Using Modern Fortran OOP for the development of an extensible library for thermodynamic calculations

Federico E. Benelli, Salvador E. Brandolín, Martín Cismondi-Duarte

International Fortran Conference 2025

CONICET
UNC Universidad Nacional de Córdoba
I P Q A

Outline

- Introduction (Who we are and what we do)
- Starting point of our project
- Using OOP to make extensibility easier
- Providing C and Python interfaces

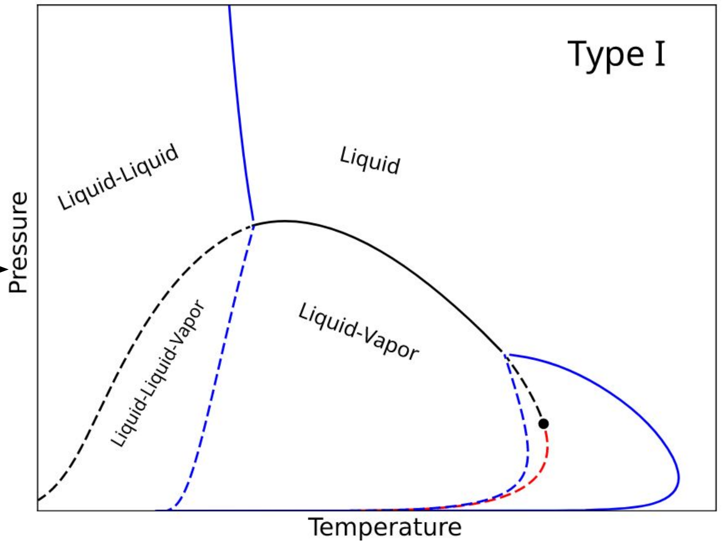
Who are we?

Fluids **thermodynamics** group

- Modelling of reservoir fluids
- Phase equilibria algorithms
- Fitting of thermodynamic models

Our focus is on phase equilibria

$$F(\vec{X}, S) = \begin{bmatrix} \ln K_1^x + \ln \hat{\phi}_1^x(\mathbf{x}, P, T) - \ln \hat{\phi}_1^w(\mathbf{w}, P, T) \\ \vdots \\ \ln K_i^x + \ln \hat{\phi}_i^x(\mathbf{x}, P, T) - \ln \hat{\phi}_i^w(\mathbf{w}, P, T) \\ \vdots \\ \ln K_N^x + \ln \hat{\phi}_N^x(\mathbf{x}, P, T) - \ln \hat{\phi}_N^w(\mathbf{w}, P, T) \\ \ln K_1^y + \ln \hat{\phi}_1^y(\mathbf{y}, P, T) - \ln \hat{\phi}_1^w(\mathbf{w}, P, T) \\ \vdots \\ \ln K_i^y + \ln \hat{\phi}_i^y(\mathbf{y}, P, T) - \ln \hat{\phi}_i^w(\mathbf{w}, P, T) \\ \vdots \\ \ln K_N^y + \ln \hat{\phi}_N^y(\mathbf{y}, P, T) - \ln \hat{\phi}_N^w(\mathbf{w}, P, T) \\ \sum_i^N (\mathbf{w}_i - 1) \\ \sum_i^N (\mathbf{x}_i - \mathbf{y}_i) \\ X_S - S \end{bmatrix} = \vec{0} \xrightarrow{\text{Specific Algorithms}}$$



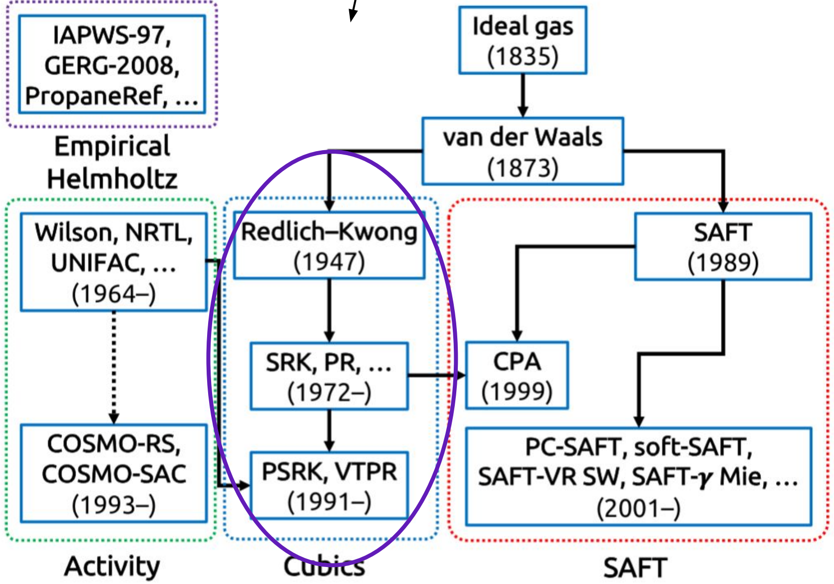
Equations of State (EoS)

$$\frac{A^r}{RT} = F^{EOS}\big(\overbrace{a,b,\ldots k,l,\ldots}^{\text{Parameters}}\overbrace{n,T,V}^{\text{Variables}}\big)$$

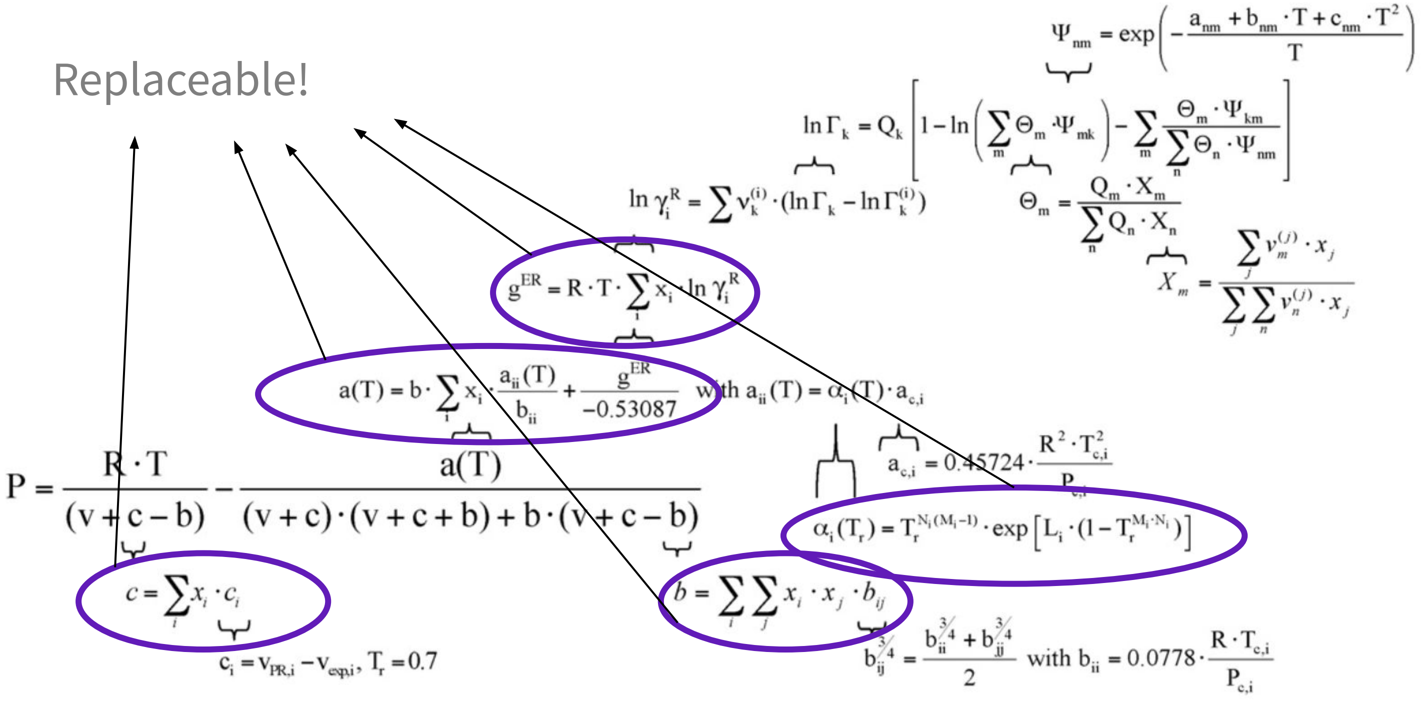
$$\ln \hat{\phi}_i = \left(\frac{\partial F}{\partial n_i}\right)_{T,V,n_j} - \ln Z$$

$$P = -RT\left(\frac{\partial F}{\partial V}\right)_{T,n} + \frac{nRT}{V}$$

There are multiple "families" of EoS (and other thermodynamic models)



