DEVELOPMENT IN WAVE FUNCTION METHODS MADE EASY

WITH IRPF90 AND THE QUANTUM PACKAGE

Anthony Scemama

07/02/2019

Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse

ISSUE

- Scientific codes need speed $\Longrightarrow$: Fortran / C / C++
- Low-level languages : close to the hardware $\Longrightarrow$ difficult to maintain
- High-level features of modern Fortran (array syntax, derived types, ...) or C++ (objects, STL) can kill the efficiency

We need to hide the code complexity and keep the code efficient.

ISSUE

A simple solution : use multiple languages.

- Low-level : computation
- High-level : text parsing, global code architecture, ...
  - Python + (NumPy, f2py, SymPy)
  - Horton, PySCF
  - Psi4
- Meta-programming : generate low-level code with a higher-level language
  - `FFTW`: C generated by an OCaml program
  - `libcint`: C generated by a Common Lisp program

Problem addressed here

Make code in the low-level language easy to write and maintain

OUTLINE

Programming with Implicit Reference to Parameters (IRP)

WHAT IS A SCIENTIFIC CODE?

A (scientific) program is a function of its input data:

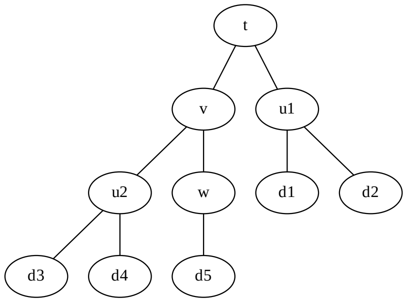
$$\text{output} = \text{program}(\text{input})$$

A program can be represented as a production tree where

- The root is the output
- The leaves are the input data
- The nodes are the intermediate variables
- The edges represent the relation needs/needed by

Example: Production tree of $t\Big(u(d_1, d_2), v\Big(u(d_3, d_4), w(d_5)\Big)\Big)$

$$\begin{aligned} u(x,y) &= x+y+1 \\ v(x,y) &= x+y+2 \\ w(x) &= x+3 \\ t(x,y) &= x+y+4 \end{aligned}$$



```
program compute_t
  implicit none
  integer :: d1, d2, d3, d4 d5
  integer :: u, v, w, t

  call read_data(d1,d2,d3,d4,d5)       !           t
                                       !         /   \
  call compute_u(d3,d4,u)              !       u       v
  call compute_w(d5,w)                 !     /  |     / \
  call compute_v(u,w,v)                !  d1   d2    u    w
  call compute_u(d1,d2,u)              !           /  \    \
  call compute_t(u,v,t)                !          d3   d4   d5

  write(*,*), "t=", t
end program
```

Imperative programming (wikipedia)

[...] programming paradigm that uses statements that change a program's state.

- The code expresses the exploration of the production tree
- The routines have to be called in the correct order
- The values of variables are time-dependent

```
program compute_t
  implicit none
  integer :: d1, d2, d3, d4 d5
  integer :: u, v, w, t

  call read_data(d1,d2,d3,d4,d5)       !           t
                                       !        /     \
  call compute_u(d3,d4, u )            !     u          v
  call compute_w(d5,w)                 !   /              \
  call compute_v( u ,w,v)              ! d1    d2     u      w
  call compute_u(d1,d2, u )            !            /    \    \
  call compute_t( u ,v,t)              !          d3     d4    d5

  write(*,*), "t=", t
end program
```

Sources of complexity

1. Time-dependence of the data (*mutable data*)
2. Handling the complexity of the production tree

Functional programming (wikipedia)

[...] programming paradigm [...] that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

No time-dependence (*immutable data*) $\Longrightarrow$ reduced complexity

"FUNCTIONAL" IMPLEMENTATION IN FORTRAN

```
program compute_t                         !          t
  implicit none                           !       /     \
  integer :: d1, d2, d3, d4 d5            !     u         v
  integer :: u, v, w, t                   !   /   |      / \
                                          ! d1   d2     u    w
  call read_data(d1,d2,d3,d4,d5)          !           /   \    \
                                          !         d3    d4   d5
  ! Functional starts here
  write(*,*), "t=", t( u(d1,d2), v( u(d3,d4), w(d5) ) )
end program
```

- Instead of telling *what to do*, we express *what we want*
- The programmer doesn't handle the execution sequence

