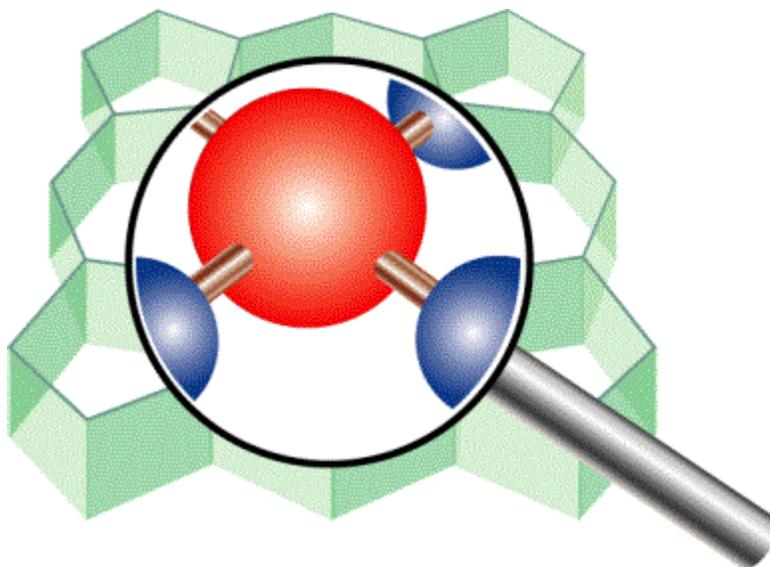# Utility Programs

ADF Program System
Release 2010

Scientific Computing & Modelling NV
Vrije Universiteit, Theoretical Chemistry
De Boelelaan 1083; 1081 HV Amsterdam; The Netherlands
E-mail: support@scm.com

Table of Contents

Introduction

The ADF package contains two main programs ADF and BAND.

Also included are programs to calculate properties, and support programs to facilitate scripting, handle files and prepare relativistic calculations (Dirac). This manual describes the usage of these utility programs.

The utilities discussed in this manual are:

- **adfprep**: prepare an ADF job from a script (or command line).
- **adfreport**: get information (including images) from an ADF result file (for use in your script, or to generate an HTML or tab-separated report).
- **pkf**, **cpkf**, **dmpkf**, **udmpkf**: the KF utilities, which are command-line utilities to process KF files.
- **dirac**: a program to generate relativistic potentials of the isolated atom, and of its frozen core, if present.
  These are used in an ADF calculation when relativistic options are applied.

Using the adfprep and adfreport utilities you can easily create scripts. For example you could implement your own geometry optimizer: set up a calculation using adfprep, run it to perform one ADF geometry step, use adfreport to get the calculated gradients, use your own optimizer to generate new coordinates, use adfprep again to generate an ADF job with the new coordinates, and so on.

The KF utilities enable you to examine ADF result files in detail. You can also use them to modify the information in any way that you like. The ADF result files are KF files. A KF file (Keyed-File) is a special type of binary file. They have a keyword driven organization.

adfprep

ADFprepare ($ADFBIN/adfprep) generates a job script from a .adf file (the template). The .adf file can be produced by ADFinput, or you can use one of the default templates included. These default templates are identical to those present in ADFinput.

Two examples are included to give you an idea what you can do with adfprep.

In $ADFHOME/examples/adf/BakersetSP you will find how to use adfprep to run a particular job (a single point calculation in this case) for all molecules in the Baker set. The molecules are simply xyz files and contain no ADF specific information. adfreport is used to collect the resulting bonding energies

In $ADFHOME/examples/adf/ConvergenceTestCH4 you will find how to use adfprep to test convergence of the bonding energy with respect to basis set and integration accuracy. adfreport is used to collect the resulting bonding energies