Flexibility for implementations is needed



$$P = \frac{R \cdot T}{(v+c-b)} - \frac{a(T)}{(v+c)\cdot(v+c+b)+b\cdot(v+c-b)}$$

$$c = \sum_i x_i \cdot c_i \qquad c_i = v_{PR,i} - v_{exp,i},\ T_r = 0.7$$

$$b = \sum_i \sum_j x_i \cdot x_j \cdot b_{ij} \qquad b_{ij}^{3/4} = \frac{b_{ii}^{3/4} + b_{jj}^{3/4}}{2} \text{ with } b_{ii} = 0.0778 \cdot \frac{R \cdot T_{c,i}}{P_{c,i}}$$

$$a(T) = b \cdot \sum_i x_i \cdot \frac{a_{ii}(T)}{b_{ii}} + \frac{g^{ER}}{-0.53087} \text{ with } a_{ii}(T) = \alpha_i(T) \cdot a_{c,i}$$

$$a_{c,i} = 0.45724 \cdot \frac{R^2 \cdot T_{c,i}^2}{P_{c,i}}$$

$$\alpha_i(T_r) = T_r^{N_i(M_i-1)} \cdot \exp\left[L_i \cdot (1 - T_r^{M_i \cdot N_i})\right]$$

$$g^{ER} = R \cdot T \cdot \sum_i x_i \cdot \ln \gamma_i^R$$

$$\ln \gamma_i^R = \sum \nu_k^{(i)} \cdot (\ln \Gamma_k - \ln \Gamma_k^{(i)})$$

$$\ln \Gamma_k = Q_k \left[1 - \ln\left(\sum_m \Theta_m \cdot \Psi_{mk}\right) - \sum_m \frac{\Theta_m \cdot \Psi_{km}}{\sum_n \Theta_n \cdot \Psi_{nm}}\right]$$

$$\Psi_{nm} = \exp\left(-\frac{a_{nm} + b_{nm} \cdot T + c_{nm} \cdot T^2}{T}\right)$$

$$\Theta_m = \frac{Q_m \cdot X_m}{\sum_n Q_n \cdot X_n}$$

$$X_m = \frac{\sum_j \nu_m^{(j)} \cdot x_j}{\sum_j \sum_n \nu_n^{(j)} \cdot x_j}$$

Figure 2: Diagram of the VTPR equations. From source Schmid and Gmehling (2012)

A brief history of our codes

Michelsen-Mollerup
DTU (Denmark)
1980s

Cismondi
2000s-Present

Benelli-Brandolín
IPQA (Argentina)
2021-Present

Many other “branches”

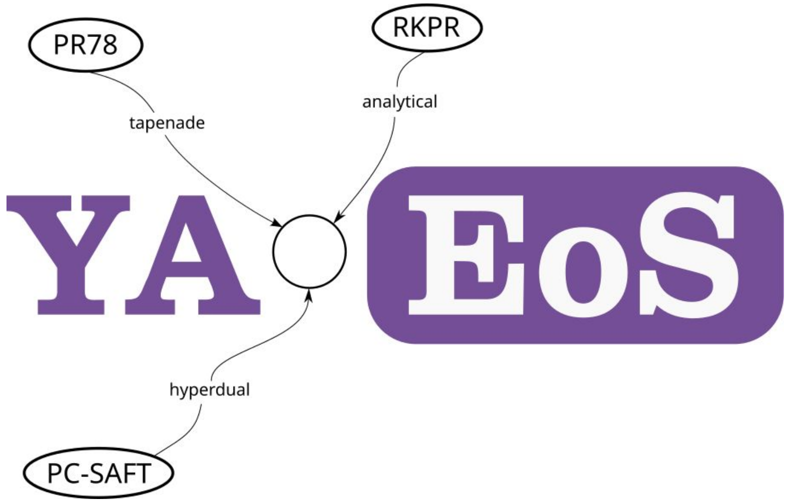
Sánchez
PLAPIQUI (Argentina)
2010s-Present

Rodriguez Reartes
URV (Spain) - PLAPIQUI
2010s-Present

We did yaeos

- Modernization of legacy codes
- Centralization of our different algorithms, avoiding multiple programs and different incompatible versions
- *Using a basic OOP paradigm to make extensibility and usage easier.*
- Provide an easy to use Fortran-API and an even easier Python-API

github.com/ipqa-research/yaeos



An example of our starting point

```
subroutine aTder(ac,Tc,rk,T,a,dadT,dadT2)
  implicit DOUBLE PRECISION (A-H,O-Z)
  COMMON /MODEL/ NMODEL
  Tr=T/Tc
  IF(NMODEL.LE.2)THEN
    rm=rk
    a=ac*(1+rm*(1-sqrt(Tr)))**2
    dadT=ac*rm*(rm-(rm+1)/sqrt(Tr))/Tc
    dadT2=ac*rm*(rm+1)/(2*Tc**2*Tr**1.5D0)
  ELSE
    a=ac*(3/(2+Tr))**rk
    dadT=-rk*a/Tc/(2+Tr)
    dadT2=-(rk+1)*dadT/Tc/(2+Tr)
  END IF
end
```

$a(T)$

Usage of COMMONs and IF statements to use the desired model

This check must be done on multiple occasions. On different routines.

For each new implementation new checks must be added.

