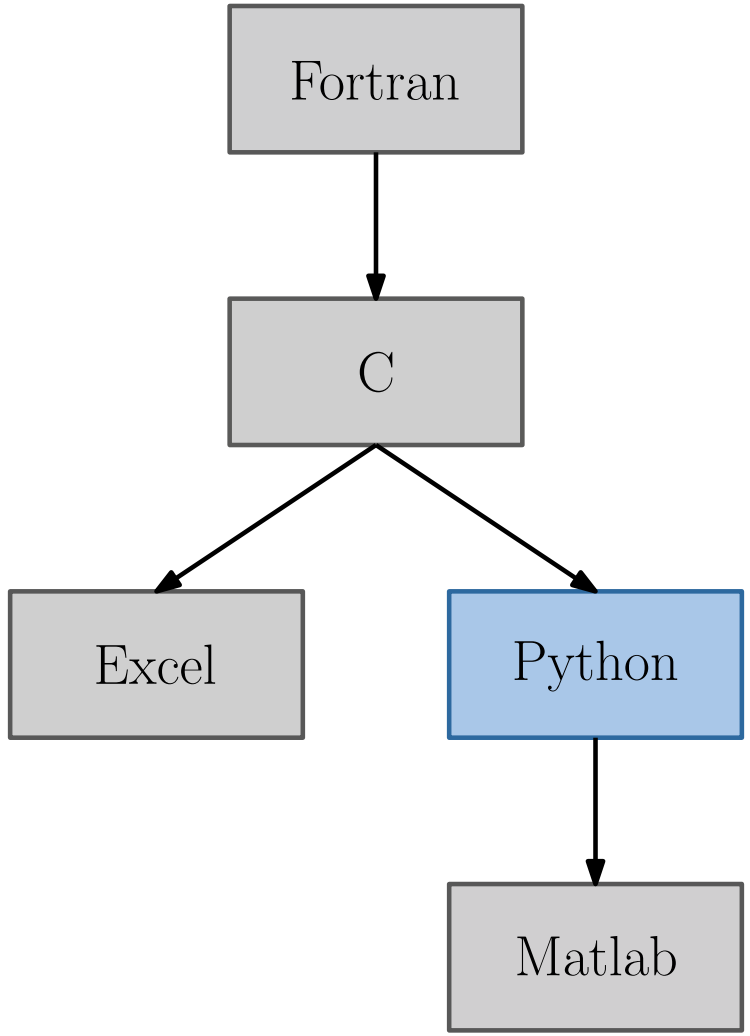
- Designed C-language API as foundation for multi-language support
  - Supports concurrent multi-solver, multi-solution use cases (CFD solvers)
  - Supports shared-memory parallel solves

- C API core uses Fortran 2008 interop features
  - Demonstrated CEA object allocation / use / deallocation from C
  - Demonstrated calling CEA library routines from C
  - Demonstrated passing character strings, arrays, etc. C ⇔ Fortran



