Modern Fortran for Fluid–Structure Simulations of Cilia and Particles

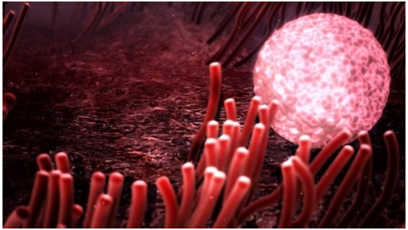
Divyaprakash

PhD Candidate
Indian Institute of Technology Delhi, India

FortranCon 2025

Motivation

Sensing and transporting particles using artificial cilia is a key challenge in microfluidics and bio-engineering.



Biological inspiration (e.g., Ovum Transport)
Source: Web

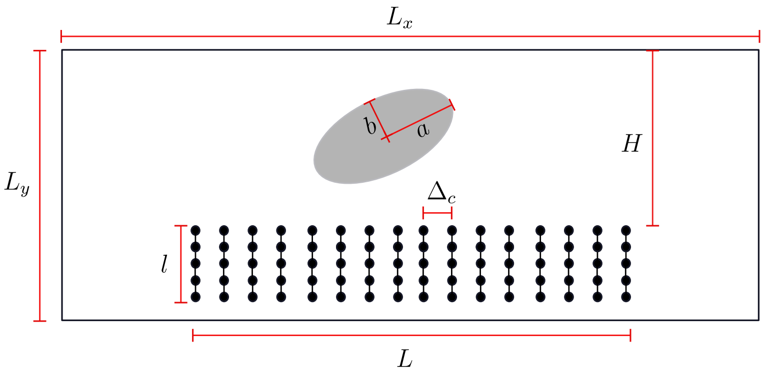
Our Goal: To build a high-fidelity simulation of this complex, multiphysics system.

This involves coupling:

- **Fluid:** A 2D incompressible Navier-Stokes solver.
- **Solids (Cilia):** Modeled as flexible Kirchhoff Rods.
- **Solids (Particles):** Modeled using Finite Elements (FEM).

Problem Statement

Develop computational models to understand and mimic cilia-particle interactions.

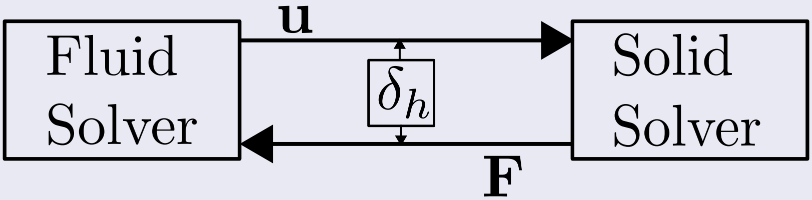


Computational Domain

The Computational Framework

We use the **Immersed Boundary Method (IBM)** to couple the Fluid and Solid solver

High-Level Solver Architecture



The Core Challenge: How do we build this entire system to be...

- **Maintainable:** Cleanly separated code (fluid vs. solid).
- **Extensible:** Easy to add new objects (e.g., capsules, different particles).
- **High-Performance:** Fast enough for large-scale simulations.

Our Solution: A framework built entirely in **Modern Fortran**.

Why Modern Fortran?

Fortran is more than just a "number-crunching" language. Modern features allow for elegant, robust, and maintainable software design.

We leverage five key features:

1. **Modular Programming (`MODULE`)**
   For separation of concerns.
2. **Object-Oriented Design (`TYPE`, `CONTAINS`)**
   To represent physical objects cleanly in code.
3. **Powerful Array Syntax**
   Built-in matrix/array operations made translating our MATLAB prototype simple.
4. **Dynamic Memory (`ALLOCATABLE`)**
   For flexible, runtime-defined problem sizes.
5. **C Interoperability (`ISO_C_BINDING`)**
   For GPU acceleration and external libraries.

Core Design: Modular Programming

Modules allow us to separate the code by its physical function: fluid, solids, coupling, I/O, etc.
The main program (`ibmc.f90`) simply composes these modules:

```fortran
program ibmc
    ! --- Core Physics Modules ---
    use mod_pressure,       only: generate_laplacian_sparse_constant, ...
    use mod_time,           only: advection, diffusion, corrector, ...
    use mod_boundary,       only: apply_boundary_channel
    use mod_ibm,            only: spread_force, interpolate_velocity

    ! --- Object Definition Modules ---
    use mod_mesh,           only: mesh
    use fem2d,              only: festruct       ! Particle type
    use mod_cilia,          only: cilium         ! Cilium type

    ! --- Helper/External Modules ---
    use mod_amgx,           only: calculate_pressure_amgx ! GPU Solver
    use mod_io,             only: write_field, write_mesh

    ! ...

    ! --- Local variables declared here ... ---
    type(mesh)      :: M
    type(cilium), allocatable :: cilia(:)
    type(festruct), allocatable :: particles(:)
    ...
end program ibmc
```

Object-Oriented Design: Derived Types

We represent physical objects (cilia, particles) as **Derived Types**. This bundles data (e.g., position, force) with behavior (e.g., 'calculate_forces').