The most convenient way to see the optoins of adfprep is to run the adfprep command without arguments. You will get output very much alike the following description, but probably more up-to-date.

```
Usage: adfprep -t template.adf [-m molecule.(adf|xyz|mol|t21)]
              [-z charge] [-s spin] [-i integration]
              [-b basis] [-c core] [-r relativity] [-x xcpotential]
              [-e xcenergy] [-j jobname] [-a adffile]

Start with a template, adjust it for this particular job,
and write the resulting job to standard output.
Values specified should match exactly the values as you
would specify using ADFinput, also for menu choices.


TEMPLATE
-t: the .adf file (saved by ADFinput) to be used as template,
        defining the whole job
    All other options override values from this job
    Instead of a .adf file, you may also specify the name of
        one of the standard templates as defined in
    ADFinput: "Single Point", Frequencies, "Geometry Optimization", etc
    Some shortcuts: SP, GO, GOS, FREQ

CHANGES TO TEMPLATE
-m: the molecule to use, element types and coordinates
    This can be taken from anything that ADFinput can import,
        for example .adf, .mol, xyz or .t21 files
-z: charge (real number)
-s: spin (integer), if not zero this implies an unrestricted calculation
-i: integration (integer)
-b: basis type (SZ, DZ, DZP, TZ, TCP, TZ2P, QZ4P)
-c: core type (None, Small, Medium, Large)
-r: relativistic level (None, Scalar, Spin-Orbit), using ZORA
-x: XC potential during SCF, one from the options available in ADFinput:
        LDA,
        GGA:BP, GGA:BLYP, GGA:PW91, GGA:mPW, GGA:PBE, GGA:RPBE,
        GGA:revPBE, GGA:mPBE, GGA:OLYP, GGA:OPBE,
        Model:SAOP, Model:LB94,
        Hartree-Fock,
        Hybrid:B3LYP, Hybrid:B3LYP*, Hybrid:B1LYP, Hybrid:KMLYP,
        Hybrid:O3LYP, Hybrid:X3LYP, Hybrid:BHandH,
```

```
        Hybrid:BHandHLYP, Hybrid:B1PW91, Hybrid:MPW1PW,
        Hybrid:MPW1K, Hybrid:PBE0, Hybrid:OPBE0
-e: XC energy after SCF (Default, LDA+GGA_METAGGA, LDA+GGA+METAGGA+HYBRIDS)
-k: replace any key, the single argument will be broken into:
     the key, the replacement value, and END for a block key
     all separated by spaces. To insert a return, add a |
     When the key is not found, it is added just before the ATOMS key
     The -k key may be repeated, and is applied at the end,
        replacing even earlier changes


OUTPUT
-j: produce a fully runnable job (as the .job files from ADFjobs),
    using the specified jobname the job script produces files like
    jobname.out, jobname.t21 etc. Several job scripts can simply
    be concatenated, the results will be stored in different files
    using th jobname parameter the default is a simple run script
    (the .run file from ADFinput, files are left as they are)
-a: save a .adf file that matches the run script, except for
    the -k arguments (they are listed in the user input field)
    adffile is the name of the adffile, including the .adf extension
    (required)
```

adfreport

ADFreport ($ADFBIN/adfreport) gets information (including images from molecules or fields) from an ADF result file. The result file is the TAPE21 produced by ADF and should have a .t21 extension. If you use the ADF-GUI or adfprep to generate your job scripts this is already the case. The selected information is printed as concisely as possible on standard output. Alternatively, you can write the information to a tab separated file or to an HTML file. If the file does not exist, adfreport will also generate one line with headers to identify the information. Images are generated using the ADF-GUI.

Results on a .t21 file may depend on the ordering of atoms. adfreport will report such results in the input order when you use one of the predefined keys, which is the order that you will normally expect. If you ask for a dump of a particular KF variable you will get the data exactly as it is present in the file, and it is up to you to make sense of the data.