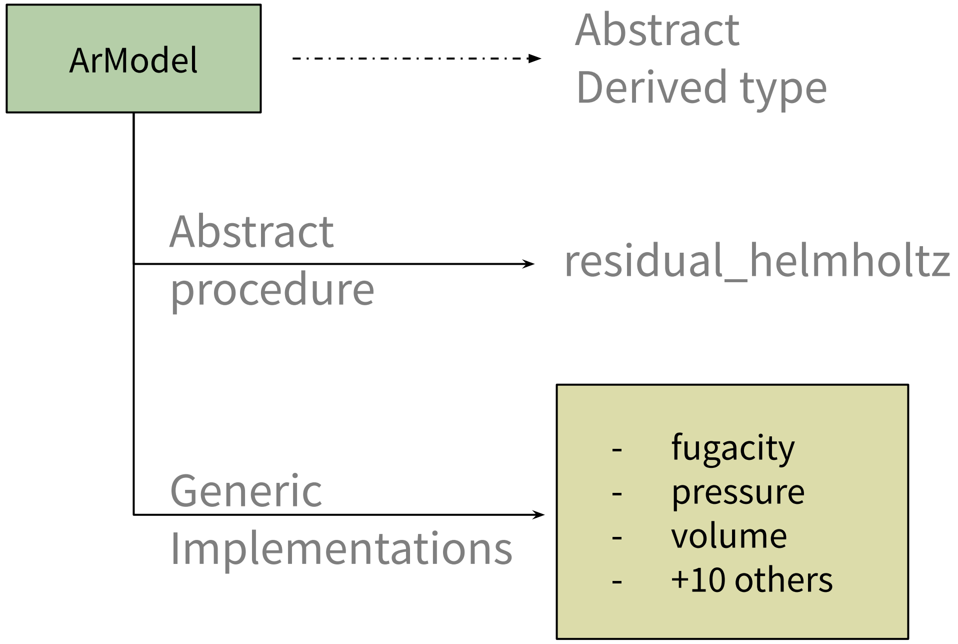
Using OOP to ease extensibility

Using OOP to make the extensibility easier

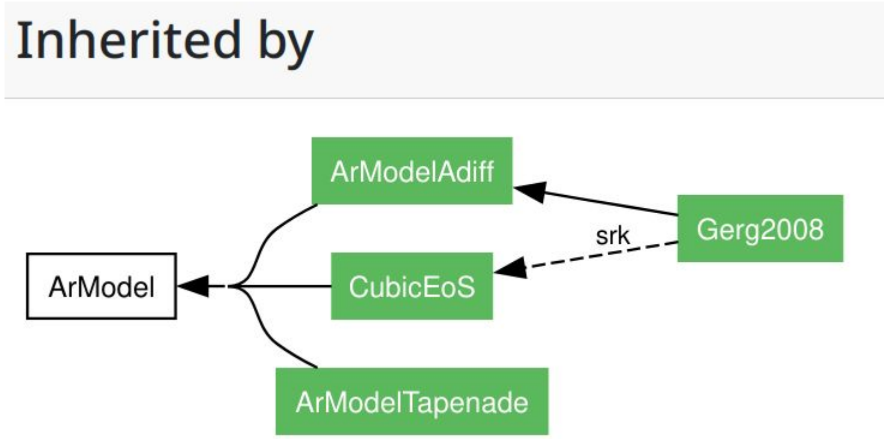
$$\frac{A^r}{RT} = F^{EOS}\left(\overbrace{a,b,\ldots k,l,\ldots}^{\text{Parameters}}\overbrace{n,T,V}^{\text{Variables}}\right)$$

$$\boxed{\ln\hat{\phi}_i = \left(\frac{\partial F}{\partial n_i}\right)_{T,V,n_j} - \ln Z}$$

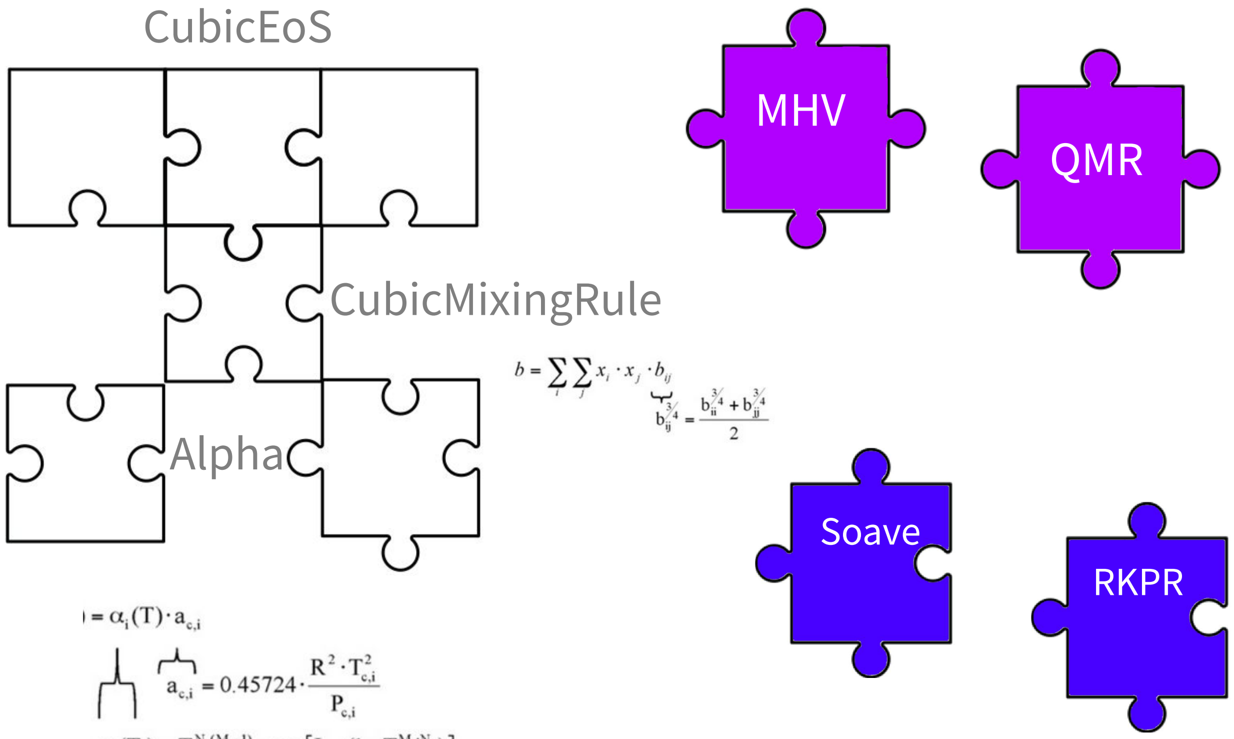
$$\boxed{P = -RT\left(\frac{\partial F}{\partial V}\right)_{T,n} + \frac{nRT}{V}}$$