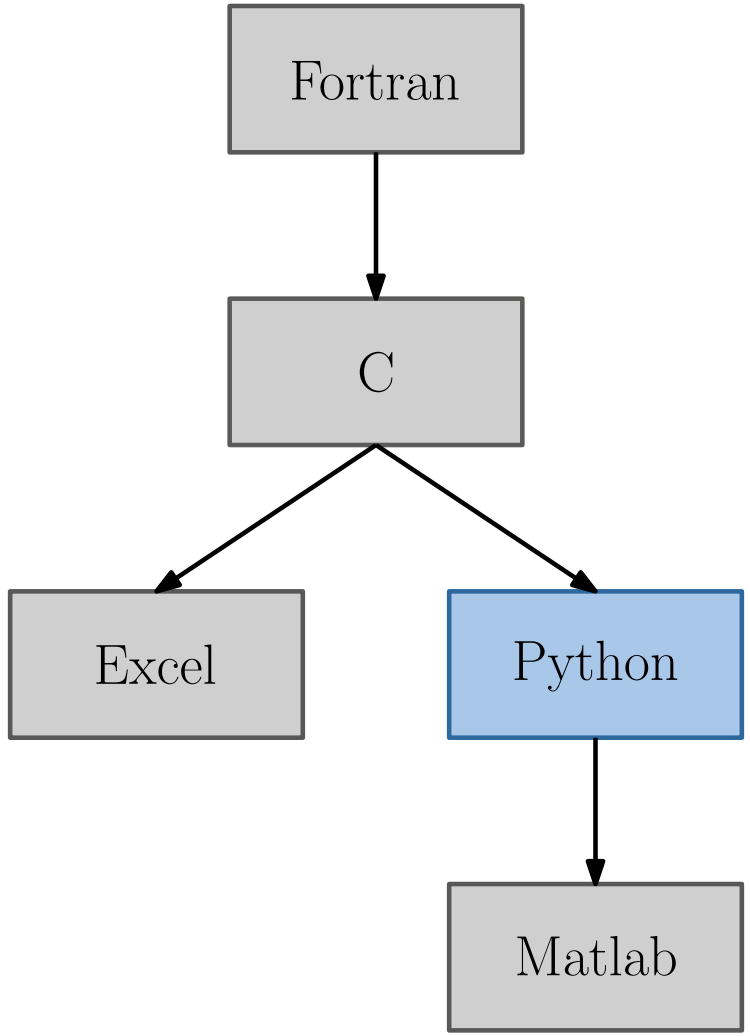
Python Interface

- Python wraps the C level interface using Cython
- Python interface builds using CMake



Python Interface

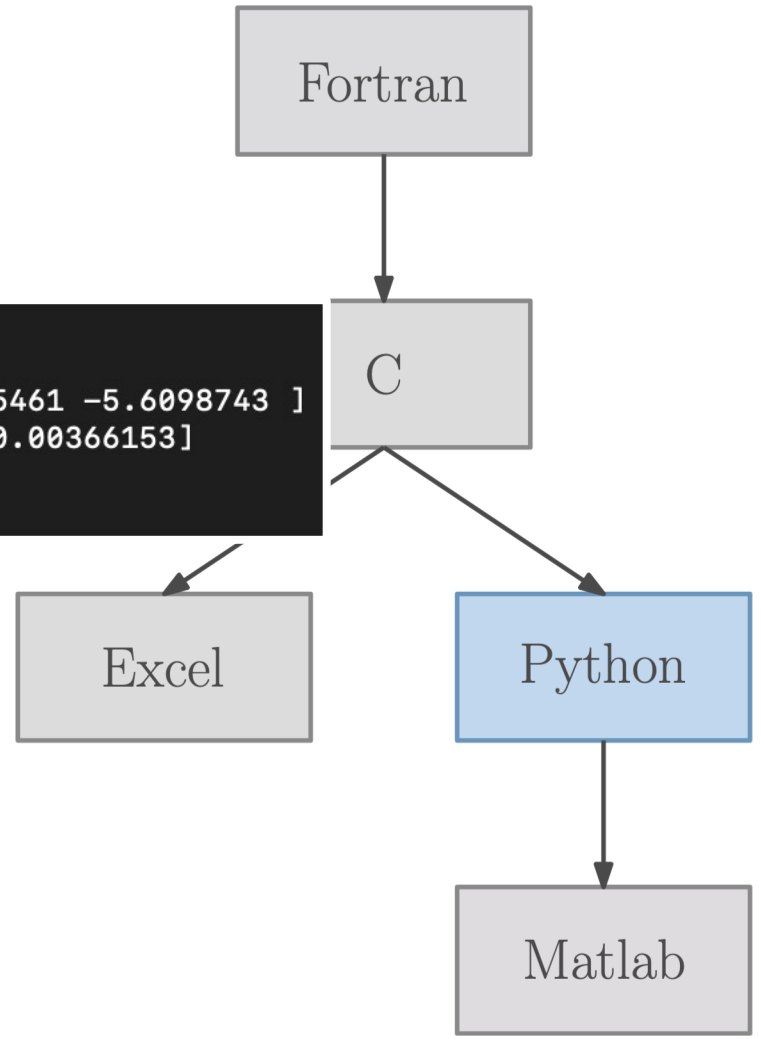
```python
import numpy as np
import _cea as cea

R = 8314.51

# Thermo states
p0 = 1.01325  # Fixed-pressure state (bar)
T0 = 2000.0   # Initial reactant temperature (K)

# Mixtures
reac = cea.Mixture([b"H2", b"O2"])
prod = cea.Mixture([b"H", b"H2", b"H2O", b"O", b"O2", b"OH"])

# Solver
solver = cea.Solver(prod, reac)

# Get the weights for one mole of each species
weights = reac.moles_to_weights(np.array([1.0, 1.0]))

# Get the fixed-enthalpy
h0 = reac.calc_property(cea.ENTHALPY, weights, T0)/R

# Equilibrium solve
solution = solver.solve(cea.HP, p0, h0, weights)

print("T = ", solution.T)
print("nj = ", solution.nj)
print("nj = ", solution.ln_nj)
print("n = ", solution.n)
print("converged?", solution.converged)
```