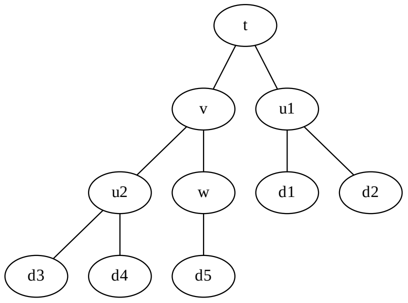
No time-dependence left

Production tree of Ψ in QMC=Chem: 149 nodes / 689 edges

1. The programmers need to have the *global knowledge* of the production tree : Production trees are usually too complex to be handled by humans
2. Programmers may not be sure that their modification did not break some other part
3. Collaborative work is difficult : any programmer can alter the production tree (accidentally or not)

Express the needed entities for each node:

- $t \to u_1$ and v
- $u_1 \to d_1$ and d_2
- $v \to u_2$ and w
- $u_2 \to d_3$ and d_4
- $w \to d_5$

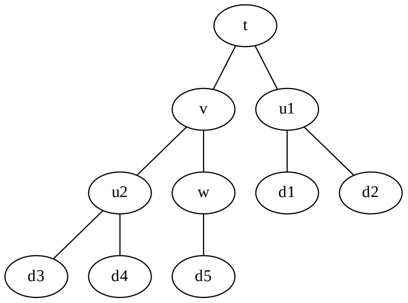


The information is now *local* and easy to handle.

Let us rewrite:

$$t\Big(u(d1, d2), v\Big(u(d3, d4), w(d5)\Big)\Big)$$

$$\begin{aligned} u(x,y) &= x+y+1 \\ v(x,y) &= x+y+2 \\ w(x) &= x+3 \\ t(x,y) &= x+y+4 \end{aligned}$$



```fortran
program compute_t
  integer, external :: t
  write(*,*), "t=", t()
end program

integer function t()
  integer, external :: u1, v
  t = u1() + v() + 4
end

integer function v()
  integer, external :: u2, w
  v = u2() + w() + 2
end

integer function w()
  integer :: d1,d2,d3,d4,d5
  call read_data(d1,d2,d3,d4,d5)
  w = d5+3
end
```

```fortran
integer function f_u(x,y)
  integer, intent(in)  :: x,y
  f_u = x+y+1
end

integer function u1()
  integer :: d1,d2,d3,d4,d5
  integer, external :: f_u
  call read_data(d1,d2,d3,d4,d5)
  u1 = f_u(d1,d2)
end

integer function u2()
  integer :: d1,d2,d3,d4,d5
  integer, external :: f_u
  call read_data(d1,d2,d3,d4,d5)
  u2 = f_u(d3,d4)
end
```

CONSEQUENCES

- The global production tree is not known by the programmer
- The program is easy to write (mechanical)
- Any change of dependencies will be handled properly automatically

CONSEQUENCES

- The global production tree is not known by the programmer
- The program is easy to write (mechanical)
- Any change of dependencies will be handled properly automatically

But: The same data may be recomputed multiple times.

CONSEQUENCES

- The global production tree is not known by the programmer
- The program is easy to write (mechanical)
- Any change of dependencies will be handled properly automatically

But: The same data may be recomputed multiple times.

Simple solution : Lazy evaluation using memo functions.

Entity Node of the production tree

Valid Fully initialized with meaningful values

Builder Subroutine that builds a valid value of an entity from its dependencies

Provider Subroutine with no argument which guarantees to return a valid value of an entity

Rules of IRP[1]

1. Each entity has only one provider
2. Before using an entity, its provider has to be called

[1]François Colonna : ”IRP programming : an efficient way to reduce inter-module coupling ”, DOI: 10.13140/RG.2.1.3833.0406

IRP EXAMPLE

```fortran
program test
    use entities
    implicit none
    call provide_t
    print *,  "t=", t
end program

module entities
  ! Entities
  integer :: u1, u2, v, w, t
  logical :: u1_is_built = .False.
  logical :: u2_is_built = .False.
  logical :: v_is_built  = .False.
  logical :: w_is_built  = .False.
  logical :: t_is_built  = .False.

  ! Leaves
  integer :: d1, d2, d3, d4, d5
  logical :: d_is_built  = .False.
end module
```

```fortran
subroutine provide_t
    use entities
    implicit none
    if (.not.t_is_built) then
        call provide_u1
        call provide_v
        call build_t(u1,v,t)
        t_is_built = .True.
    end if
end subroutine provide_t

subroutine build_t(x,y,result)
    implicit none
    integer, intent(in)  :: x, y
    integer, intent(out) :: result
    result = x + y + 4
end subroutine build_t
```