With extension now we can implement more models



In the specific case of CubicEoS: Strategy design pattern



$$b = \sum_i \sum_j x_i \cdot x_j \cdot b_{ij}$$

$$b_{ij}^{3/4} = \frac{b_{ii}^{3/4} + b_{jj}^{3/4}}{2}$$

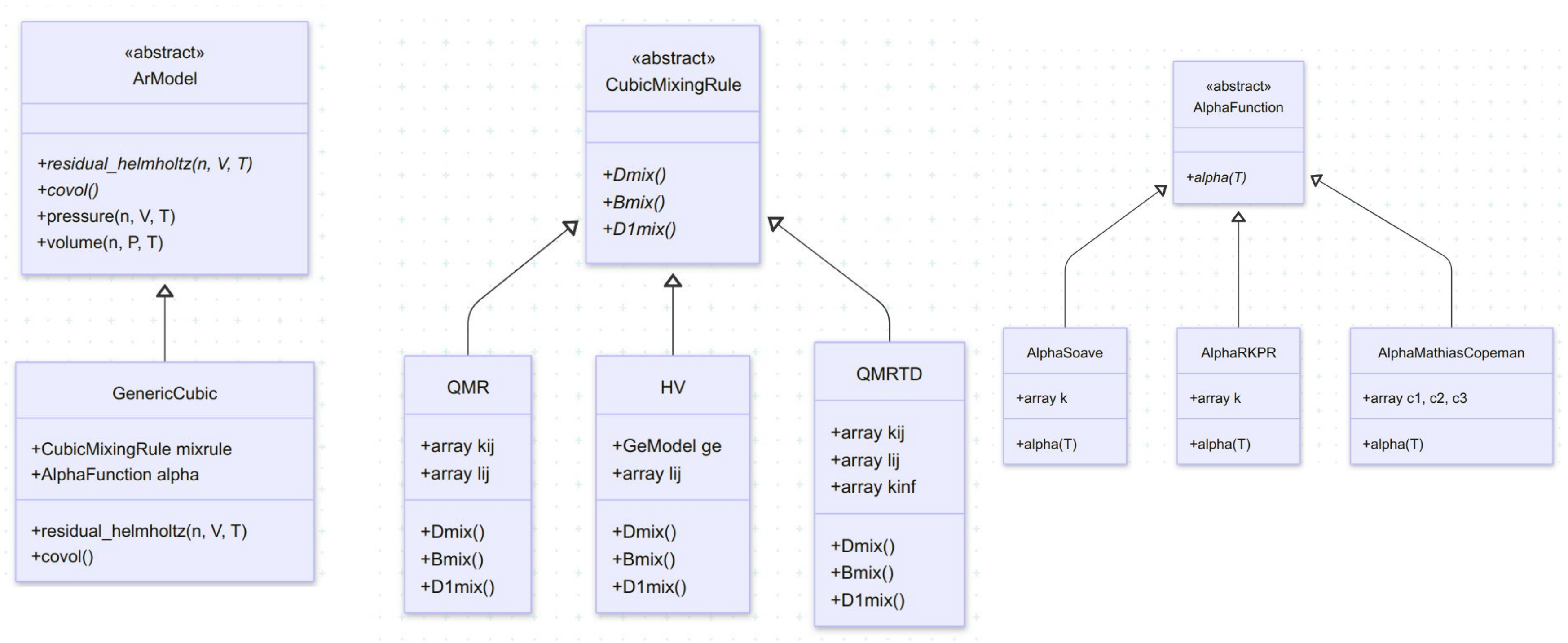
$$) = \alpha_i(T) \cdot a_{c,i}$$

$$a_{c,i} = 0.45724 \cdot \frac{R^2 \cdot T_{c,i}^2}{P_{c,i}}$$

$$\alpha_i(T_r) = T_r^{N_i(M_i-1)} \cdot \exp\left[L_i \cdot (1 - T_r^{M_i \cdot N_i})\right]$$

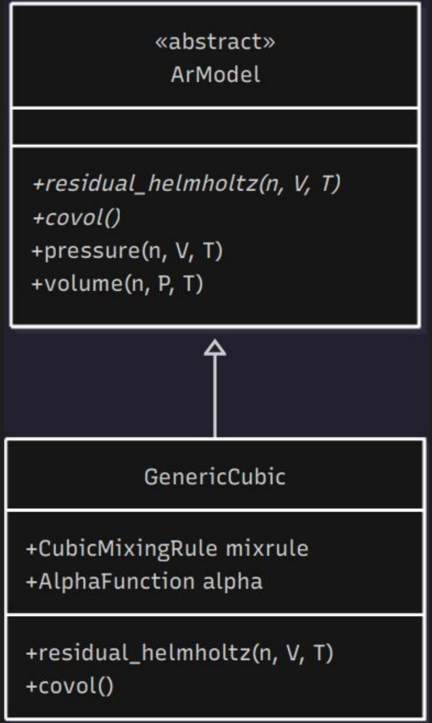
Different possible behaviours “strategies”

We can define a “Generic” CubicEoS class/type



Definition of a Cubic model with OOP

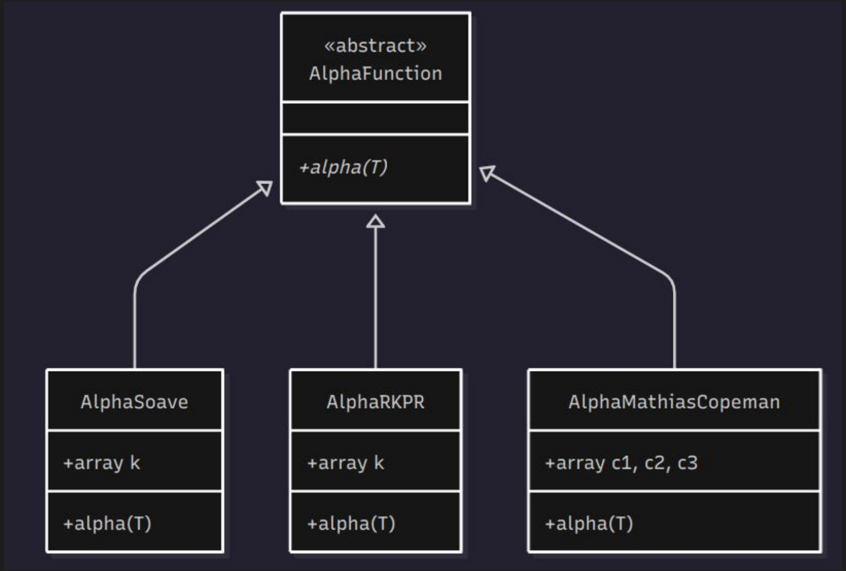
```fortran
type, extends(ArModel) :: CubicEoS
     !! # Cubic Equation of State.
     class(CubicMixRule), allocatable :: mixrule
     class(AlphaFunction), allocatable :: alpha
     real(pr), allocatable :: ac(:) !! Attractive critical parameter
     real(pr), allocatable :: b(:) !! Repulsive parameter
     real(pr), allocatable :: del1(:) !! \(\delta_1\) parameter
     real(pr), allocatable :: del2(:) !! \(\delta_2\) parameter
  contains
     procedure :: residual_helmholtz => GenericCubic_Ar
     procedure :: get_v0 => v0
     procedure :: volume => volume
     procedure :: set_delta1 => set_delta1
     procedure :: set_mixrule => set_mixrule
  end type CubicEoS
```



```fortran
type, extends(AlphaFunction) :: AlphaSoave
   !! Soave \(\alpha\) function.
   !! \( \alpha(T_r) = (1 + k (1 - \sqrt{Tr}))^2 \)
   real(pr), allocatable :: k(:) !! \(k\) parameter.
contains
   procedure :: alpha !! Alpha function
end type AlphaSoave

type, extends(AlphaFunction) :: AlphaRKPR
   !! RKPR \(\alpha\) function
   !! \[
   !! \alpha(T_r) = \left(\frac{3}{2 + T_r}\right)^k
   !! \]
   real(pr), allocatable :: k(:) !! \(k\) parameter.
contains
   procedure :: alpha => alpha_rkpr
end type AlphaRKPR

type, extends(AlphaFunction) :: AlphaMathiasCopeman
   !! Mathias Copeman \(\alpha\) function.
   real(pr), allocatable :: c1(:)
   real(pr), allocatable :: c2(:)
   real(pr), allocatable :: c3(:)
contains
   procedure :: alpha => alpha_mc
end type AlphaMathiasCopeman
```



Now the implementation does not need conditionals

```fortran
subroutine GenericCubic_Ar(self, n, V, T, Ar, ArV, ArT, ArTV, ArV2, ArT2, Arn, ArVn, ArTn, Arn2)
…
! ========================================================================
! Attractive parameter and derivatives
! ------------------------------------------------------------------------
call self%alpha%alpha(Tr, a, dadt, dadt2)
a = self%ac * a
dadt = self%ac * dadt / self%components%Tc
dadt2 = self%ac * dadt2 / self%components%Tc**2

! ========================================================================
! Mixing rules
! ------------------------------------------------------------------------
call self%mixrule%D1mix(n, self%del1, D1, dD1i, dD1ij)
call self%mixrule%Bmix(n, self%b, B, dBi, dBij)
call self%mixrule%Dmix(n, T, a, dadt, dadt2, D, dDdT, dDdT2, dDi, dDidT, dDij)
```

