The CamCASP Programmers Guide

Alston J. Misquitta and Anthony J. Stone
The University Chemical Laboratory, Lensfield Road, Cambridge CB2 1EW
(Dated: September 27, 2019)

Contents

I. INTRODUCTION

The CamCASP program is designed for calculations of intermolecular interaction energies, molecular properties in single centre and distributed forms, model creation using this *ab initio* information, and assessment of the models.

II. BASIC ITEMS

A. Precision

CamCASP uses the F90 precision concept. `module precision` defines four precision types:

- **Real, single precision**: `real(sp)`
- **Real, double precision**: `real(dp)`
- **Short Integer**: `integer(si)`
- **Long Integer**: `integer(li)`

So *do not use `sp`, `dp`, `si` and `li` as variables.* Do not use the `double precision` or any other F77 precision declaration. Here's an example of usage:

```
subroutine example(x)
use precision
implicit none
real(dp), intent(in) :: x
integer(li) :: i
...
return
end subroutine example
```

In much of CamCASP the integers are not specified as long-integers. Bad.

B. Parameters

Parameters for CamCASP are in `module parameters`. Many parameters have the prefix `par_`, but this is not the convention.

C. Run Data

Data items storing variables specific to a run that do not fall into any specific category are stored in `module run_data`. At present, only the titles of the run are stored here.

D. Global Data

Global data are set in `module global_data`. Global variables are all preceeded by `g_`. It is probably best to have a look in this module to see the current settings and defaults. Only the units used will be described here.

1. Units

Internal units used are:

- **Length**: Atomic units.
- **Energy**: Atomic units.
- **Angles**: radians.

Input quantities can be read in a variety of units. These are set globally using the `SET GLOBAL_DATA` section (see *User's Guide*) but many modules allow local units to be defined that override the global settings. Three sets of parameters determine the global units:

- **Length**: `g_length_factor` and `g_length_units`
- **Energy**: `g_energy_factor` and `g_energy_units`
- **Angle**: `g_angle_factor` and `g_angle_units`

Any input variable obtains its internal units by dividing by the factor and an internal variable obtains its output units by multiplying by the factor. Thus, if X is an input length variable,

$$X_{\text{internal}} = X/g\_length\_factor$$

and

$$X_{\text{output}} = X_{\text{internal}} * g\_length\_factor$$

The units of the outputted quantity are stored in the `character(8)` units variable. In the length example above, the output units of X will be `g_length_units`. Defaults are:

- `g_length_factor = 1.0` and `g_length_units = 'BOHR'`
- `g_energy_factor = 1.0` and `g_energy_units = 'HARTREE'`
- `g_angle_factor = 180/`π and `g_angle_units = 'DEGREES'`

These variables can be used to output X correctly and helpfully as follows:

```
print "(a,f8.4,1x,a)","Length is ",X*g_length_factor,trim(g_length_units)
```

which should print something like:

```
Length is  34.3456 BOHR
```

This makes the output unambiguous and as the programmer, your life is easy as all you have to keep in mind is the conversion convention to go from internal to external units and the type of units to use. Maybe, if types were used, this could be done automatically, but not yet.

III. FILE FORMATS

The CamCASP package uses a few binary files (Fortran unformatted sequential files) to transfer data between programs. The format of these files is given here in case they are needed for other purposes.

A. vectA.data

The `vectA.data` and `vectB.data` files are used to transfer molecular orbital eigenvalues and eigenvectors from the *ab initio* program to CAMCASP. If there are N basis functions, the format is

- Record 1: N orbital eigenvalues in ascending order (N `real*8` values in a.u.)
- Record 2: N eigenvectors, each described by N orbital coefficients, in the same order as the corresponding eigenvalues. Note that the whole set of eigenvectors is provided as a single record.

It is for the user to ensure that the basis set used in CamCASPis the same as the basis set used to generate the eigenvectors. If the *ab initio* program deletes some basis functions due to near linear dependence, there will be fewer than N eigenvalues and eigenvectors. In this case the set of eigenvalues and eigenvectors must be made up to N with the necessary number of dummy eigenvalues (with some arbitrary large positive value) and dummy eigenvectors (zero vectors).

B. h1A.data

The `h1A.data` and `h1B.data` files contain the Hessian matrices for the respective molecules.

C. Polarization response files

CamCASP produces polarization response files (suffix `.p2p`) for use by the Pfit program in the WSM procedure for refining the local polarizabilities. The format is

- Record 1: number of points, n. (Integer*4)
- Next n records: point positions as cartesian coordinates in a.u. ($3 \times$ real*8)
- Next $n(n+1)/2$ records: response value matrix in triangle format, i.e. in the order (1,1), (2,1), (2,2), (3,1), (3,2), (3,3), ... Values in a.u., one record for each element, real*8.