With the IRP method:

1. Code is easy to develop for a new developer : Adding a new feature only requires to know the *names* of the needed entities
2. If one developer changes the dependence tree, the others will not be affected : collaborative work is simple
3. Forces to write clear code : one builder builds only one thing
4. Forces to write efficient code (spatial and temporal localities are good)

With the IRP method:

1. Code is easy to develop for a new developer : Adding a new feature only requires to know the *names* of the needed entities
2. If one developer changes the dependence tree, the others will not be affected : collaborative work is simple
3. Forces to write clear code : one builder builds only one thing
4. Forces to write efficient code (spatial and temporal localities are good)

But in real life:

1. A lot more typing is required
2. Programmers are lazy

IRPF90

- Extends Fortran with additional keywords
- Fortran code generator (source-to-source compiler)
- Writes all the mechanical IRP code

Useful features:

- Automatic Makefile generation
- Automatic Documentation
- Text editor integration
- Some Introspection
- Meta programming
- Some features targeted for HPC

`http://irpf90.ups-tlse.fr`

`https://gitlab.com/scemama/irpf90`

`https://www.gitbook.com/book/scemama/irpf90`

```
program irp_example
  print *, 't=', t
end

BEGIN_PROVIDER [ integer, t ]
  t = u1+v+4
END_PROVIDER

BEGIN_PROVIDER [ integer,w ]
  w = d5+3
END_PROVIDER

BEGIN_PROVIDER [ integer, v ]
  v = u2+w+2
END_PROVIDER
```

```
BEGIN_PROVIDER [ integer, u1 ]
  integer, external :: fu
  u1 = fu(d1,d2)
END_PROVIDER

BEGIN_PROVIDER [ integer, u2 ]
  integer, external :: fu
  u2 = fu(d3,d4)
END_PROVIDER

integer function fu(x,y)
  integer, intent(in) :: x,y
  fu = x+y+1
end function
```

```
BEGIN_PROVIDER [ double precision, A, (dim1, 3) ]
  ...
END_PROVIDER
```

- Allocation of IRP arrays done automatically
- Dimensioning variables can be IRP entities, provided before the memory allocation
- `FREE` keyword to force to free memory. Invalidates the entity.

```
BEGIN_PROVIDER [ double precision, &
    SCF_density_matrix_ao, (ao_num,ao_num) ]
 implicit none
 BEGIN_DOC
 ! Density matrix in the AO basis, used in the SCF.
 END_DOC
 ...
END_PROVIDER
```

```
$ irpman fock_matrix_mo
```

IRPF90 entities(l) scf_density_matrix_ao IRPF90 entities(l)

Declaration

```
double precision, allocatable :: scf_density_matrix_ao (ao_num,ao_num)
```

Description

Density matrix in the AO basis, used in the SCF.

File

scf_utils/scf_density_matrix_ao.irp.f

Needs

- ao_num
- elec_alpha_num
- elec_beta_num
- scf_density_matrix_ao_alpha
- scf_density_matrix_ao_beta

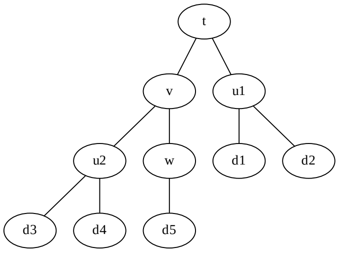
Needed by

- fps_spf_matrix_ao

IRPF90 entities scf_density_matrix_ao IRPF90 entities(l)

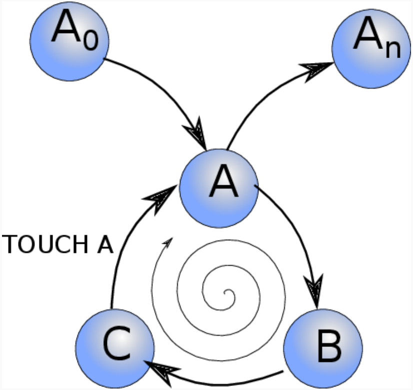
- Start with 3 files : `irp_example1.irp.f, uvwt.irp.f, input.irp.f`
- `irpf90 --init` : Creates `Makefile`
- `make` : Compiles the code and creates `irp_example1, irpf90_entities, tags, IRPF90_man/*, IRPF90_temp/*`.
- `./irp_example1` : Run the program