First, we define an **abstract** base type for all solids (`mod_solid.f90`):

```fortran
module mod_solid
  implicit none
  type, abstract :: solid
    ! ... (common data)
    real(real64), allocatable :: XE(:,:) ! Coords
    real(real64), allocatable :: fden(:,:) ! Force
    real(real64), allocatable :: U(:,:)    !
    ↪  Velocity
    real(real64) :: dl
    ! ...
  end type solid
end module mod_solid
```

Then, **Cilia** and **Particles** *extend* this base type and add their own specific logic (`mod_cilia.f90`):

```fortran
module mod_cilia
  use mod_krod ! Extends solid
  implicit none

  type, extends(krod) :: cilium
    ! --- Data ---
    integer(int32) :: NV ! Num vertices
    real(real64)   :: K, B ! Stiffness
    real(real64), allocatable :: Mom(:)

  contains
    ! --- Behavior (Type-Bound Procs) ---
    procedure :: forces
    procedure :: moments
    ! ..
  end type cilium
  ...
contains
  elemental impure subroutine forces(self,...)
    class(cilium), intent(inout) :: self
```

High-Level Syntax & Compiler-Friendly Code

Fortran's array-oriented nature extends to derived types, which was critical for moving from a MATLAB prototype to efficient, compiled code.

1. Elemental Procedures We define our object methods as `elemental`. This means the subroutine is written to act on a *scalar* object (`self`).

```
elemental impure subroutine forces(self, ftip)
  class(cilium), intent(inout) :: self
  real(real64), optional, intent(in) :: ftip
  ! ...
end subroutine forces
```

2. High-Level Array Calls This allows us to call the procedure on an *entire array* of objects with a clean, MATLAB-like syntax.

```
! In main program (ibmc.f90)
type(cilium), allocatable :: cilia(:)
allocate(cilia(ncilia))
...
! This single line...
call cilia(:)%forces(ftip)
```

Why this is efficient

The `elemental` keyword is a powerful optimization flag. It explicitly states that the `forces` calculation for one cilium is **independent** of all others. This gives the compiler permission to automatically **vectorize** or **parallelize** the resulting loop, leading to significant performance gains.

Polymorphism: Writing Generic Solvers

The Problem: Our coupling module (`mod_ibm`) needs to `spread_force` from *different* kinds of objects (cilia, particles) to the fluid grid. We want to avoid writing duplicate code.

1. Inheritance: Create a Base Type First, we define an `abstract` base type, `solid`, which holds data common to all objects (like position and force).

```
! In mod_solid.f90
module mod_solid
  type, abstract :: solid
    real(real64), allocatable :: XE(:,:) ! Coords
    real(real64), allocatable :: fden(:,:) ! Force
    real(real64), allocatable :: U(:,:)     !
    ↪  Velocity
    real(real64) :: dl
    ...
  end type solid
end module mod_solid
```

Our objects then `extend` this base type.

```
! In fem2d.f90 (Particle)
type, extends(solid) :: festruct
  ...
end type festruct

! In mod_cilia.f90 (via mod_krod)
type, extends(krod) :: cilium
  ...
end type cilium
```

2. Polymorphism: Use `CLASS` The `spread_force` subroutine accepts `class(solid)`, not `type(cilium)` or `type(festruct)`.

```
! In mod_ibm.f90
subroutine spread_force(M, B, Fx, Fy)
  class(mesh), intent(in) :: M

  ! This argument is POLYMORPHIC
  class(solid), intent(in out) :: B(:)

  real(real64), intent(in out) :: Fx(...), Fy(...)
  ...
  ! Accesses common 'solid' data
  Lx = B(irod)%XE(inp,1)
  Flx = B(irod)%fden(inp,1) * B(irod)%dl
  ...
end subroutine spread_force
```

This allows the *same function* to be called with different object arrays

```
! In main program (ibmc.f90)
call spread_force(M, particles, Fx, Fy)
call spread_force(M, cilia, Fx, Fy)
```

