Achilles Fortran Interface

Josh Isaacson

with William Jay, Alessandro Lovato, Pedro Machado, Noemi Rocco and Luke Pickering

Workshop on Neutrino Event Generators

15 March 2023

Motivation

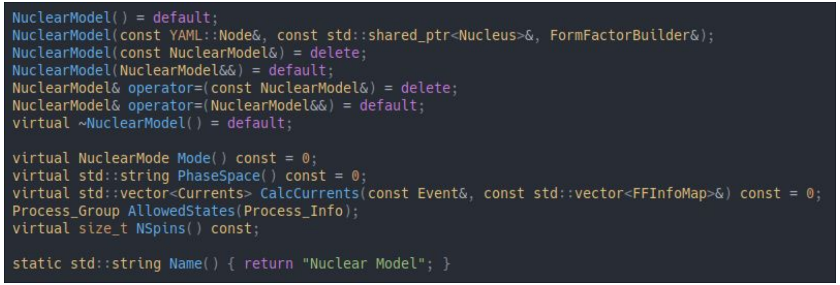
- Many nuclear models are written in Fortran
- Most event generators are written in C++
- Providing a standardized interface provides the ability to quickly adopt new models in experimental analyses
- This will require some modifications of both the C++ code and the Fortran code
- Achilles provides a methodology of interfacing the two codes that is minimally intrusive, but in such a way to ensure consistency of physical parameters throughout the models

General Requirements on Nuclear Models

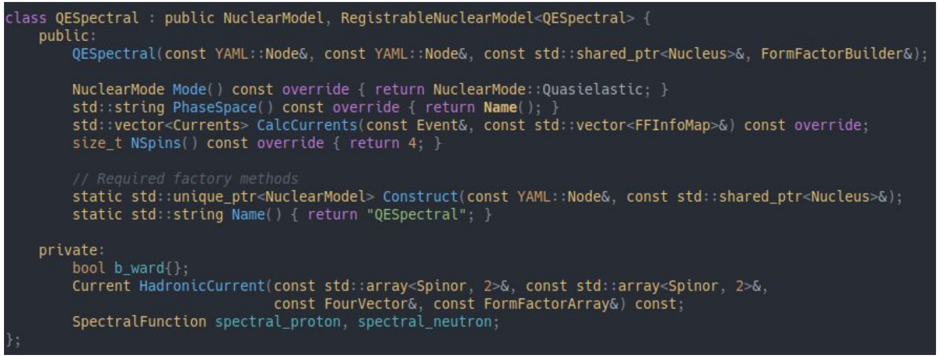
- Models should be fully exclusive (i.e. retains information about all initial and final state particles)
- Ideally, support calculation of just the hadronic current (but can support hadronic tensors if needed)
- Must define:
  - Name of the model
  - Reference to paper model is based on (in progress)
  - Mode: Quasielastic, meson exchange current, resonance production, DIS, etc.
  - How to handle the initial state momentum generation
  - Form factors used in the calculation (in progress)

Overview of C++ Interface

- Models are automatically registered into a factory using CRTP
- Initialization of an arbitrary model is handled by passing in a YAML Node containing runtime parameters
- A general purpose form factor builder is used to load and handle all required form factors in the calculation



```cpp
NuclearModel() = default;
NuclearModel(const YAML::Node&, const std::shared_ptr<Nucleus>&, FormFactorBuilder&);
NuclearModel(const NuclearModel&) = delete;
NuclearModel(NuclearModel&&) = default;
NuclearModel& operator=(const NuclearModel&) = delete;
NuclearModel& operator=(NuclearModel&&) = default;
virtual ~NuclearModel() = default;

virtual NuclearMode Mode() const = 0;
virtual std::string PhaseSpace() const = 0;
virtual std::vector<Currents> CalcCurrents(const Event&, const std::vector<FFInfoMap>&) const = 0;
Process_Group AllowedStates(Process_Info);
virtual size_t NSpins() const;

static std::string Name() { return "Nuclear Model"; }
```



```cpp
class QESpectral : public NuclearModel, RegistrableNuclearModel<QESpectral> {
    public:
        QESpectral(const YAML::Node&, const YAML::Node&, const std::shared_ptr<Nucleus>&, FormFactorBuilder&);

        NuclearMode Mode() const override { return NuclearMode::Quasielastic; }
        std::string PhaseSpace() const override { return Name(); }
        std::vector<Currents> CalcCurrents(const Event&, const std::vector<FFInfoMap>&) const override;
        size_t NSpins() const override { return 4; }

        // Required factory methods
        static std::unique_ptr<NuclearModel> Construct(const YAML::Node&, const std::shared_ptr<Nucleus>&);
        static std::string Name() { return "QESpectral"; }

    private:
        bool b_ward{};
        Current HadronicCurrent(const std::array<Spinor, 2>&, const std::array<Spinor, 2>&,
                                const FourVector&, const FormFactorArray&) const;
        SpectralFunction spectral_proton, spectral_neutron;
};
```