- `vim Makefile` : Edit the `Makefile` to add the `-d` option
- `make && ./irp_example1` : Run the program with debug on

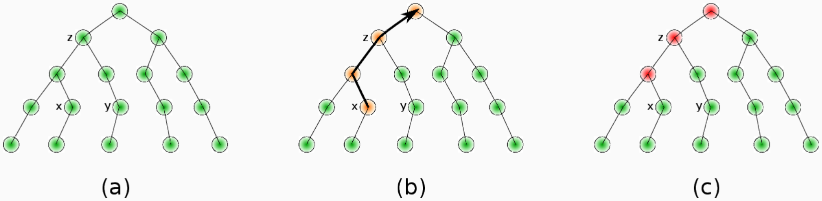


- `irpman t ; irpman fu`
- Multiple executables : Create `irp_example2.irp.f` which prints `t` and `v`
- Integration with Vim : Syntax coloring, `Ctrl-]`, tag, K, `vim -t`

Iterative processes involve cyclic dependencies



`TOUCH A` : `A` is valid, but everything that needs `A` is invalidated.



(a) Everything is valid

(b) *x* is modified

(c) *x* `TOUCH`ed

- Assert keyword, Templates
- Variables can be declared *anywhere*
- +=, −=, ∗= operators
- Dependencies are known by IRPF90 → Makefiles are built *automatically*
- Array alignment, Variable substitution
- Codelet generation
- TSC Profiler
- Thread safety (OpenMP)
- Syntax highlighting in Vim
- Generation of tags to navigate in the code
- No problem using external libraries (MKL, MPI, etc)
- ...

Quantum Package

QUANTUM PACKAGE

IRPF90 library for post-HF quantum chemistry

- Developed at LCPQ (Toulouse) and LCT (Paris)
- Open Source (AGPL), Hosted on GitHub: `https://github.com/QuantumPackage/qp2`
- Goal : Easy for the user *and* the programmer
- Long term objective : Massively parallel post-HF

`https://quantumpackage.github.io/qp2/`

Why another package for quantum chemistry?

Telling a programmer that someone already wrote a routine for this is like telling a songwriter that someone already wrote a love song.

Some guy on twitter...

Perturbatively Selected Configuration Interaction (CIPSI)

- Don't explore the complete CI space, but select determinants on-the-fly (CIPSI) with perturbation theory.
- Target spaces : Full-CI, MR-CISD, large CAS
- Use PT2 to estimate the missing part
- Requires Determinant-driven algorithms

CIPSI Algorithm

1. Start with $|\Psi_0\rangle = |\mathrm{HF}\rangle$

CIPSI Algorithm

1. Start with $|\Psi_0\rangle = |\mathrm{HF}\rangle$
2. $\forall\{|i\rangle\} \notin \Psi_n \,\mathrm{but} \in \{\hat{T}_{\mathrm{SD}}|\Psi_n\rangle\}$, compute $e_i = \frac{\langle i|\mathcal{H}|\Psi_n\rangle^2}{E(\Psi_n)-\langle i|\mathcal{H}|i\rangle}$

CIPSI Algorithm

1. Start with $|\Psi_0\rangle = |\mathrm{HF}\rangle$
2. $\forall \{|i\rangle\} \notin \Psi_n \, \mathrm{but} \in \{\hat{T}_{\mathrm{SD}}|\Psi_n\rangle\}$, compute $e_i = \frac{\langle i|\mathcal{H}|\Psi_n\rangle^2}{E(\Psi_n) - \langle i|\mathcal{H}|i\rangle}$
3. if $|e_i| > \epsilon_n$, select $|i\rangle$

CIPSI Algorithm

1. Start with $|\Psi_0\rangle = |\mathrm{HF}\rangle$
2. $\forall \{|i\rangle\} \notin \Psi_n \, \mathrm{but} \in \{\hat{T}_{\mathrm{SD}}|\Psi_n\rangle\}$, compute $e_i = \frac{\langle i|\mathcal{H}|\Psi_n\rangle^2}{E(\Psi_n)-\langle i|\mathcal{H}|i\rangle}$
3. if $|e_i| > \epsilon_n$, select $|i\rangle$
4. Estimated energy : $E(\Psi_n) + E(PT2)_n = E(\Psi_n) + \sum_i e_i$