The Payoff: A Clean Main Loop

The Object-Oriented design makes the main program (`ibmc.f90`) read like the algorithm, hiding the complexity inside each object's methods.

```fortran
! (Inside main loop...)

! 1. Calculate internal forces for all solid objects
!    This is a "polymorphic" call on the object arrays
call particles(:)%calculate_forces()
call cilia(:)%forces(ftip)

! 2. Spread forces from objects to the fluid grid (IBM)
Fx = 0.0d0; Fy = 0.0d0
! Note: spread_force takes 'class(solid), allocatable :: B(:)'
call spread_force(M, particles, Fx, Fy)
call spread_force(M, cilia, Fx, Fy)

! 3. Solve Fluid (Advection, Diffusion, Pressure-Project)
call advection(M,u,v,au,av)
...
call calculate_pressure_amgx(A,P,R,init_status)
call corrector(M,u,v,us,vs,P,rho,dt)
```

Performance: Fortran-C Interoperability

The bottleneck is solving the pressure Poisson equation ($Ax = b$). We accelerate this by offloading it to an NVIDIA AmgX (GPU) library.
This is achieved with the `iso_c_binding` module.

1. Define the C Interface In `ftn_c.f90`, we define the C function's prototype:

```fortran
module ftn_c
  interface
    integer (c_int) function solveamg(rhs, sol) &
                   bind(c, name='solveamg')
      use iso_c_binding
      implicit none
      type (c_ptr), value :: rhs
      type (c_ptr), value :: sol
    end function solveamg
  end interface
end module ftn_c
```

2. Call C from Fortran In `mod_amgx.f90`, we pass Fortran data using `c_loc`:

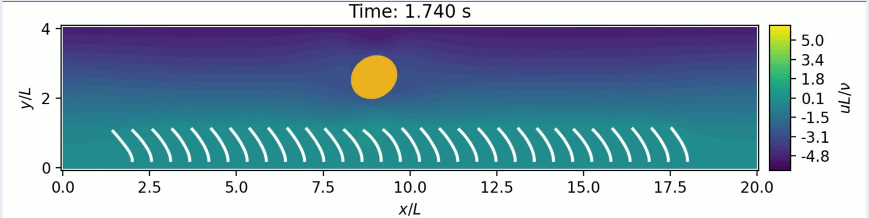
```fortran
module mod_amgx
  use iso_c_binding, only: c_int, c_double, c_loc
  use ftn_c ! Our interface module
  ...
contains
  subroutine solve_amgx()
    ! rhsv and sol are Fortran allocatables
    ! defined with 'target' attribute
    real(c_double), allocatable, target :: rhsv(:)
    real(c_double), allocatable, target :: sol(:)
    ...
    ! Pass the memory location to C
    solve_am = solveamg(c_loc(rhsv), c_loc(sol))
  end subroutine solve_amgx
```

Results: The Solver in Action

The resulting framework is robust, high-performance, and produces physically accurate results.

Demonstration: 3D Simulation

Simulation of an elliptical particle moving over an array of cilia.



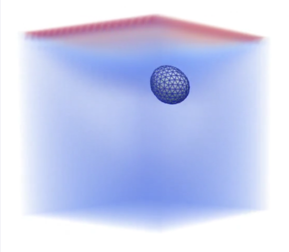
Example Application: The framework can be used to generate data for machine learning models to sense particle properties from cilia deflection alone (as shown in my PhD work).

Future Work: Coupling Fortran with OpenFOAM (C++)

The same `iso_c_binding` interoperability was used to couple our Fortran solid solver with the C++-based OpenFOAM library, enabling full 3D simulations.

Demonstration

3D simulation of a deformable particle, computed with the coupled OpenFOAM-Fortran solver.



A simplified version of this solver is open-source:

https://github.com/divyaprakash-iitd/ciliaparticlefoam

Summary and Conclusions

We have successfully built a complex, multiphysics FSI solver for cilia and particles from scratch.

Modern Fortran was the key enabler:

- **Modules** provided a clean, decoupled architecture.
- **Derived Types** and **Type-Bound Procedures** allowed us to create a clean, object-oriented design that is readable and extensible.
- **Polymorphism** (‘class(solid)‘) enabled us to write generic, reusable coupling code (like `spread_force`).
- `iso_c_binding` provided a seamless, high-performance path to GPU acceleration.

Main Message

Modern Fortran provides the performance of a low-level language with the high-level abstractions needed to tackle complex, modern scientific challenges.

Thank You

Author:

Divyaprakash
divyaprakash.poddar@gmail.com
https://dpcfd.com

Advisor:

Prof. Amitabh Bhattacharya
Department of Applied Mechanics
Indian Institute of Technology Delhi, India

Code Repository:

https://github.com/divyaprakash-iitd/ibmc

Note: This is a similar, simplified version. The full repository will be open-sourced soon.

Questions?