Handling of the phase space

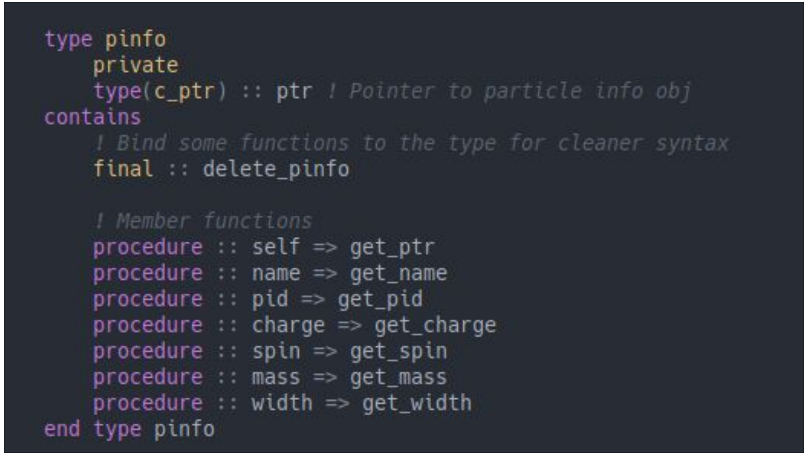
- Phase space is handled by factorizing the phase space into:
  - Neutrino beam
  - Initial nuclear state
  - Final state phase space
- Currently implemented initial nuclear states:
  - Stationary nucleus
  - Stationary nucleon
  - Single nucleon momentum distributed according to a spectral function
- Straightforward to implement new initial states, just need a parameterization

Interface to External Calculations

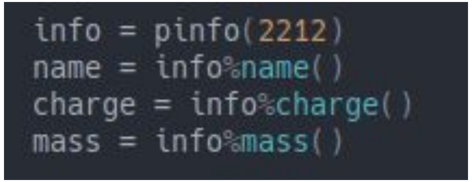
- Achilles supplies:
  - All physical constants ($\hbar$, c, etc.)
  - All particle properties
  - An interface to spectral functions
  - Four momentum and particle ids for all particles in the hadronic current
  - A string to use to initialize all properties of the nuclear model
- External code supplies:
  - Hadronic current (tensor) for a given phase space point and particle ids
  - An initialization function that takes an input string and sets up any needed components

Fortran Interface: Achilles to Nuclear Model

- Provides interface to Achilles runtime particle database:

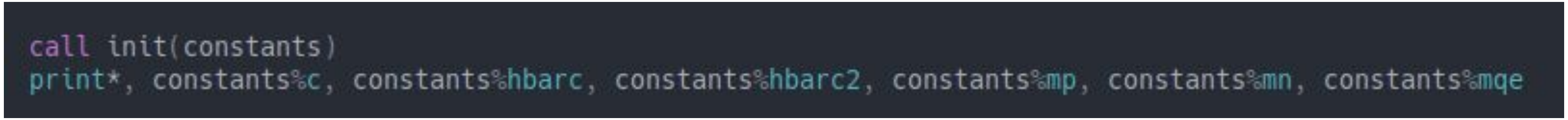


```fortran
type pinfo
    private
    type(c_ptr) :: ptr ! Pointer to particle info obj
contains
    ! Bind some functions to the type for cleaner syntax
    final :: delete_pinfo

    ! Member functions
    procedure :: self => get_ptr
    procedure :: name => get_name
    procedure :: pid => get_pid
    procedure :: charge => get_charge
    procedure :: spin => get_spin
    procedure :: mass => get_mass
    procedure :: width => get_width
end type pinfo
```



```fortran
info = pinfo(2212)
name = info%name()
charge = info%charge()
mass = info%mass()
```