A simple example to get the bonding energy from a file:

```
adfreport job.t21 BondingEnergy
```

or to generate high-quality pictures of some orbitals:

```
adfreport job.t21 HOMO LUMO+1 -v "-grid Fine" -v "-antialias" -v "-bgcolor
#ffffff"
```

The most convenient way to see the optoins of adfreport is to run the adfreport command without arguments. You will get output very much alike the following description, but probably more up-to-date.

```
Usage: adfreport [-i] result.t21 [-o result.html] [[-r] result] [-v adfview]

Successive calls will with the same output file will update the
    information contained in that file.

The following flags may be used:

-i: result.t21 is the name of the input .t21 file with the ADF
    results the -i is optional, if no -i flag found the first
    argument without flag will be used

-o: result.tsv is the name of the tab separated file in which
        the results will be saved
  : result.html is the name of the html formatted file
        (a simple html table) to save the results
  : absent: the output will be written to standard output,
        only the requested result (no headers)

-r: result is a particular result to be reported from the .t21 file.
    The -r flag is optional: all arguments without flags
        will be considered to be result arguments,
    with exception of the first argument if the -i flag is also absent

    Any proper KF variable is allowed
        (section%variable, see the KF utilities documentation)
    After the variable you can specify details, all separated with #
    - range: one or two numbers, separated with
        a :, array counts start at 1
    - format: TclTk format string for one number, like 8.3f or 12.6g
    - %nperline: (% with number) insert new line
        after nperline items (not used for .tsv)
```

For example:
    -r "Geometry%xyz#12.4f##3"
      prints a nicely formatted table of the coordinates
    -r "Geometry%xyz#1:9#12.4f##3"
      similar, for the first two atoms only
    -r "Geometry%xyz#12.4f#1:9"
      the coordinates of the first two atoms, all on one line
    -r "Geometry%xyz#1"
      print just the first coordinate
    -r "Energys%Bond Energy"
      print the bond energy (a scalar)

When a result without % is specified, it must be one of the
    pre-defined keys.
If you use HTML output, the name of the keys are used as table headers.
The keys are not cases-sensitive, and * matches any text.

```
title         : title of the calculation
type          : calculation type (single point,
                  geometry optimization, ...)
weight        : molecular weight
symmetry      : molecular symmetry
natoms        : number of atoms
integration     : integration accuracy
integration-min : minimum integration accuracy
integration-max : maximum integration accuracy
charges       : shorthand for Voronoi, Hirshfeld and
                  Mulliken charges
voronoi       : voronoi deformation charges
hirshfeld     : hirshfeld fragment charges, will only
                  work with atomic fragments
mdc           : all available MDC atom charges
mdc-m         : MDC-M charges
mdc-d         : MDC-D charges
mdc-q         : MDC-Q charges
mulliken      : mulliken charges
nmr           : nmr chemical shifts
nmr-c*        : nmr chemical shifts
nmr-shielding-tensor : nmr shielding tensor
nmr-j-coupling-tensor : nmr j coupling tensor
nmr-k-coupling-tensor : nmr k coupling tensor
nmr-j-coupling-constant : nmr j coupling constant
nmr-k-coupling-constant : nmr k coupling constant
dipolev*      : dipole vector
dipole        : dipole moment (length of dipole vector)
quadrupole    : quadrupole tensor
orbital-info : orbital info (energy, occupation and label),
                  format for energy after #,
                  range after # with HOMO or LUMO, for examples:
                  orbital-info#HOMO, orbital-info#HOMO-1,
orbital-info#HOMO-2:LUMO+2,
                  orbital-info#HOMO#12.8f
orbital-e*    : orbital energies, format and range after # as in
orbital-info
orbital-o*    : orbital occupations, format and range after # as in
orbital-info
orbital-l*    : orbital labels, format and range after # as in
```

```
orbital-info
      atomlabel          : name of atoms with sequence number, starting at 0
      atomlabel-from0    : name of atoms with sequence number, starting at 0
      atomlabel-from1    : name of atoms with sequence number, starting at 1
      geometry,
      geometry-a*,
      geometry-b* : geometry (element type and coordinates), in input order,
                       in angstrom or bohr (default bohr)
      hessian*    : hessian (from GeoOpt%Hessian_CART),
                       in input order, fmt and nperline options after #
      gradients*  : gradients (from GeoOpt%Gradients),
                       in input order, fmt and nperline options after #
      energies    : all avaliable energies (bonding up to xc, with labels)
      bonding*    : total bonding energy
      pauli*      : total pauli repulsion
      steric*     : total steric interaction
      orbital*    : total orbital interaction
      electrostatic*  : electrostatic energy
      kinetic*        : electrostatic energy
      coulomb*        : coulomb (steric +OrbInt) energy
      xc*             : XC energy
      frequencies*    : IR Frequencies, format, nperline and
                        range (n, or n:n, start at 1) after #
      freqint*        : IR Intensities, format, nperline and
                        range (n, or n:n, start at 1) after #
      freqlabel*      : IR Frequencies label (symmetry), format,
                        nperline and range (n, or n:n, start at 1) after #
      normalmode*     : Normal Modes (mass weighted), format,
                        nperline and range(n, or n:n, start at 1) after #
      zeropoint*      : Zero-Point energy
      excitation*     : Excitation energies, format,
                        nperline and range (n, or n:n, start at 1) after #
      oscillatorstrength*: Oscillator strengths for the excitation energies
                           format, nperline and range (n, or n:n, start at 1)
after #
      excitlabel*        : Excitation labels (symmetry), format,
                           nperline and range (n, or n:n, start at 1) after #

   The -r flag (or arguments without flag) may be repeated for multiple
results

-v: command line to pass to adfview (without filenames) to generate an image
    The image will be generated by ADFview, the image will be stored in a
    directory with a name based on the result file, and with extension .jpgs.
    The result file will contain a path to the image file (directly, or in an
IMG tag)
    After the -v the arguments must be listed, with proper quoting.
    Repeat the -v flag for multiple arguments.

    Some shortcuts have been defined (HOMO, HOMO+1, LUMO, Molecule, Density,
Potential)

    Some useful flags include
        -scmgeometry (default 200x200), -bgcolor (default #220000),
        -zoom (default 1.0), -viewplane (default {1 2 5}),
        -antialias (off when not present, especially useful with light
bgcolors),
```

```
        -grid (Coarse when not present,
              Medium when specified, or value after flag if present)


    Examples: Molecule HOMO LUMO
              HOMO-1 LUMO+1 -v "-viewplane {0 0 1}" -v "-grid Fine" -v
"-antialias"
```