IV. MODULE COMMON_ROUTINES

Many useful subroutines are included in this module. Here's a brief description of each of these:

- **Error Checking**
  - `check_info`: This subroutine should be used after every call to a *high-level* subroutine that returns an `info` value. Its default behaviour is to stop on error (negative value of `info`) but this can be prevented by an optional `do_i_stop=.false.` argument. In the latter case, it reports the subroutine and line number at which the error was logged and then returns. This behaviour is preferred in low-level subroutines as it enables the error messages and locations to be passed to the high-level routines and printed out so debugging is possible. Here's an example:

```
subroutine high_level
use precision
use common_routines
implicit none
character(len=*), parameter :: this_routine='high_level'
...
call low_level(...,info)
call check_info(info,'low_level',this_routine,__LINE__,do_i_stop=.true.)
...
end subroutine high_level
```

And the low-level subroutine would call `check_info` as follows:

```
subroutine low_level(...,info)
use precision
use common_routines
implicit none
character(len=*), parameter :: this_routine='low_level'
logical :: myinfo
...
call matmult_types(A,B,C,...,myinfo,...)
call check_info(myinfo,'low_level',this_routine,__LINE__,&
   & do_i_stop=.false.)
if (myinfo.lt.0) then; info=-1; return; endif
...
end subroutine low_level
```

Now, if there is an error in `matmult_types`, the `info` returned is negative. This is spotted by the call to `check_info` which prints out the routine name and line number at which the error was flagged, but doesn't stop. Instead, the `if (myinfo.lt.0)...` statement sets `info` to an appropriate negative value and returns to the high-level subroutine where `check_info` prints out the final location and info value and causes CamCASP to print out a lot of information and finally stop.

This means that the user gets the full "lightening bolt" of error info values, at which lines the errors happened, and the names of the subroutines involved. So now, debugging is probably a lot easier! Here's an example of a real error in CamCASP:

```
matrix_write_types: ERROR
real_matrix A is not in memory
Matrix name is : H21 A   = H2 * H1
ERROR code            -1  from routine matrix_write_types
which was called by routine matmult_types
at line number          1656
ERROR code            -2  from routine matmult_types
which was called by routine init_prop
at line number           783
INTERNAL ERROR: in subroutine check_info
```

```
at line number          136
INFO flagged error
Contact program developers with this problem.

Closing open files
...
list of open files, their names, and contents follows...
...
```

This tells me that an attempt was made to write to file the matrix containing the product of two Hessians but this matrix was not in memory. The error happened in `matrix_write_types` which was called by `matmult_types` at line 1656 which was called by `init_prop` at line 783, and finally, `init_prop` called `check_info` with `do_i_stop=.true.` which initiated the stop-sequence in CamCASP.

The formatting could be improved, but it's still better than a `FORTRAN STOP`.

  - `check_allocate`
  - `check_deallocate`
  - `check_a_file`

- **Utilities**
  - `getunit`
  - `my_timer`
  - `get_available_file`
  - `name_tmp_file`
  - `name_mol_file`
- **Error Reporting**
  - `internal_error`
  - `general_error`
  - `close_file_units`

V. COMMON ERRORS

A. Seg Faults with types

These frequently happen because the contents of the type are accessed without the type being in memory. So here's a common situation:

```
type(real_matrix) :: A
...
...
call create_type(A,mat_name,rowsA,colsA)
...
call release_type(A)
...
...
A%matrix = 1.0_dp
...
```

This will result in a seg fault as `A%matrix` has been released before it is accessed. The correct code would be:

```
type(real_matrix) :: A
...
...
call create_type(A,mat_name,rowsA,colsA)
```

```
...
call release_type(A)
...
...
if (.not.A%in_memory) call reclaim_type(A)
A%matrix = 1.0_dp
...
```

A more devious case is:

```
type(real_matrix) :: A, B, C
...
...
call create_type(A,mat_name,rowsA,colsA)
...
call matmult_types(A,B,C,...,release=.false.)
...
...
A%matrix = 1.0_dp
...
```

So you think that the `release=.false.` option in `matmult_types` will keep all types in memory on exit. This is true only when the large-memory route is chosen in `matmult_types`. In the small-memory route, all types are released on exit irrespective of the value of `release`. So all works well for small jobs (where the large-memory route is used) and then for large jobs you find a seg fault. Once again, the correct code is:

```
type(real_matrix) :: A, B, C
...
...
call create_type(A,mat_name,rowsA,colsA)
...
call matmult_types(A,B,C,...,release=.false.)
...
...
if (.not.A%in_memory) call reclaim_type(A)
A%matrix = 1.0_dp
...
```

B. Small and large memory routes gives different results

This can happen when types are used. Of course, the error could be in the lower-level routines you've called, but as many of these routines are quite mature, it is probably more likely that the error is in a high-level routine that uses the low-level ones.

The annoying thing is that all seems well when the large-memory route is used but things go wrong for large jobs when the memory declared is insufficient and matrices need to be only partially in memory. This could make debugging a pain, as large matrices cannot be generally seen and intermediates printed out to enable debugging. There is a nice way around this: Use a small job (really small) and set `MEMORY 10 BYTES` in the preamble. This will force the small-memory routes for this really small job for which all matrices might be printable to screen.