We can define functions for each specific model

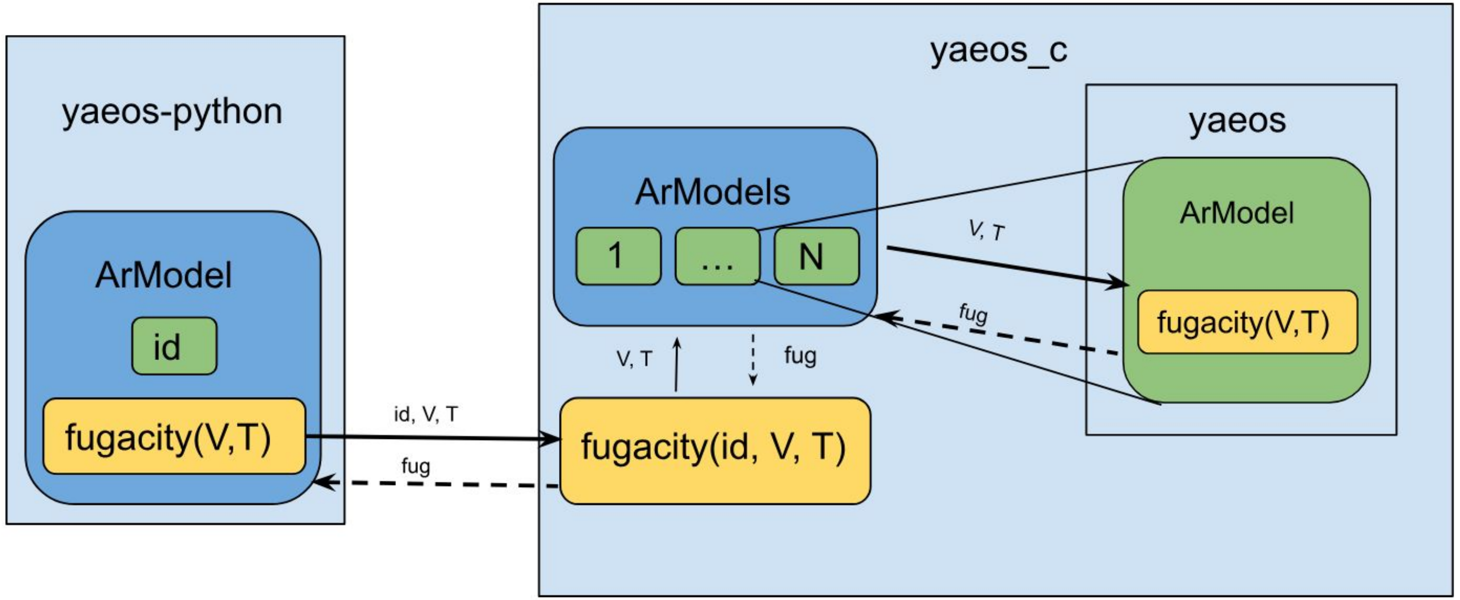
```fortran
type(CubicEoS) function PengRobinson78(tc, pc, w, kij, lij) result(model)
   type(QMR) :: mixrule
   type(AlphaSoave) :: alpha
   allocate(alpha%k(nc))
   where (w <=0.491)
      alpha%k = 0.37464 + 1.54226 * w - 0.26992 *w**2
   elsewhere
      alpha%k = 0.379642 + 1.48503 * w - 0.164423 *w**2 + 0.016666 *w**3
   end where

   mixrule%k = kij

   model%del1 = [(1 + sqrt(2.0_pr), i=1,nc)]
   model%del2 = [(1 - sqrt(2.0_pr), i=1,nc)]
   model%alpha = alpha
   model%mixrule = mixrule
end function PengRobinson78
```

Providing a C and Python interface

Our main goal: Calling from Python



Using containers and ids (Python-C API)

```fortran
type :: ArModelContainer
   !! Container type for ArModels
   class(ArModel), allocatable :: model
end type ArModelContainer

class(ArModel), allocatable :: ar_model
  !! Singleton to hold temporal ArModels

! Containers of models
logical :: free_ar_model(max_models) = .true.
  !! List to store the availability of ArModel ids

class(ArModelContainer), allocatable :: ar_models(:)
  !! List of allocated ArModels
```

We define a container that will hold some model

A singleton model to make intermediate assignations

This allows us to save a list of models in use

For each new model we extend the list

```fortran
subroutine extend_ar_models_list(id)
     !! Find the first available model container and allocate the model
     !! there. Then return the found id.
     integer(c_int), intent(out) :: id !! Saved model id
     integer :: i
     if (.not. allocated(ar_models)) allocate(ar_models(max_models))

     ! Find the first not allocated model
     do i=1,max_models
        if (free_ar_model(i)) then
           free_ar_model(i) = .false.
           id = i
           call move_alloc(ar_model, ar_models(i)%model)
           exit
        end if
     end do
     if (id == max_models) error stop 1
  end subroutine extend_ar_models_list
```

We iterate through the list of models until we find a free space and allocate the singleton there

Calling the real method

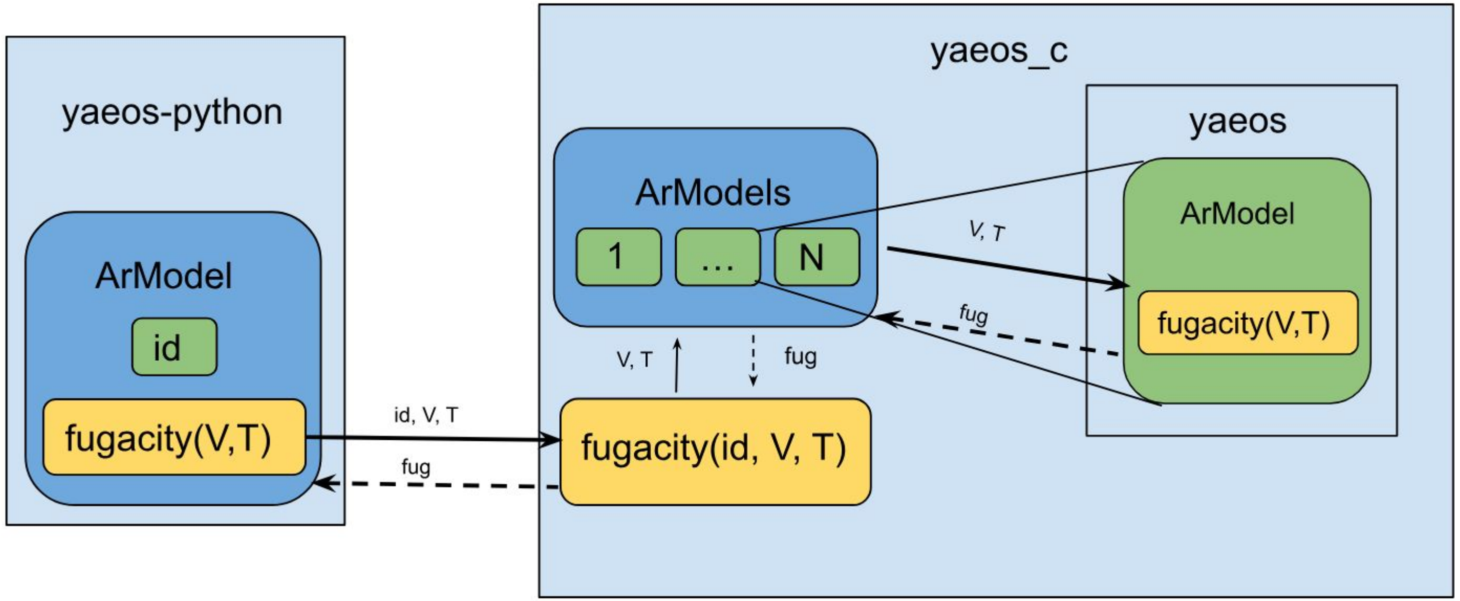
```
subroutine pressure(id, n, V, T, P, dPdV, dPdT, dPdn)
   integer(c_int), intent(in) :: id
   real(c_double), intent(in) :: n(:), V, T
   real(c_double), intent(out) :: P
   real(c_double), intent(in out) :: dPdV, dPdT, dPdn(size(n))

   call ar_models(id)%model%pressure(&
      n, V, T, P, dPdV, dPdT, dPdn &
      )
end subroutine pressure
```