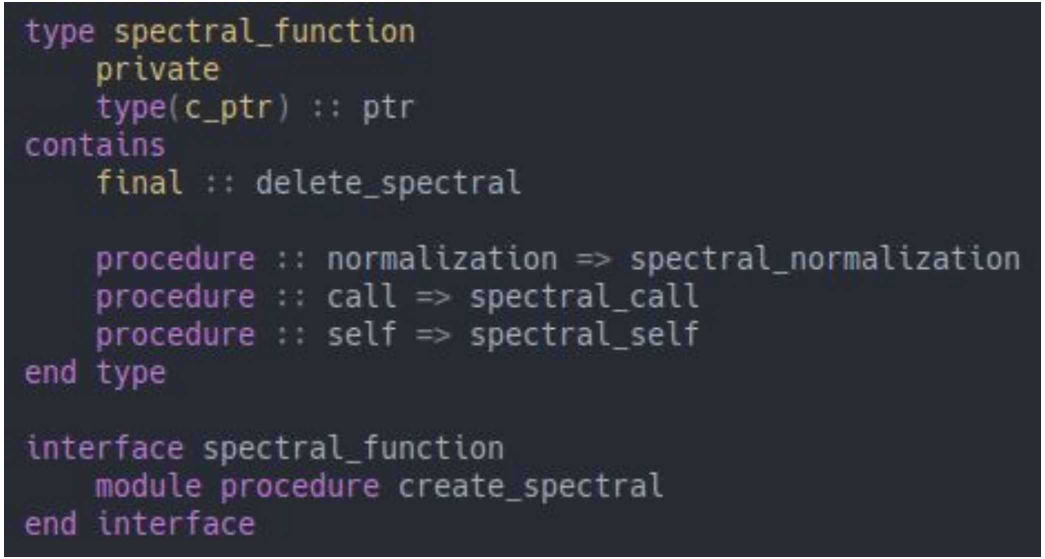
- Provides interface to physical constants defined in Achilles:



```fortran
call init(constants)
print*, constants%c, constants%hbarc, constants%hbarc2, constants%mp, constants%mn, constants%mqe
```

Fortran Interface: Achilles to Nuclear Model

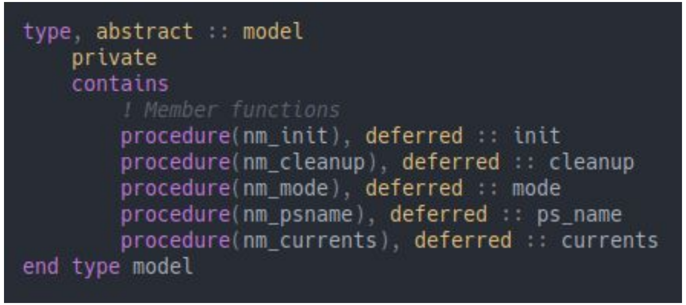
- Provides interface to Achilles handling of spectral functions:



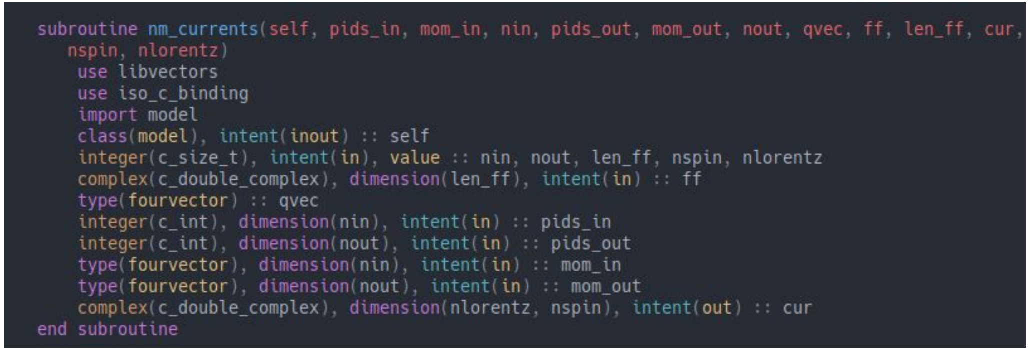
```fortran
type spectral_function
    private
    type(c_ptr) :: ptr
contains
    final :: delete_spectral

    procedure :: normalization => spectral_normalization
    procedure :: call => spectral_call
    procedure :: self => spectral_self
end type

interface spectral_function
    module procedure create_spectral
end interface
```

Fortran Interface: Nuclear Model to Achilles

- Model inherits from an abstract model type and must define:
  - An initialization function
  - A clean-up function for handling any allocated memory
  - A function returning the nuclear mode
  - The name of the required initial state phase space generator
  - A function that calculates the hadronic currents



