# NNPred Fortran API Document

Weishuo Liu[a]

[a]*School of Energy and Power Engineering, Beihang University, 37 Xueyuan Road, Haidian District, Beijing, 100191, China*

## A+B Model and Code Example

A simple **A + B = C** model is presented to show the deployment process. This model computes the sum of two 2D arrays (i.e., **A** and **B**) and holds the result in the output node, **C**. The I/O nodes' dimension sizes are $n \times 2$, with $n$ being the number of data instances, and their data type is specified as the single-precision floating point. Figure. 1 shows a minimal application case of the model deployment and interaction with external arrays (i.e., `Array_A`, `Array_B` and `Array_C` ). In this case, `Array_B` has different data type from the node's definition (`int` vs `float`), and `Array_C` differs to node, `output_c`, in both data type (`double` vs. `float`) and memory layout ($3 \times 2$ vs. $3 \times 2$).

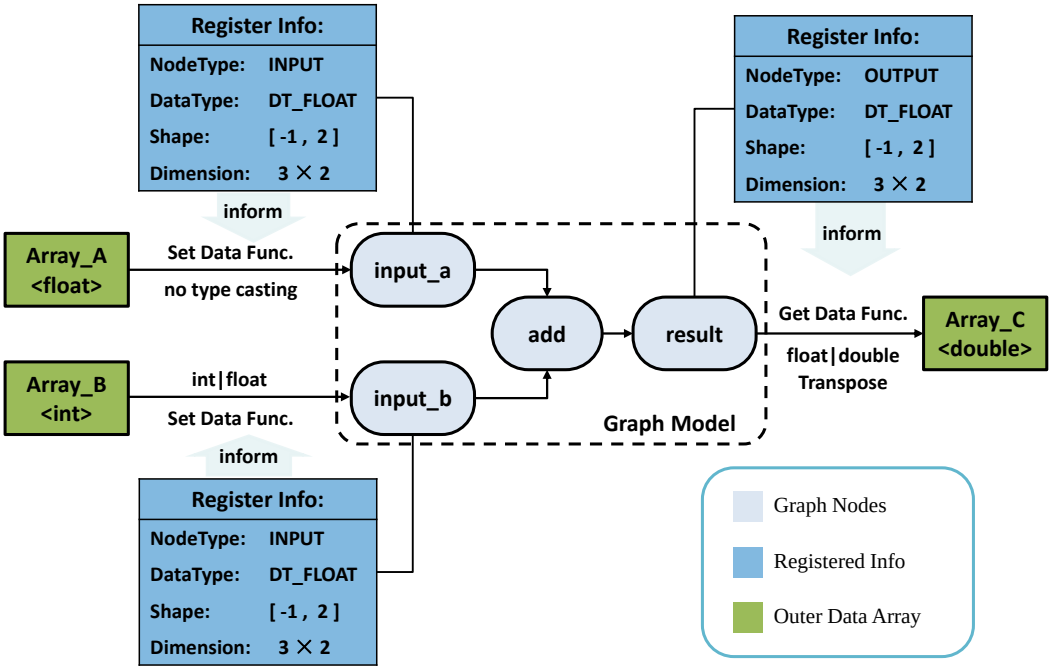


Figure 1: Interaction relation within NNPred elements and with outer memory spaces during a prediction process.

The code example to demonstrate the entire deploying process can be found in Listing 1. The whole process can be divided into two major parts: initialization and prediction. The initialization consists of loading the model, registering the input/output nodes, and setting the number of data instances, whereas the prediction part includes setting the input data, running predictions, and extracting the result. The location and function prototype of each step are listed in Table. 1.

