IRP programming paradigm and its implementation in Fortran for quantum Monte Carlo

Roland Assaraf and **Julien Toulouse**
Université Pierre & Marie Curie and **CNRS**, Paris, France

CECAM discussion meeting, IDRIS, Orsay
March 2015

Quantum Monte Carlo program CHAMP

Scientific features

- VMC and DMC electronic-structure calculations of atoms, molecules, periodic solids, model systems.
- All-electron and pseudopotential calculations. Several types of basis functions.
- General multideterminant Jastrow-Slater wave functions.
- Wave-function optimization for ground and excited states.
- Calculations of energies and other properties.

Implementation features

- About 140 000 source lines of code.
- Parallelization by MPI. Scalability $\approx 100\%$ tested up to 4096 cores on IBM Blue Gene. Almost no inter-core communications.
- Initialy written in standard imperative Fortran by C. Umrigar and C. Filippi. **Progressively transformed in IRP** by J. Toulouse.

IRP?

A programming paradigm invented par François Colonna (LCT, Paris) which aims at producing **programs of low complexity**.

It was given different names over the years:

- *Open Structured Interfaceable Programming Environment* (OSIPE), 1994.
- *Deductive Object Programming*, 2006.
- *Implicit Reference to Parameters* (IRP), 2009.

- F. Colonna, L.-H. Jolly, R. A. Poirier, J. G. Ángyán, and G. Jansen, Comp. Phys. Comm. **81**, 293 (1994).
- F. Colonna, arxiv.org/abs/cs/0601035v1 (2006).
- A. Scemama, arxiv.org/abs/0909.5012v1 (2009).

Here we describe a **Fortran implementation** mainly developed by Roland Assaraf for the program QMCMOL, and later adapted for CHAMP.

See notes: `http://www.lct.jussieu.fr/pagesperso/toulouse/recherche/champ.pdf`

Basic IRP concepts and ideas

Objects

- **A computer program produces objects**, which in our case are Fortran variables (scalar/array) containing quantities that we are after.
- For example, `psi` may be an object containing the value of a wave function evaluated at some electron coordinates.
- This object `psi` is constructed from other objects, for example from the objects `jastrow` and `determinant`.
- The later objects are themselves constructed from yet other objects. And so on.

Basic IRP concepts and ideas

Objects

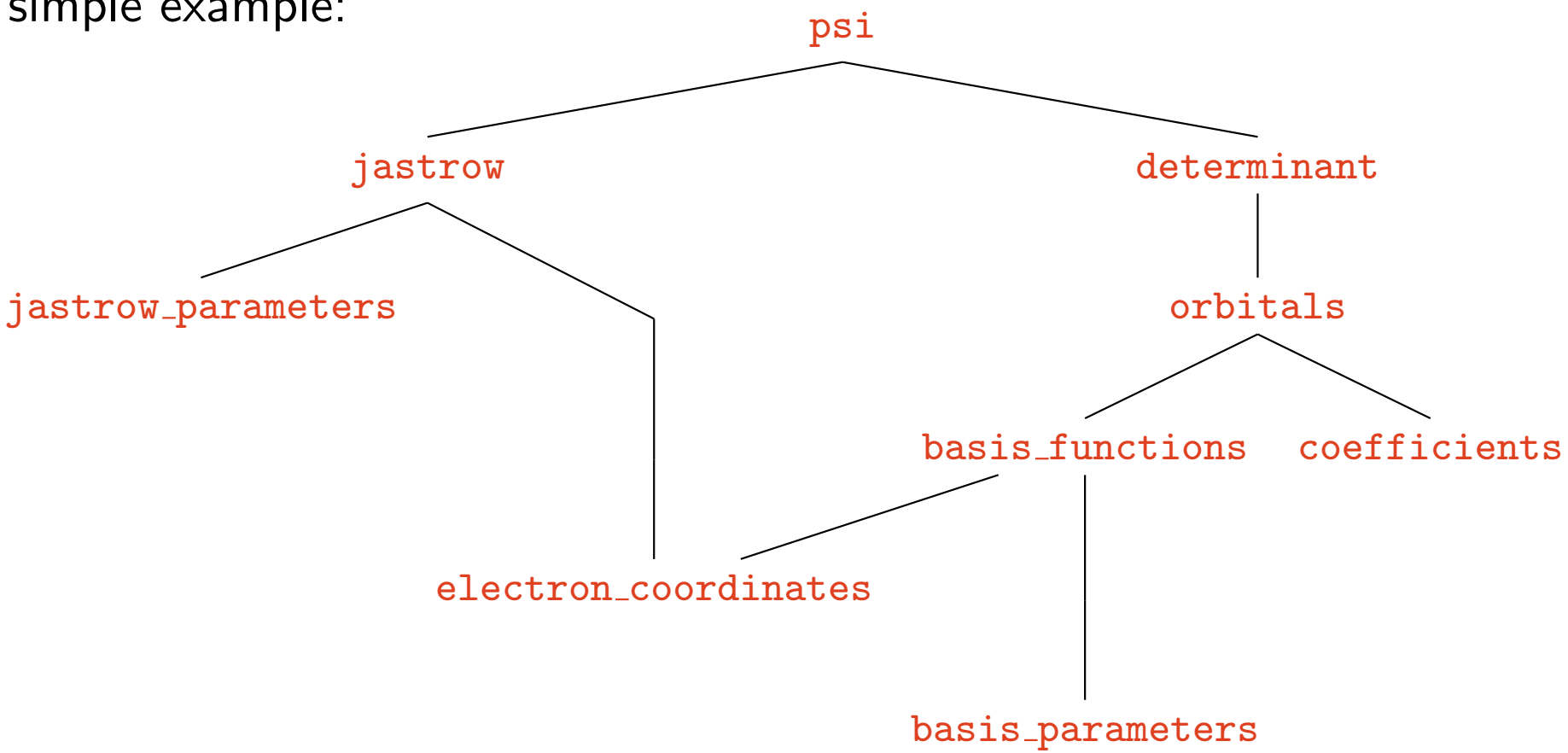
- **A computer program produces objects**, which in our case are Fortran variables (scalar/array) containing quantities that we are after.
- For example, `psi` may be an object containing the value of a wave function evaluated at some electron coordinates.
- This object `psi` is constructed from other objects, for example from the objects `jastrow` and `determinant`.
- The later objects are themselves constructed from yet other objects. And so on.