Python Interface

```python
import numpy as np
import _cea as cea

R = 8314.51

# Thermo states
p0 = 1.01325  # Fixed-pressure state (bar)
T0 = 2000.0   # Initial reactant temperature (K)

# Mixtures
reac = cea.Mixture([b"H2", b
prod = cea.Mixture([b"H", b

# Solver
solver = cea.Solver(prod, reac)

# Get the weights for one mole of each species
weights = reac.moles_to_weights(np.array([1.0, 1.0]))

# Get the fixed-enthalpy
h0 = reac.calc_property(cea.ENTHALPY, weights, T0)/R

# Equilibrium solve
solution = solver.solve(cea.HP, p0, h0, weights)

print("T = ", solution.T)
print("nj = ", solution.nj)
print("nj = ", solution.ln_nj)
print("n = ", solution.n)
print("converged?", solution.converged)
```

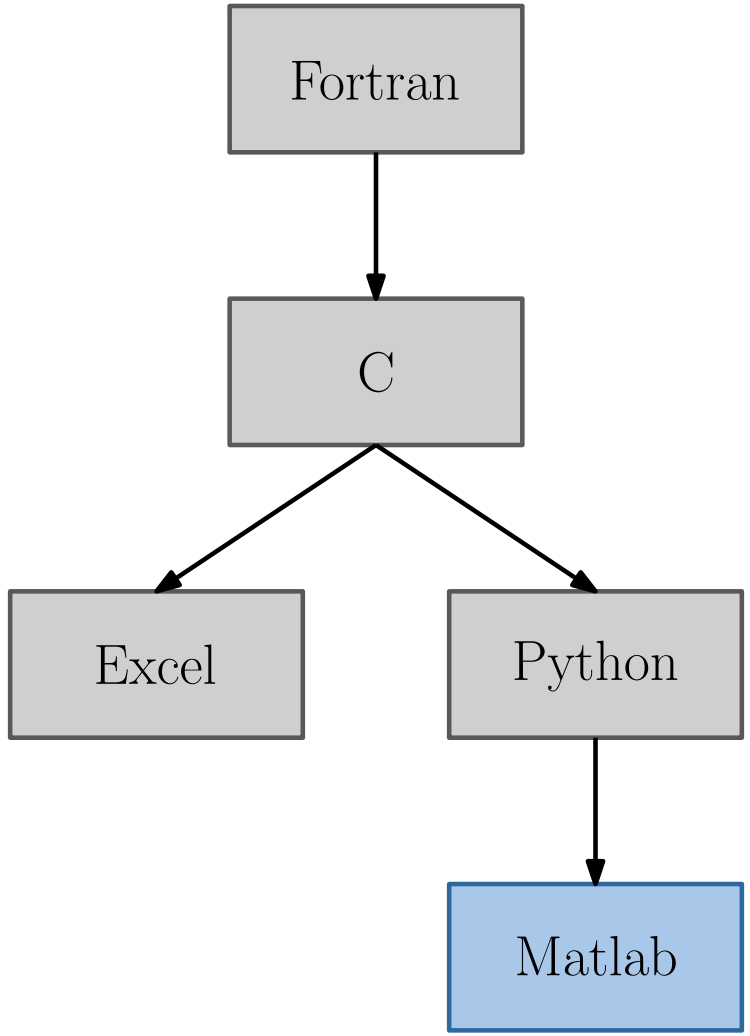
```
(cea) mleader@GRLAL0224020072 samples % python h2_o2.py
T =  3458.6924975345105
nj =  [-9.03916123 -7.76751404 -3.6087491  -7.49212988 -4.28695461 -5.6098743 ]
nj =  [0.00011867 0.00042326 0.02708571 0.00055745 0.01374673 0.00366153]
n =  0.045593351261682034
converged? True
```