KF command line utilities

There are four utility programs for manipulating KF files from the command shell. Two of them convert kf files from binary to ASCII and vice versa. See the *pkf* and *dmpkf* utilities for a description of the ASCII format of a kf file. An ASCII version of a KF file can be useful to inspect its contents in detail.

In the versions of ADF prior to ADF2006, the conversion back and forth between binary and ASCII was necessary when a binary KF file generated on a particular platform was to be used on another platform that is not binary compatible. To do so one had to convert binary file to ASCII, transfer to the other platform and transfer back to binary. Although a bit tedious, it was occasionally the only way to avoid recomputing a TAPE21 result file only because you needed it on another machine.

As of ADF2006 this is no longer necessary. All programs from the package will convert a KF file to the format native to this platform if necessary. In such a case, the original file will be renamed to a file with tilde "~" appended to its name and a message will printed on the standard output.

The KF software (KF= Keyed File) has been developed at the Vrije Universiteit in Amsterdam as a general-purpose package to store data on files and retrieve it again by keyword-driven procedures. For more information about the KF package (usage, implementation) consult the SCM web site (http://www.scm.com) where information about the ADF software is available.

pkf

```
pkf file1 { file2 ... filen }
```

pkf prints a summary of the contents of the kf files file1... filen.

The output should be more or less self-documenting: all variables are listed by name, type (integer, real, character, logical) and size (number of array elements) and they are grouped together in named sections.

To put the results in an ASCII file for later inspection:

```
pkf file > ascii_result
```

Each section on the file contains an index of its variables and their associated values. All data are organized in blocks. Each section may have any number of index blocks and any number of data blocks (this depends simply on the amount of data to be stored in such a block). In addition there is one special section, the SuperIndex, which is an index of all sections on the file.