Dependencies

- Clearly, **there are dependencies between these objects**.
- Usually, these dependencies are **not** systematically handled.
- **In IRP, the dependencies are systematically handled**. The programmer does not need to take care of the order of construction of the objects.

The dependency "tree" of the objects

A simple example:



Vocabulary:

- `psi` is a **child** of `jastrow` and `determinant`.
- `jastrow` and `determinant` are the **parents** of `psi`.
- `jastrow_parameters`, `electron_coordinates`, `basis_parameters`, and `coefficients` are **leaves**.

Constructing the objects

Each object which is not a leaf of the tree has a **building subroutine** which constructs it.

For example, for `psi`, schematically:

```
subroutine psi_bld
  call object_provide('jastrow')
  call object_provide('determinant')
  psi = jastrow * determinant
end subroutine
```

The key subroutine: object_provide

`call object_provide('determinant')` does the following:

It checks if `determinant` is **valid**, i.e. already calculated and can be used.

- If yes, then nothing is done.
- If no, then its checks if its parent `orbitals` is valid.
  - If it is valid, it calls the building subroutine `determinant_bld` and marks `determinant` as valid.
  - If `orbitals` is not valid, then it checks its parents. And so on.

The key subroutine: object_provide

`call object_provide('determinant')` does the following:

It checks if `determinant` is **valid**, i.e. already calculated and can be used.

- If yes, then nothing is done.
- If no, then its checks if its parent `orbitals` is valid.
  - If it is valid, it calls the building subroutine `determinant_bld` and marks `determinant` as valid.
  - If `orbitals` is not valid, then it checks its parents. And so on.

Thus, `object_provide('determinant')` goes down recursively the dependency tree under `determinant` until it finds valid objects. It then climbs up the dependency tree, constructing the objects one after the other, in the correct order, until it finally constructs `determinant`.

Advantages of object_provide

`object_provide('determinant')` has several advantages:

- It is **simple**. The programmer just needs to know the name of the object that he wants, here `determinant`. He does not need to know about intermediate objects such as `orbitals`.
- It is **efficient**. This mechanism ensures that only the objects needed are calculated.
- It is **safe**. This mechanism ensures that each object is constructed before it is used. If a leaf object is needed but not valid, the program properly stops.

call object_modified

If the value of an object, say `electron_coordinates`, is modified we need to make sure that any child or grandchild object is recalculated if needed. This is done as follows:

```
electron_coordinates = (-2.1, 0.7, 1.5)
call object_modified('electron_coordinates')
```

call object_modified

If the value of an object, say `electron_coordinates`, is modified we need to make sure that any child or grandchild object is recalculated if needed. This is done as follows:

```
electron_coordinates = (-2.1, 0.7, 1.5)
call object_modified('electron_coordinates')
```

- The program marks `electron_coordinates` as valid, and recursively climbs up the dependency tree to mark as **invalid** all the objects depending on it.
- This is a **safe** mechanism since it prevents the programmer from forgetting to update objects in an iterative algorithm.
- A particular case of using `object_modified` is after reading leaf objects from the input or after calculating objects which are not in the dependency tree.