CIPSI Algorithm

1. Start with $|\Psi_0\rangle = |\mathrm{HF}\rangle$
2. $\forall \{|i\rangle\} \notin \Psi_n \,\mathrm{but} \in \{\hat{T}_{\mathrm{SD}}|\Psi_n\rangle\}$, compute $e_i = \frac{\langle i|\mathcal{H}|\Psi_n\rangle^2}{E(\Psi_n) - \langle i|\mathcal{H}|i\rangle}$
3. if $|e_i| > \epsilon_n$, select $|i\rangle$
4. Estimated energy : $E(\Psi_n) + E(PT2)_n = E(\Psi_n) + \sum_i e_i$
5. $\Psi_{n+1} = \Psi_n + \sum_{i(\mathrm{selected})} c_i |i\rangle$

CIPSI Algorithm

1. Start with $|\Psi_0\rangle = |\mathrm{HF}\rangle$
2. $\forall\{|i\rangle\} \notin \Psi_n$ but $\in \{\hat{T}_{\mathrm{SD}}|\Psi_n\rangle\}$, compute $e_i = \frac{\langle i|\mathcal{H}|\Psi_n\rangle^2}{E(\Psi_n)-\langle i|\mathcal{H}|i\rangle}$
3. if $|e_i| > \epsilon_n$, select $|i\rangle$
4. Estimated energy : $E(\Psi_n) + E(PT2)_n = E(\Psi_n) + \sum_i e_i$
5. $\Psi_{n+1} = \Psi_n + \sum_{i(\mathrm{selected})} c_i|i\rangle$
6. Minimize $E(\Psi_{n+1})$ (Davidson)

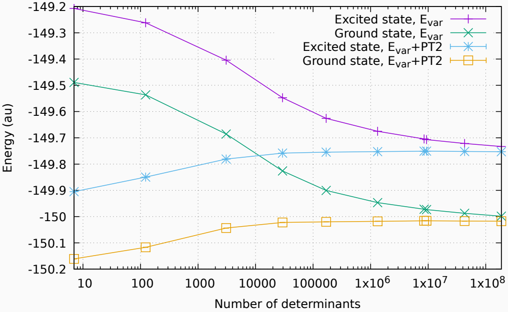
CIPSI Algorithm

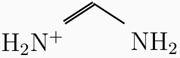
1. Start with $|\Psi_0\rangle = |\mathrm{HF}\rangle$
2. $\forall \{|i\rangle\} \notin \Psi_n \text{ but } \in \{\hat{T}_{\mathrm{SD}}|\Psi_n\rangle\}$, compute $e_i = \frac{\langle i|\mathcal{H}|\Psi_n\rangle^2}{E(\Psi_n) - \langle i|\mathcal{H}|i\rangle}$
3. if $|e_i| > \epsilon_n$, select $|i\rangle$
4. Estimated energy : $E(\Psi_n) + E(PT2)_n = E(\Psi_n) + \sum_i e_i$
5. $\Psi_{n+1} = \Psi_n + \sum_{i(\mathrm{selected})} c_i |i\rangle$
6. Minimize $E(\Psi_{n+1})$ (Davidson)
7. Choose $\epsilon_{n+1} < \epsilon_n$

CIPSI Algorithm

1. Start with $|\Psi_0\rangle = |\mathrm{HF}\rangle$
2. $\forall\{|i\rangle\} \notin \Psi_n$ but $\in \{\hat{T}_{\mathrm{SD}}|\Psi_n\rangle\}$, compute $e_i = \frac{\langle i|\mathcal{H}|\Psi_n\rangle^2}{E(\Psi_n)-\langle i|\mathcal{H}|i\rangle}$
3. if $|e_i| > \epsilon_n$, select $|i\rangle$
4. Estimated energy : $E(\Psi_n) + E(PT2)_n = E(\Psi_n) + \sum_i e_i$
5. $\Psi_{n+1} = \Psi_n + \sum_{i(\mathrm{selected})} c_i|i\rangle$
6. Minimize $E(\Psi_{n+1})$ (Davidson)
7. Choose $\epsilon_{n+1} < \epsilon_n$
8. Go to step 2

- When $n \to \infty$, $E(PT2)_{n=\infty} = 0$, so the complete CI problem is solved.
- Every CI problem can be solved by iterative perturbative selection





aug-cc-pVDZ

QUANTUM PACKAGE DEMO

JCTC Journal of Chemical Theory and Computation

Article

pubs.acs.org/JCTC

Taming the First-Row Diatomics: A Full Configuration Interaction Quantum Monte Carlo Study

Deidre Cleland, George H. Booth, Catherine Overy, and Ali Alavi*

Department of Chemistry, University of Cambridge, Lensfield Road, Cambridge CB2 1EW, U.K.