Matlab Interface

- First: linking with “C” library, but the Matlab supported compilers and capability were more limited than I expected
- Solution: call the CEA Python library from Matlab instead
- Matlab calls Python directly → easier build, and only need to support/maintain one interface (Python)

- Matlab’s Python interface does not support creating object instances using instances of other objects
  - This breaks the expected CEA API, down to the Fortran level, so we create a separate CEA-Python wrapper just for Matlab
    - E.g. “import cea_matlab”, not “import cea”



Editor - /Users/mleader/git/cea/source/bind/matlab/test/example1.m

example1.m | h2_o2.m

```
1   clear; clc;
2
3   cea = py.importlib.import_module('_cea');
4
5   atm_to_bar = 1.01325;
6   % Species
7   reac_names = py.list({'H2', 'Air'});
8   prod_names = py.list({'Ar',    'C',    'CO',   'CO2',  'H', ...
9                         'H2',    'H2O',  'HNO',  'HO2',  'HNO2', ...
10                        'HNO3',  'N',    'NH',   'NO',   'N2', ...
11                        'N2O3',  'O',    'O2',   'OH',   'O3'});
12
13  % Thermo states
14  pressures = py.numpy.array(atm_to_bar*[1.0, 0.1, 0.01]);
15  temperatures = py.numpy.array([3000.0, 2000.0]);
16
17  % Mixture states
18  fuel_moles = py.numpy.array([1.0, 0.0]);
19  oxidant_moles = py.numpy.array([0.0, 1.0]);
20  chem_eq_ratios = py.numpy.array([1.0, 1.5]);
21
22  % Solve the equilibrium problem
23  soln = cea.eq_solve(cea.HP, 396.5543, 1.01325, reac_names, py.numpy.array([2.0159, 31.999]));
```

Workspace

Name	Value
atm_to_bar	1.0133
cea	*1x1 module*
chem_eq_ratios	*1x1 ndarray*
fuel_moles	*1x1 ndarray*
oxidant_moles	*1x1 ndarray*
pressures	*1x1 ndarray*
prod_names	*1x20 list*
reac_names	*1x2 list*
soln	*1x1 Solution*
temperatures	*1x1 ndarray*

Command Window

```
species =  ['H2', 'Air']
species =  ['H2', 'Air']
>>
```

Equilibrium Problem Validation Example

From RP-1311 Example 4 (UV problem)

- $\rho = 14.428$ kg/m3
- $u/R = -45.1343$ (kg-mol)(K)/kg
- o/f = 17
- Oxidant: Air @ 700 K
- Fuel: 60% C_7H_8(L), 40% C_8H_{19}(L) @ 298.15 K

	CEA2	CEA2022	Relative Error (%)
T	**2418.538**2574474097	**2418.538**4473019085	7.85×10^{-6}
$\ln(n_j)$: (HCOOH)2	**-52.3748**90133250204	**-52.3748**88695317857	2.75×10^{-8}
$\ln(n_j)$: Ar	**-8.09402656**40663025	**-8.09402656**28475903	1.51×10^{-6}
$\ln(n_j)$: C	**-42.9570**51147438705	**-42.9570**47510708030	8.47×10^{-6}
$\ln(n_j)$: CO	**-9.75915**61204275976	**-9.75915**54029655168	7.35×10^{-6}

Rocket Problem Validation Example

From RP-1311 Example 8 (IAC problem)

- Oxidant: O_2(L) @ 90.17 K
- Fuel: H_2(L) @ 20.27 K
- o/f = 5.55157

Station	Variable	CEA2	CEA2022	Relative Error (%)
Combustor	T	**3383.8446259451043**	**3383.8446259451043**	0.0
Combustor	$\ln(n_j)$: H_2	**-3.7643107135196541**	**-3.7643107135196558**	4.72×10^{-14}
Throat	T	**3185.6**731730807096	**3185.6**335035883212	1.25×10^{-3}
Throat	$\ln(n_j)$: H_2	**-3.77589**18668469459	**-3.77589**38050310880	5.13×10^{-5}
Exit (p_i/p_e)	T	**2567.34**01804441942	**2567.34**11236822476	3.67×10^{-5}
Exit (p_i/p_e)	$\ln(n_j)$: H_2	**-3.7880413**204534467	**-3.7880413**298752891	2.49×10^{-7}