The output of *pkf* consists of:

- General information about the file (name of the file, internally used unit numbers during processing the file...)
- A summary of the SuperIndex: an index of blocks on the file and the sections they are associated with.
- A summary: total numbers of blocks associated with the different types of blocks.
- For each section a list of its variables with for each variable:
    ◦ Its name.
    ◦ Its length: the amount of space reserved on the file for the variable.
    ◦ Its size ('used'): the amount of data associated with the variable; for reals, integers and logicals:
      the number of such elements; for strings: the number of characters.
    ◦ The (logical) index of the data block it is stored on;
    ◦ Off-set: its position within the data block in which it is stored;
    ◦ Its value (for an array: the value of the first element);

Remark: 'length' and 'size' are usually the same, but not necessarily.

cpkf

```
cpkf file1  file2 {key1 .. keyn}
```

cpkf copies the sections and/or variables key1 .. keyn from file1 to file2.

If a referenced section or variable already exists on file2 it is overwritten, else it is created. Sections and variables which are already present on file2 but which are not referenced in the command are not affected.

If no sections and/or variables are explicitly mentioned at all the copy is carried out for *all* sections and variables on file1.

A side effect of the copy is that any 'holes' that may be present on the first file are not copied to the second file so that they no longer take up any space. The data copied to the second file is rearranged for optimum storage efficiency. 'Holes' may be due to sections and variables that have existed in the past but that have formally been deleted later; one of the KF functionalities is to delete variables or sections. Such activity does not actually rearrange the KF file, but simply deletes the corresponding entries in index tables.

dmpkf

A utility to extract information from a KF file and make it available in ASCII format.

```
dmpkf file {key1 .. keyn}
```

dmpkf prints the sections and/or variables from the file file indicated by key1 .. keyn on standard output. If no sections and/or variables are explicitly mentioned the complete file is printed.

The format to be used for key1 et cetera is:

```
Sec%Var
```

where Var is the variable, which must exist in section Sec. If no Var is mentioned, the complete section Sec is dumped.

By redirecting the result to another file you get an ASCII version of file:

```
dmpkf file > ascii_result
```

The output contains for each printed variable:

- One line with the name of the section it belongs to;
- One line with the name of the variable itself;
- One line with three integers:
    ◦ The amount of space reserved for the variable on the file (this aspect is relevant for the software that handles kf files and is mentioned here only for completeness);
    ◦ The amount of data associated with the variable: for reals, integers, logicals: the number of such elements; for strings: the number of characters;
    ◦ An integer code for the data type of the variable: 1=integer, 2=real, 3=character, 4=logical;
- The values of the variable (on as many lines as necessary): for scalar variables only one value, for arrays as many values as the array contains.

udmpkf

A utility to put information read from standard input into a KF file.

```
udmpkf file
```

udmpkf reads an ASCII file in the format created by *dmpkf* from standard input and creates the KF file file from this data if it does not already exist, otherwise it adds the sections and/or variables in the input file to the existing kf file. If sections or variables on the input file already exist in the target file, that data on the target file is overwritten. Other data on the target file are not affected.

The combination of *dmpkf* and *udmpkf* makes it easy to modify KF files with a normal text editor:

```
dmpkf TAPE21 > t21_ASCII
```

If necessary, edit and modify t21_ASCII, and then

```
udmpkf < t21_ASCII TAPE21_new
```

Note carefully that *dmpkf* and *udmpkf* do NOT have two arguments, but only one each. The ASCII 'argument' is standard input, respectively output, which you may of course redirect to a file.

Dirac: Relativistic Potentials

The auxiliary program DIRAC, which is installed together with ADF, serves to compute relativistic frozen core potentials (and densities), necessary to apply the (scalar) relativistic option in ADF. The database *atomicdata* has a subdirectory *Dirac*, which contains input files for DIRAC for all atoms of the periodic table of elements. The *names* of the input files indicate the frozen core level: *Ag.3d* for instance is the input file for a calculation on a Silver atom with a frozen core up to and including the 3d shell (i.e.: 1s, 2s, 2p, 3s, 3p, and 3d). The frozen core level used in the DIRAC calculation defines the core data computed and should therefore match the frozen core level in the ADF Create run for the atom that it will be used for.