ABSTRACT: The initiator full configuration interaction quantum Monte Carlo (*i*-FCIQMC) method has recently been developed as a highly accurate stochastic electronic structure technique. It has been shown to calculate the exact basis-set ground state energy of small molecules, to within modest stochastic error bars, using tractable computational cost. Here, we use this technique to elucidate an often troublesome series of first-row diatomics consisting of Be_2, C_2, CN, CO, N_2, NO, O_2, and F_2. Using *i*-FCIQMC, the dissociation energies of these molecules are obtained almost entirely to within chemical accuracy of experimental results. Furthermore, the *i*-FCIQMC calculations are performed in a relatively black-box manner, without any a priori knowledge or specification of the wave function. The size consistency of *i*-FCIQMC is also demonstrated with regards to these diatomics at their more multiconfigurational stretched geometries. The clear and simple *i*-FCIQMC wave functions obtained for these systems are then compared and investigated to demonstrate the dynamic identification of the dominant determinants contributing to significant static correlation. The appearance and nature of such determinants is shown to provide insight into both the *i*-FCIQMC algorithm and the diatomics themselves.

Journal of Chemical Theory and Computation — Article

Table 1. *i*-FCIQMC Energies of the Series of First Row Diatomics and Their Constituent Atoms (Hartree)[a]

system	VDZ	VTZ	VQZ	V(TQ)Z	VQZ+$\Delta E_{F12}^{ccsd(T)}$
Be (^1S)[b]	−14.65182(3)	−14.66244(5)	−14.66568(4)	−14.66803(6)	
C (^3P)	−37.76069(1)	−37.78121(1)	−37.786960(9)	−37.79039(1)	−37.788368(9)
N (^4S)	−54.47858(1)	−54.51491(1)	−54.52506(1)	−54.53115(2)	−54.52802(1)
O (^3P)	−74.91010(3)	−74.97414(3)	−74.99388(3)	−75.00602(4)	−75.00103(3)
F (^2P)	−99.52772(4)	−99.6205(1)	−99.65052(7)	−99.6686(2)	−99.66275(7)
Be_2 ($^1\Sigma_g^+$)[b]	−29.30449(8)	−29.32772(7)	−29.3350(1)	−29.3403(1)	
C_2 ($^1\Sigma_g^+$)[b]	−75.7285(1)	−75.7850(1)	−75.8023(3)	−75.8127(3)	−75.8082(3)
CN ($^2\Sigma^+$)	−92.4933(1)	−92.5698(1)	−92.5938(1)	−92.6081(2)	−92.6028(1)
N_2 ($^1\Sigma_g^+$)	−109.2767(1)	−109.3754(1)	−109.4058(1)	−109.4245(1)	−109.4179(1)
CO ($^1\Sigma^+$)	−113.05564(9)	−113.15639(7)	−113.1887(1)	−113.2080(2)	−113.2016(1)
NO ($^2\Pi$)	−129.59995(8)	−129.7185(1)	−129.7562(2)	−129.7793(2)	−129.7713(2)
O_2 ($^3\Sigma_g^-$)	−149.98781(8)	−150.1305(1)	−150.1750(2)	−150.2027(2)	−150.1934(2)
F_2 ($^1\Sigma_g^+$)	−199.09941(9)	−199.2977(1)	−199.3598(2)	−199.3984(2)	−199.3870(2)

[a]Except when noted, these systems had their core electrons frozen and were calculated at the experimental equilibrium bond lengths given by Huber and Herzberg.[105] The VQZ+$\Delta E_{F12}^{ccsd(T)}$ results refer to the *i*-FCIQMC VQZ energy corrected by a CCSD(T)-F12/B contribution, and V(TQ)Z to the basis set extrapolation given by eq 8. The Be_2 experimental bond length was taken from ref 106. The standard F12 basis sets were not available for Be, and so, the corrected energies were omitted for consistency. [b]All electron calculations use the equivalent cc-pCV*X*Z basis sets.