Shock Problem Validation Example

From RP-1311 Example 7

- 0.05 moles H_2 and O_2 + 0.9 moles Ar @ 300 K
- P = 0.1 bar
- u_1 = 1,400 m/s

Type	Variable	CEA2	CEA2022	Relative Error (%)
Incident	T (K)	**2268**.1839840465000	**2268**.4047056277263	9.30×10^{-5}
Incident	P (bar)	**1.923**5852279853596	**1.923**7641552369096	9.73×10^{-5}
Incident	$\ln(n_j)$: HO_2	**-17.22**3079678243373	**-17.22**396748742256	3.97×10^{-5}
Incident	$\ln(n_j)$: OH	**-9.625**9080229365601	**-9.625**1244117341326	8.14×10^{-5}
Reflected	T (K)	**325**3.2881222920269	**325**4.3115063133464	3.15×10^{-4}
Reflected	P (bar)	**6.88**57588009600743	**6.88**63847357399406	9.09×10^{-5}
Reflected	$\ln(n_j)$: HO_2	**-15.14**9034932828400	**-15.14**8447374699135	3.88×10^{-5}
Reflected	$\ln(n_j)$: OH	**-7.68**11535944113283	**-7.68**01166285479265	1.35×10^{-4}

Detonation Problem Validation Example

From RP-1311 Example 6

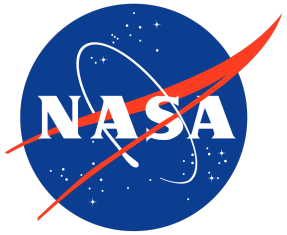
- Oxidant: O_2 @ 298.15 K
- Fuel: H_2 @ 298.15 K
- $r = 1$

	CEA2	CEA2022	Relative Error (%)
T (K)	**3674.2**8083110510 22	**3674.2**7786359820 45	8.08×10^{-7}
P (bar)	**18.7684**4669357106 5	**18.7684**5957960903 6	6.87×10^{-7}
$\ln(n_j)$: H_2	**-4.49361**23449492360	**-4.49361**24180425319	1.63×10^{-8}
$\ln(n_j)$: H_2O_2	**-13.4918960**07609208	**-13.4918960**98472500	6.73×10^{-9}
$\ln(n_j)$: OH	**-4.630083**6293723897	**-4.630083**9021590434	5.89×10^{-8}

Conclusion

- Modernization of CEA: replicate existing capability of CEA2, with added features and improved interface options
  - Added species to thermodynamic database for green propellants and sustainable aviation fuels
  - Supports thread-safe solves to allow runs in parallel
  - Added support for inert, and negative reactants
  - Results have been validated against CEA2 for each application type
  - Sustain CEA with ongoing improvement and fixes based on user-feedback
- Plan is to release open-source soon, targeting by the end of the calendar year

Future Work



Immediate priorities:

- Finish Excel interface
- Finish analytic derivatives
- Improve testing
  - Integration tests
  - Add tests for each interface

Near term:

- Add useful outputs for non-equilibrium sanity checks
  - Example: % volume occupied by condensed species
- Integration with other tools, e.g. NPSS
- Algorithm improvements
  - Improved initial guess

Acknowledgements

- Thanks to NESC for funding this work
- Thanks to HTP, RVLT, and TTT programs for their support

mark.leader@nasa.gov

Thermodynamic Properties

$$\frac{C_p}{R} = a_1 T^{-2} + a_2 T^{-1} + a_3 + a_4 T + a_5 T^2 + a_6 T^3 + a_7 T^4$$

$$\frac{H^\circ}{RT} = \frac{\int C_p^\circ dT}{RT}$$

$$\frac{H^\circ}{RT} = -a_1 T^{-2} + a_2 T^{-1} \ln T + a_3 + a_4 \frac{T}{2} + a_5 \frac{T^2}{3} + a_6 \frac{T^3}{4} + a_7 \frac{T^4}{5} + \frac{a_8}{T}$$

$$\frac{S^\circ}{R} = \int \frac{C_p^\circ}{RT} dT$$

$$\frac{S^\circ}{R} = -a_1 \frac{T^{-2}}{2} - a_2 T^{-1} + a_3 \ln T + a_4 T + a_5 \frac{T^2}{2} + a_6 \frac{T^3}{3} + a_7 \frac{T^4}{4} + a_9$$