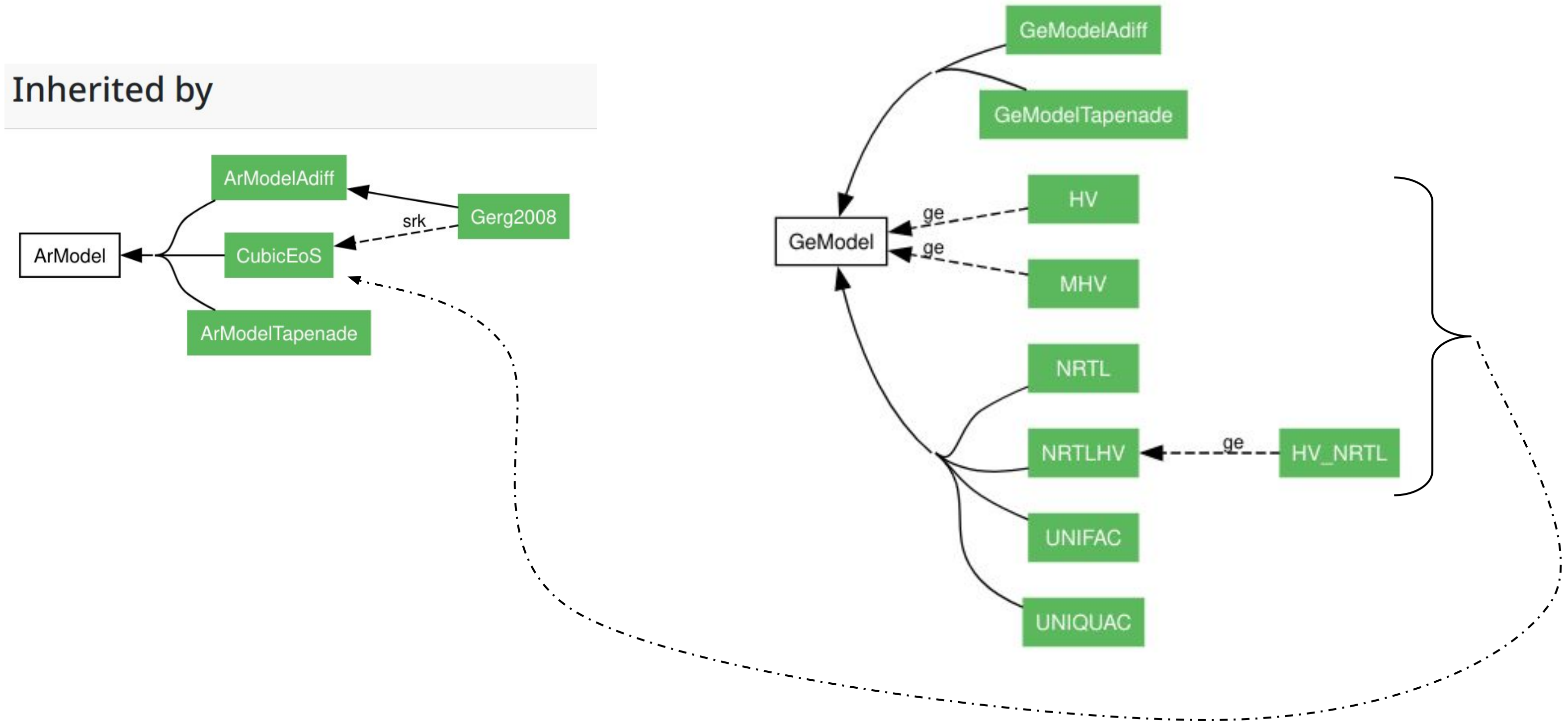
Using the provided **id** we look for the model instance in the array

Calling from Python

The main project can be compiled with fpm, and then the C interface with f2py.



We've shown only a small section

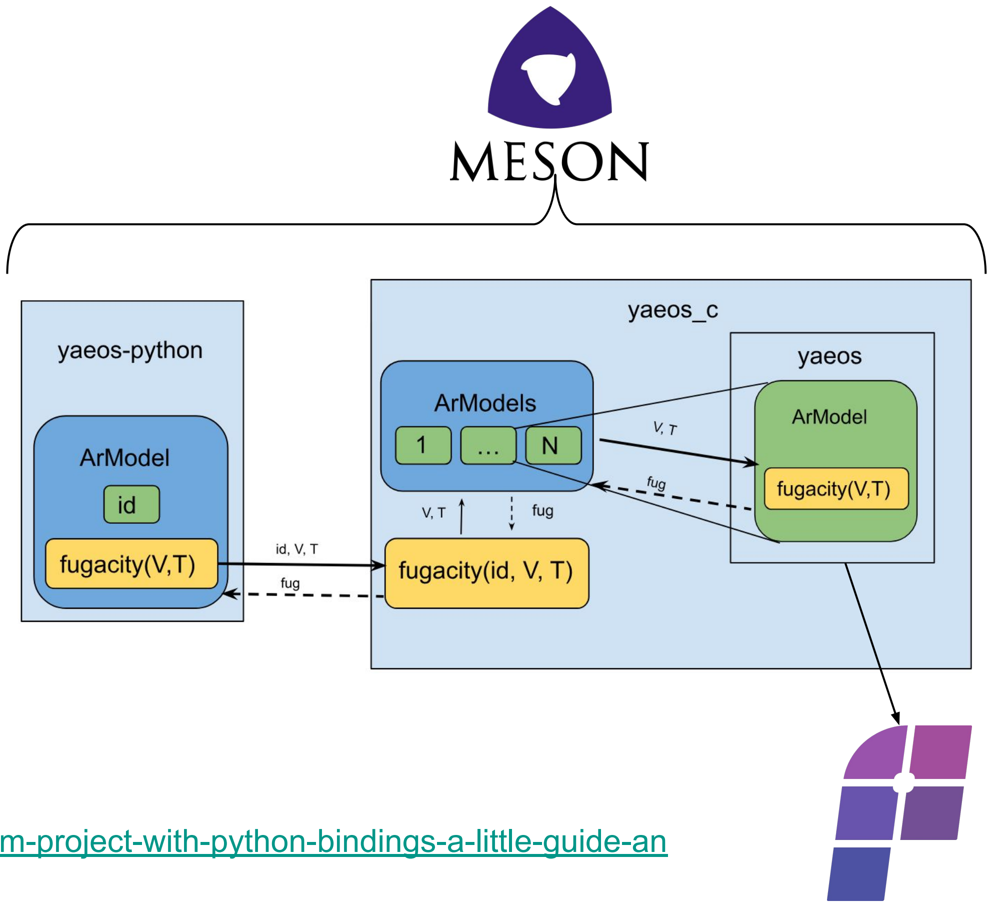


Build system

We use meson to provide a distributable Python package.