F_2, cc-pVQZ : -199.3598(2) a.u.

- File `f2.zmt` contains:

  ```
  F
  F  1  1.4119
  ```

- `qp create_ezfio -b cc-pvqz f2.zmt`
- `qp run scf`
- `qp set_frozen_core`
- `qp set determinants n_det_max 400e3`
- `qp run fci`

In the meantime... Let's program a Hartree-Fock!

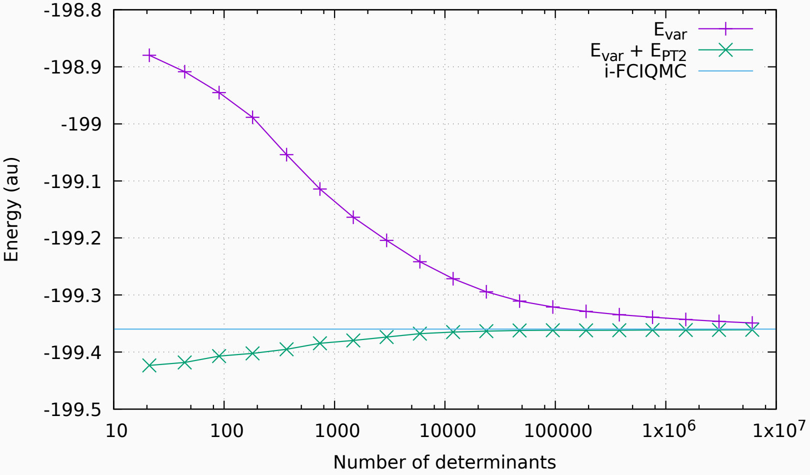
- `qp plugins create -n SimpleHF hartree_fock`
- `qp plugins install SimpleHF`
- `cd plugins/local/SimpleHF ; ninja`
- Test : `h2o.xyz`

  ```
  3
  H2O
  H  0.  0.7572  -0.4692
  H  0. -0.7572  -0.4692
  O  0.  0.        0.1173
  ```
- `qp create_ezfio -b cc-pvdz h2o.xyz`
- `vim SimpleHF.irp.f`
- `qp run SimpleHF`

- vim SimpleHF.irp.f

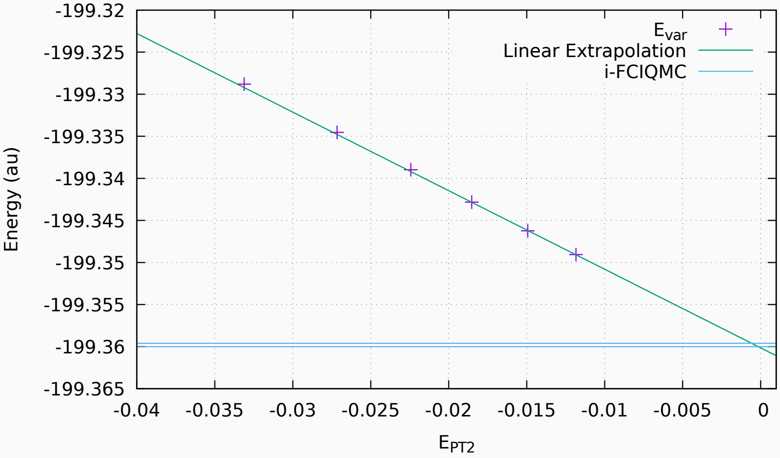
```
program SimpleHF
  implicit none
  BEGIN_DOC
  ! My simple Hartree-Fock program
  END_DOC
  integer :: i
  print *, '------- SCF starts here ------'
  do i=1,30
    print *, i, HF_energy
    mo_coef = eigenvectors_fock_matrix_mo
    TOUCH mo_coef
  end do
  print *, 'Final energy : ', HF_energy
  print *, '------- SCF ends   here ------'
end
```

- Compile with `ninja`
- un with `qp run SimpleHF`



-198.8
-198.9
-199
-199.1
-199.2
-199.3
-199.4
-199.5
Energy (au)
10
100
1000
10000
100000
1x10^6
1x10^7
Number of determinants
E_var
E_var + E_PT2
i-FCIQMC

ENERGY EXTRAPOLATION



IRPF90:

`http://irpf90.ups-tlse.fr`

Quantum Package:

`https://quantumpackage.github.io/qp2`