1. $B = A * A^T$

Tried to calculate $B = A * A^T$ with `matmult_types` and get an error. This would usually involve a call like:

```
call matmult_types(A,A,B,...,'N','T',...)
```

which would work correctly in the large-memory route but fail in the small-memory route as in that case, `matmult_types` regards the two instances of A as distinct objects and reclaims different portions of the two instances - which is impossible as they are the same object. So errors occur. The correct way would be (if A was small):

```
use type_operations
type(real_matrix) :: myA
...
...
myA = A  !using type equality defined in module type_operations
call matmult_types(A,myA,B,...,'N','T',...)
call check_info(...)
call destroy_type(myA)
...
```

Now there won't be a problem.

2. First fine, rest wrong...

Another error occurs in the following scenario:

```
call create_type(A,...)
do k = 1, N

  A%matrix = f(k,:,:)  !values in A change as loop progresses
  call matmult_types(A,B,C,...)

enddo
```

Once again, the large-memory route will be OK. But in the small memory route you will find that all is well for `k=1`, but the value of C is wrong for larger `k`. This is because, for `k=1`, `A%onfile = .false.`, so `matmult_types` writes `A` to file in the small-memory route and sets `A%onfile = .true.`. All is well so far. Now, for `k=2`, `A%matrix` has been updated but not written to file *and* it has `A%onfile = .true.`. Now, `matmult_types`, seeing that `A%onfile = .true.`, doesn't bother writing `A` to file (why waste the effort), only the contents of `A` in memory and on file are different. So you get the wrong results.

There are two solutions to this problem. One is that you explicitely write `A` to file:

```
call create_type(A,...)
do k = 1, N

  A%matrix = f(k,:,:)  !values in A change as loop progresses
  call matrix_write_types(A,'N',...)
  call check_info(...)
  call matmult_types(A,B,C,...)

enddo
```

but this is wasteful if the large-memory route is used as the writing to file is then unnecessary and it is done in a loop. A much more elegant solution is:

```
call create_type(A,...)
do k = 1, N

  A%matrix = f(k,:,:)  !values in A change as loop progresses
  A%onfile = .false.
  call matmult_types(A,B,C,...)

enddo
```

Now, `A` will be written to file by `matmult_types` only in the small-memory route as it sees that `A%onfile = .false.`. So all will be well.

C. Assignment and Types

In module `type_operations`, most of the user-defined types contained in module `types` have their respective assignment operations defined and overloaded onto the assignment operator `=`. For this to work correctly, subroutines must use `type_operations`. This is easy to forget and errors then result. These are generally hard to spot as the code looks exactly as it should — apart from the missing "`use type_operations`" statement. An example:

```
subroutine name(A)
use types
type (basis_set), intent(inout) :: A
type (basis_set) :: B
...
A = B    !wrong type assignment
...
```

This may not work as planned as `use type_operations` is missing. The correct version is:

```
subroutine name(A)
use types
use type_operations  !<---***------------------
type (basis_set), intent(inout) :: A          |
type (basis_set) :: B                         |
...                                           |
A = B    !correct type assignment because of --
...
```

The compiler won't flag as error as it tries some sort of assignment operation on the types involved. This is generally not quite what you want.

NOTE: I've really seen this error occur for `type (real_matrix)` objects only. Perhaps this is because a naive equality of two such types will almost never result in the required behaviour as the `filename` field would be the same and the type may not be in memory thus causing nothing to be put into the assigned type.

D. `intent(inout)` versus `intent(out)`

The `intent()` command is a wonderful one for debugging, but it can cause problems that are very hard to spot if used incorrectly with pointers or user-defined types that involve pointers. The latter would include types `real_matrix`, `real_vector`.

Consider the following example:

```
subroutine sub1
use types
implicit none
type(real_matrix) :: A
...
! define A to be of size 10 by 100
call create_type(A,'Name of A',10,100)
...
! call a subroutine to fill A
call sub2(A)

return
end subroutine sub1

subroutine sub2(A)
use types
implicit none
type(real_matrix), intent(out) :: A
!
do i = 1, A%rows
```

```
  do j = 1, A%cols
    A%matrix(i,j) = float(i+j)
  enddo
enddo
!
return
end subroutine sub2
```

This will fail, probably with a run-time error, or seg-fault. The problem is the `intent(out)` qualifier in `subroutine sub2`. If `A` was a normal variable (not a pointer or a type containing one, as it is here), this would be OK. But what the `intent(out)` does to a pointer argument is to nullify it — after all, this is a quantity that is to be OUTPUT only, so initialize (for some compilers) on input! The bad part is that only some compilers will initialize it, and only these will cause the problem.

To avoid such a problem, always use either `intent(in)` or `intent(inout)` for arguments involving pointers or derived types that use pointers.