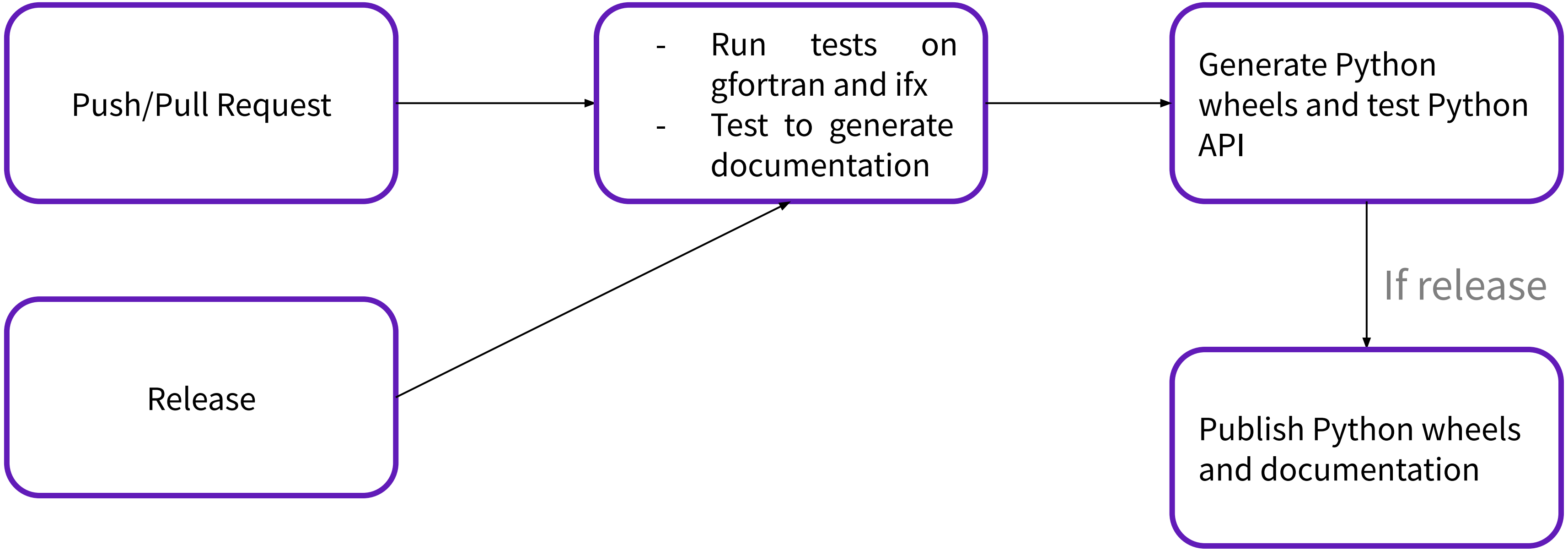
Meson:

- Runs **fpm** to generate a static library.
- Runs f2py to generate a C-Python extension.
- Compiles the extension and links it to the static library.



More info:
https://fortran-lang.discourse.group/t/packaging-a-fpm-project-with-python-bindings-a-little-guide-and-insights-from-our-experience/8495/6

We automate our tests, documentation and releases with GitHub Actions



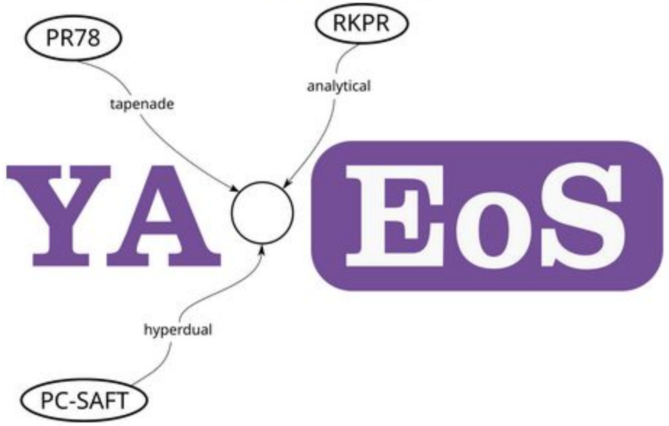
Push/Pull Request
- Run tests on gfortran and ifx
- Test to generate documentation
Generate Python wheels and test Python API
Release
If release
Publish Python wheels and documentation

The documentation is generated with a mix of FORD for Fortran and Sphinx for the Python API, it can be accessed here: https://ipqa-research.github.io/yaeos/

Thank you for your attention! Any Questions?

Fortran | fpm Fortran package manager | ford Documentation | License MPL 2.0 | CI passing | codecov 95%

Open in Colab



GitHub: https://github.com/ipqa-research/yaeos
Contact: federico.benelli@unc.edu.ar

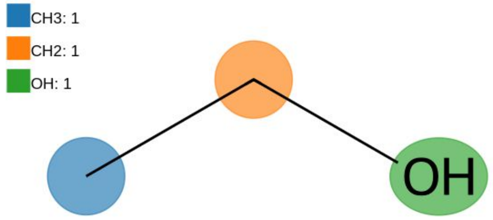
International Fortran Conference 2025

Extra

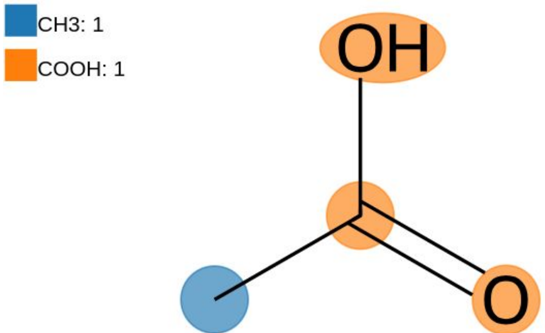
OOP to better understand what is happening

Using UNIFAC as an example.