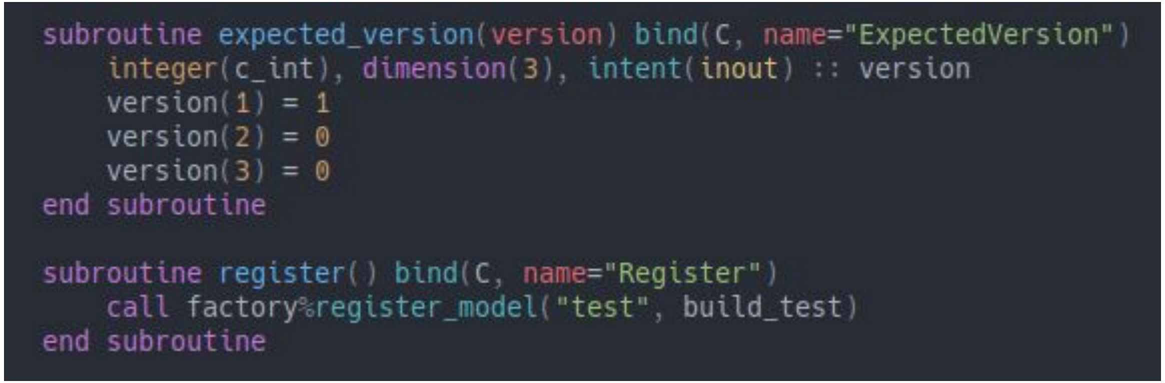
```fortran
type, abstract :: model
    private
    contains
        ! Member functions
        procedure(nm_init), deferred :: init
        procedure(nm_cleanup), deferred :: cleanup
        procedure(nm_mode), deferred :: mode
        procedure(nm_psname), deferred :: ps_name
        procedure(nm_currents), deferred :: currents
end type model
```



```fortran
subroutine nm_currents(self, pids_in, mom_in, nin, pids_out, mom_out, nout, qvec, ff, len_ff, cur,
    nspin, nlorentz)
     use libvectors
     use iso_c_binding
     import model
     class(model), intent(inout) :: self
     integer(c_size_t), intent(in), value :: nin, nout, len_ff, nspin, nlorentz
     complex(c_double_complex), dimension(len_ff), intent(in) :: ff
     type(fourvector) :: qvec
     integer(c_int), dimension(nin), intent(in) :: pids_in
     integer(c_int), dimension(nout), intent(in) :: pids_out
     type(fourvector), dimension(nin), intent(in) :: mom_in
     type(fourvector), dimension(nout), intent(in) :: mom_out
     complex(c_double_complex), dimension(nlorentz, nspin), intent(out) :: cur
end subroutine
```

Fortran Interface: Nuclear Model to Achilles

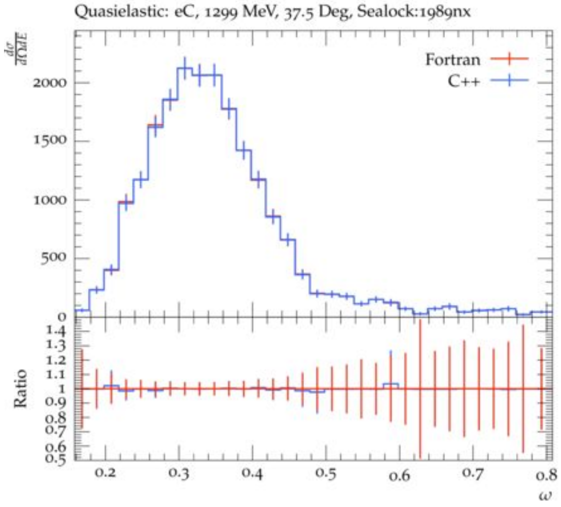
- Dynamically loading as a plugin must:
  - Supply function “expected_version” returning the minimum Achilles version
  - Supply function “register” that registers the fortran model with Achilles
  - Be compiled into a shared library of the format libAchillesPlugin_*.so (on linux for example)
  - The library must be in the Achilles installed share folder, or in a folder in the ACHILLES_PLUGIN_PATH environmental variable



```fortran
subroutine expected_version(version) bind(C, name="ExpectedVersion")
    integer(c_int), dimension(3), intent(inout) :: version
    version(1) = 1
    version(2) = 0
    version(3) = 0
end subroutine

subroutine register() bind(C, name="Register")
    call factory%register_model("test", build_test)
end subroutine
```

Validation

- Comparison between C++ version and Fortran version
- Demonstrates that the two reproduce the same results
- This is a sample of 10k events, and uncertainty bands show the statistical uncertainty
- There is a similar level of agreement in more exclusive channels



Conclusions and Open Questions

- Conclusions:
  - Achilles provides an interface to arbitrary nuclear models in either C++ or Fortran (other languages can be added if there is a need)
  - We enforce a minimum number of requirements to ensure that the physical parameters are consistent amongst all nuclear models
  - We provide a method of dynamically including your model using a plugin structure. This enables the theory community to test out changes to their model without having to modify the internals of the Achilles code
- Open Questions:
  - Is this structure flexible enough for your nuclear model?
  - What else can we provide to make the inclusion of your favorite nuclear model more straightforward?
  - Is there anything we might have missed in this interface?