*Email address:* `liuweishuo@buaa.edu.cn` (Weishuo Liu)

Table 1: ID, function prototypes and locations of necessary steps in Fortran

| Step ID | Usage | Function Prototype | Location |
|---|---|---|---|
| F01 | Load Model | `ptr = C_CreatePredictor(file_name)` * | 13-14 |
| F02 | Register Inputs | `C_PredictorRegisterInputNode(ptr, in_node_name)` | 16-18 |
| F03 | Register Outputs | `C_PredictorRegisterOutputNode(ptr, in_node_name)` | 20-21 |
| F04 | Set Data Counts | `C_PredictorSetDataCount(ptr, n_data)` | 23-24 |
| F05 | Set Input Data | `C_PredictorSetNodeData(ptr, in_node, input_arr , n_element)` | 26-28 |
| F06 | Run Model | `C_PredictorRun(ptr)` | 30-31 |
| F07 | Get Output Data | `C_PredictorGetNodeData(ptr, out_node, output_arr , n_element)` | 33-34 |
| F08 | Finalize Model | `C_DeletePredictor(ptr)` | 39-40 |

* The type of the returned value should be declared as: `type(c_ptr) :: ptr`.

```fortran
program main
use ml_predictor  !The predictor module that declared all the function interfaces
use iso_c_binding, only: c_ptr  !C-pointer to the Predictor instance

    implicit none
    type(c_ptr) :: ptr    ! C-pointer to the Predictor instance

    ! External input/output arrays from fortran program (Support up to 6d)
    real(kind=4), dimension(2,3)  :: arr_a = reshape((/0.0, 1.1, 2.2, 3.3, 4.4, 5.5/),
  (/2,3/))
    integer(kind=4), dimension(6) :: arr_b = (/5, 4, 3, 2, 1, 0/)
    real(kind=8), dimension(3,2)  :: arr_c = 0.0

    ! Create predictor from *.pb
    ptr = C_CreatePredictor("simple_graph_tf2.pb")

    ! Register input nodes
    call C_PredictorRegisterInputNode(ptr, "input_a")
    call C_PredictorRegisterInputNode(ptr, "input_b")

    ! Register output nodes
    call C_PredictorRegisterOutputNode(ptr, "result")

    ! Set number of data instances
    call C_PredictorSetDataCount(ptr, 3)

    ! Set the input data
    call C_PredictorSetNodeData(ptr, "input_a", arr_a, 6)
    call C_PredictorSetNodeData(ptr, "input_b", arr_b, 6)

    ! Run the model
    call C_PredictorRun(ptr)

    ! Get the model output data into arr_c with transpose options
    call C_PredictorGetNodeDataTranspose(ptr, "result", arr_c, 6)

    ! Print the output
    print*, "Calculation Result:", arr_c

    ! Delete predictor when it is not used anymore
    call C_DeletePredictor(ptr)

end program main

```

Listing 1: Minimal example to run A + B model in Fortran

In the initialization process, the node information (name, shape, and data type) is cached by the register-node step after the model is loaded. Once the number of data instances is set, the unknown dimension is determined and the

internal buffers are created. In the prediction process, the set-data function will fill the input buffers, and the get-data function extracts the results after the prediction is executed. Meanwhile, the I/O functions could automatically cast the data type to get correct results, and the array can be mapped to a different memory layout with transpose options.

### Entire API list for Fortran

- **F01: Loading Model:**
  - `ptr = C_CreatePredictor(file_name)`
    - `file_name` – the Fortran `CHARACTER` array specifying the file name of the PB graph.

    To create the `Predictor` object from a `*.pb` format, which return the reference to the `Predictor` object, the return value, `ptr` should be defined as `type(c_ptr)::ptr`.
  - `ptr = C_CreatePredictor(model_dir, tag)`
    - `model_dir` – the Fortran `CHARACTER` array specifying the directory of the `SavedModel` format (this format itself is a folder).
    - `tags` – the Fortran `CHARACTER` array specifying the tags label within a `SavedModel` format, by default is `serve`.

    To create the `Predictor` object from a `SavedModel` format, which return the reference to the `Predictor` object, the return value, `ptr` should be defined as `type(c_ptr)::ptr`.
- **F02: Register Inputs:**
  - `C_PredictorRegisterInputNode(ptr, in_node_name)`
    - `ptr` – the C pointer reference to the created `Predictor` object.
    - `in_node_name` – the Fortran `CHARACTER` array specifying the name of the node to be registered as input.

    To register input node for feeding data.
- **F03: Register Outputs:**
  - `C_PredictorRegisterOutputNode(ptr, out_node_name)`
    - `ptr` – the C pointer reference to the created `Predictor` object.
    - `out_node_name` – the Fortran `CHARACTER` array specifying the name of the node to be registered as output.