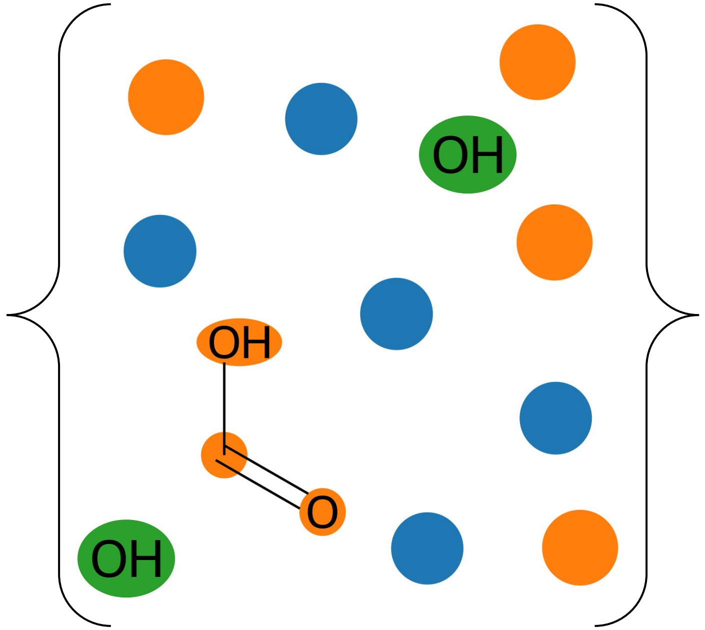
Ethanol



Acetic acid

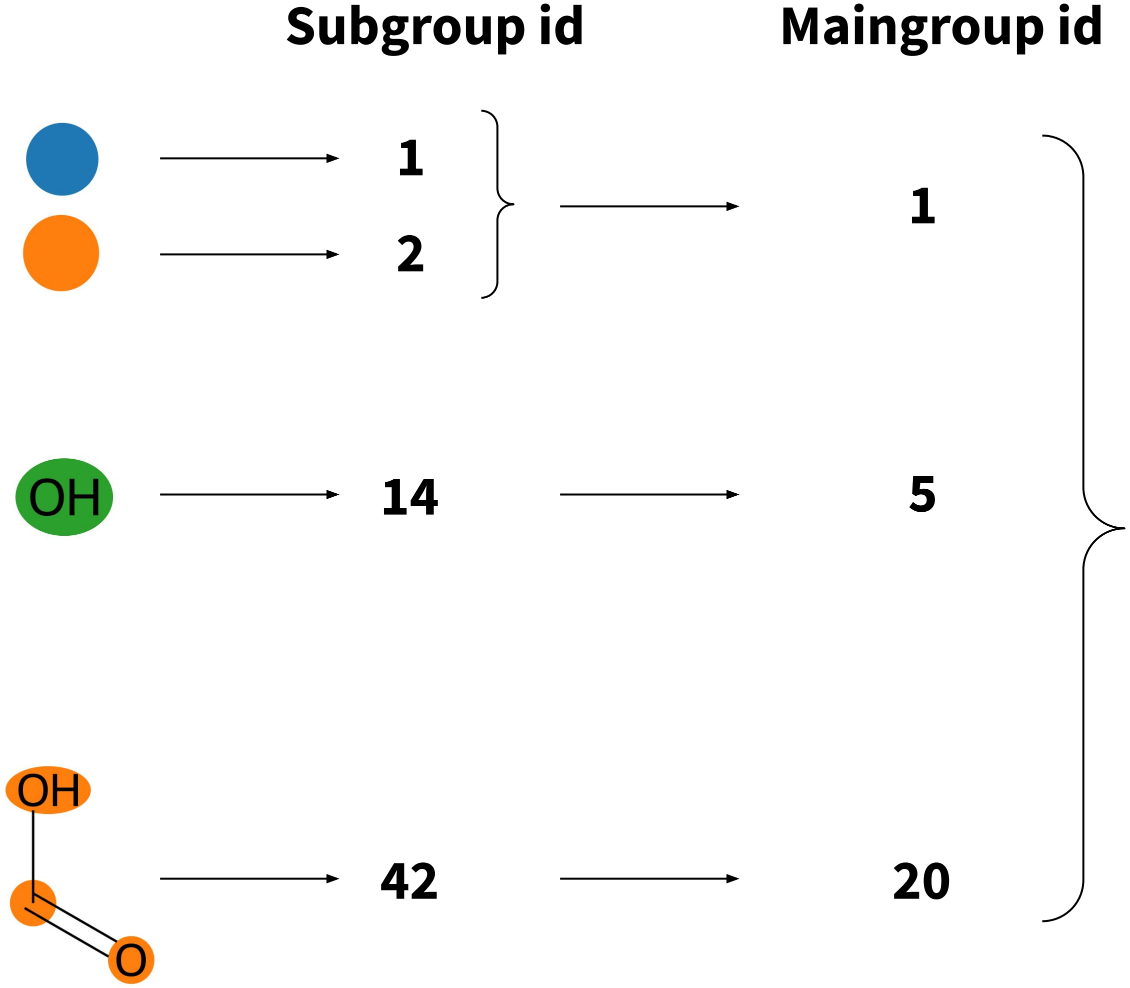


Groups stew



Properties of the mixture:

ethanol - acetic acid



Interaction between two Subgroups Is calculated From the interaction between their Maingroups

```fortran
type :: GeGCModelParameters
   integer, allocatable :: subgroups_ids(:)
   !! ID of each model's subgroup
   integer, allocatable :: maingroups_ids(:)
   !! ID of each model's maingroup
   integer, allocatable :: subgroups_maingroups(:)
   !! Maingroup of each subgroup
   real(pr), allocatable :: subgroups_Rs(:)
   !! \(R \) value of each subgroup
   real(pr), allocatable :: subgroups_Qs(:)
   !! \(Q \) value of each subgroup
   real(pr), allocatable :: maingroups_aij(:,:)
   !! Maingroup \(a_{ij} \) interaction parameters matrix
   real(pr), allocatable :: maingroups_bij(:,:)
   !! Maingroup \(b_{ij} \) interaction parameters matrix
   real(pr), allocatable :: maingroups_cij(:,:)
   !! Maingroup \(c_{ij} \) interaction parameters matrix
contains
   procedure :: get_subgroup_index => get_subgroup_index
   procedure :: get_maingroup_index => get_maingroup_index
   procedure :: get_subgroup_maingroup => get_subgroup_maingroup
   procedure :: get_subgroup_R => get_subgroup_R
   procedure :: get_subgroup_Q => get_subgroup_Q
   procedure :: get_maingroups_aij => get_maingroups_aij
   procedure :: get_maingroups_bij => get_maingroups_bij
   procedure :: get_maingroups_cij => get_maingroups_cij
   procedure :: get_subgroups_aij => get_subgroups_aij
   procedure :: get_subgroups_bij => get_subgroups_bij
   procedure :: get_subgroups_cij => get_subgroups_cij
   procedure :: check_consistency
end type GeGCModelParameters
```

Subgroups IDs:

[1, 2, 3, 50, 51, 63]

Subgroup id $\neq$ Vector index

Subgroups' maingroups

[1, 1, 1, 2, 2, 5]

Maingroup id $\neq$ Vector index

```fortran
  function get_subgroup_index(self, subgroup_id) result(subgroup_idx)
     class(GeGCModelParameters) :: self

     integer, intent(in) :: subgroup_id
     !! ID of the subgroup
     integer :: subgroup_idx
     !! Index of the subgroup on the `self%subgroups_ids` vector

     subgroup_idx = findloc(self%subgroups_ids, subgroup_id, dim=1)
  end function get_subgroup_index
```

Find the index of a subgroup id

```fortran
function get_maingroup_index(self, maingroup_id) result(maingroup_idx)
     class(GeGCModelParameters) :: self

     integer, intent(in) :: maingroup_id
     !! ID of the subgroup
     integer :: maingroup_idx
     !! Index of the maingroup on the `self%maingroups_ids` vector

     maingroup_idx = findloc(self%maingroups_ids, maingroup_id, dim=1)
  end function get_maingroup_index
```

The same, but maingroups

```fortran
function get_subgroup_maingroup(self, subgroup_id)
result(subgroup_maingroup)
     class(GeGCModelParameters) :: self

     integer, intent(in) :: subgroup_id
     !! ID of the subgroup
     integer :: subgroup_maingroup
     !! Maingroup of the subgroup

     integer :: subgroup_idx

     subgroup_idx = self%get_subgroup_index(subgroup_id)

     subgroup_maingroup = self%subgroups_maingroups(subgroup_idx)
  end function get_subgroup_maingroup
```

Reusing a method

Accessing an attribute

```fortran
function get_subgroups_aij(self, subgroup_i_id, subgroup_j_id) result(aij)
   class(GeGCModelParameters) :: self

   integer, intent(in) :: subgroup_i_id
   !! ID of the subgroup `i`
   integer, intent(in) :: subgroup_j_id
   !! ID of the subgroup `j`
   real(pr) :: aij
   !! Interaction parameter \(a_{ij}\)

   integer :: mi_id, mj_id, i, j

   mi_id = self%get_subgroup_maingroup(subgroup_i_id)
   mj_id = self%get_subgroup_maingroup(subgroup_j_id)

   i = self%get_maingroup_index(mi_id)
   j = self%get_maingroup_index(mj_id)

   aij = self%maingroups_aij(i, j)
end function get_subgroups_aij
```

Reusing a method to obtain the subgroups' maingroups

Obtain maingroups indexes

Access interaction matrix attribute. Retrieve interaction between subgroups

Some caveats of (our) OOP, and how we fight it

Inheritance can also cause a lot of headaches,

```fortran
subroutine Bmix(self, n, bi, B, dBi, dBij)
     class(QMR), intent(in) :: self
     real(pr), intent(in) :: n(:)
     real(pr), intent(in) :: bi(:)
     real(pr), intent(out) :: B, dBi(:), dBij(:, :)
     call bmix_qmr(n, bi, self%l, b, dbi, dbij)
  end subroutine Bmix
```

```fortran
subroutine BmixMHV(self, n, bi, B, dBi, dBij)
     class(MHV), intent(in) :: self
     real(pr), intent(in) :: n(:)
     real(pr), intent(in) :: bi(:)
     real(pr), intent(out) :: B, dBi(:), dBij(:, :)
     call bmix_qmr(n, bi, self%l, b, dbi, dbij)
  end subroutine BmixMHV
```