    To register output node for extracting data.
- **F04: Set Data Counts:**
  - `C_PredictorSetDataCount(ptr, n_data)`
    - `ptr` – the C pointer reference to the created `Predictor` object.
    - `n_data` – the Fortran `INTEGER` variable specifying number of data instances.

    To set the number of data instances to substitute the unknown dimension of input/output tensors. For example, in the A + B example, the input/output shapes are all [-1, 2], the function could set the unknown -1 into concrete value so that the inner data containers can be created.
- **F05: Set Input Data:**
  - `C_PredictorSetNodeData(ptr, in_node, input_arr , n_element)`
    - `ptr` – the C pointer reference to the created `Predictor` object.
    - `in_node` – the Fortran `CHARACTER` array specifying the name of the input node to be fed with external data.
    - `input_arr` – the Fortran numerical array holding the external data, available data type: `INTEGER`, `REAL(4)` and `REAL(8)`.
    - `n_element` – the Fortran `INTEGER` variable specifying the number of data elements of the external data array.

    To feed the internal input data container registered under `in_node` with external data array. Because the shape information of a Fortran array is lost when passing to a C library, so the total number of elements in the array needs to be specified. If the data type of the internal container and external source are different, this function would automatically cast the datatype to resolve the difference.
  - `C_PredictorSetNodeDataTranspose(ptr, in_node, input_arr , n_element)`

    * `ptr` – the C pointer reference to the created `Predictor` object.
    * `in_node` – the Fortran `CHARACTER` array specifying the name of the input node to be fed with external data.
    * `input_arr` – the Fortran numerical array holding the external data, available data type: `INTEGER`, `REAL(4)` and `REAL(8)`.
    * `n_element` – the Fortran `INTEGER` variable specifying the number of data elements of the external data array.

  To feed the internal input data container registered under `in_node` with external data array. Because the shape information of a Fortran array is lost when passing to a C library, so the total number of elements in the array needs to be specified. If the data type of the internal container and external source are different, this function would automatically cast the datatype to resolve the difference. The memory layout would be tansposed during the mapping process.

- **F06: Run Model:**
  - `C_PredictorRun(ptr)`
    * `ptr` – the C pointer reference to the created `Predictor` object.

    To run the model prediction, the result will be stored in the internal data container holding the model's output.
- **F07: Get Output Data:**
  - `C_PredictorGetNodeData(ptr, out_node, output_arr , n_element)`
    * `ptr` – the C pointer reference to the created `Predictor` object.
    * `out_node` – the Fortran `CHARACTER` array specifying the name of the output node's data to be extracted to external array.
    * `output_arr` – the Fortran numerical array holding the external data, available data type: `INTEGER`, `REAL(4)` and `REAL(8)`.
    * `n_element` – the Fortran `INTEGER` variable specifying the number of data elements of the external data array.

    To extract the data stored in the the internal output data container registered under `out_node` into external data array. Because the shape information of a Fortran array is lost when passing to a C library, so the total number of elements in the array needs to be specified. If the data type of the internal container and external source are different, this function would automatically cast the datatype to resolve the difference.
  - `C_PredictorGetNodeDataTranspose(ptr, out_node, output_arr , n_element)`
    * `ptr` – the C pointer reference to the created `Predictor` object.
    * `out_node` – the Fortran `CHARACTER` array specifying the name of the output node's data to be extracted to external array.
    * `output_arr` – the Fortran numerical array holding the external data, available data type: `INTEGER`, `REAL(4)` and `REAL(8)`.
    * `n_element` – the Fortran `INTEGER` variable specifying the number of data elements of the external data array.

    To extract the data stored in the the internal output data container registered under `out_node` into external data array. Because the shape information of a Fortran array is lost when passing to a C library, so the total number of elements in the array needs to be specified. If the data type of the internal container and external source are different, this function would automatically cast the datatype to resolve the difference. The memory layout would be tansposed during the mapping process.
- **F08: Finalize Model:**
  - `C_DeletePredictor(ptr)`
    * `ptr` – the C pointer reference to the created `Predictor` object.

    To delete the `Predictor` object pointed by `ptr`, the manual finalization is needed because the resource created by C++ library would not be automatically released by Fortran.