A DIRAC run with the inputs provided in the database involves a fully relativistic calculation on the atom (spin-orbit coupling, double group symmetries). It generates a file TAPE12 with the corresponding core potential and density (a table of values for a sequence of radial distance values). Other files produced by DIRAC should be removed after the DIRAC run; they are needed for other applications of the program but play no role here.

If you run DIRAC while a file TAPE12 already exists the computed core data will be written at the end of it, after the existing data. The program will assume, however, that the existing data on the file are also core-data from DIRAC runs, and may abort otherwise.

Starting from ADF2006.01 it is not necessary anymore to make one big TAPE12 which contains data for all atoms involved in the molecular calculation. Instead only in the ADF Create run for each atom one needs a TAPE12 which contains data for the atom that is created. The corresponding core data is written to the TAPE21 of this atomic fragment. In the molecular ADF run one then should not include the CorePotentials key, such that ADF will read core data on the TAPE21's of the atomic fragments. One can still use the CorePotentials key, but then one should proceed as in previous releases.

In previous releases (ADF2005 and older), if a CorePotentials file was needed for an *adf* calculation with the (scalar) relativistic option, the simplest approach was to subsequently run DIRAC for each of the involved atoms types. This builds up the TAPE12 file for this particular molecule. Then, specify in the ADF input which sections correspond to the distinct atom types. Alternatively, which we do not recommend, if you frequently perform relativistic ADF runs, with many different types of atoms, you might, once and for all, construct one big TAPE12 file, containing the core potentials of all atoms that you may ever need, and use that file again and again. Of course you need then to remember which section numbers correspond to which atoms.

Implied options

The DIRAC calculations imply the *local* Density Functional in its simple X-alpha approximation without any gradient corrections. Not the *scalar* relativistic but the *fully* relativistic Hamiltonian is used, including spin-orbit coupling. In ADF you may use the *scalar* relativistic Hamiltonian and most users will employ a more sophisticated lda than X-alpha, such as the default vwn (Vosko, Wilk, Nusair) formulas, and may in addition routinely apply gradient corrections. The core potential may not exactly match the Fock operator applied in the molecular calculation. The effect is very small and one can neglect the discrepancy.

Input

The ascii input files for DIRAC, as available in the database directory *$ADFHOME/atomicdata/Dirac*, have a structure as described below. With this information you should be able to construct alternative input files, with other frozen cores for instance.

1 Title (60 characters at most). Plays no role

2 Ngrid, Nshell, rmin, rmax, Z, Xion, Anuc
Ngrid=number of radial grid points, in which the core potentials are computed.
Nshell=number of atomic orbital shells
rmin, rmax=minimum and maximum radial grid values
Z=nuclear charge
Xion=net charge of the 'atom'
Anuc=atomic weight

3 Pinit, Pfinal, eps, del, delrv
Pinit, Pfinal= initial and final density iteration averaging factors. Each iteration cycle changes the actual averaging factor by taking the average of the previous and the final one, starting with the 'initial' one.
eps=Exp(-sqrt(eps)) is set to zero, so eps determines the exponential underflow control.
del=absolute convergence criterion for orbital eigenenergies.
delrv=convergence criterion on the potential (multiplied by the radial distance r).

4 Idirc, Nmax, Ndebu, Nprin, Ipun, Ircor, Iwcor
Idirc=zero for non-relativistic, otherwise one.
Nmax=maximum number of iterations allowed to reach convergence.
Ndebu=non-zero for additional output (for debugging purposes mainly)
Nprin=print parameter. Use 2 or larger to get the orbitals printed.
Ipun=punched output is produced if Ipun is non-zero. (out-of-date)
Ircor=number of core orbitals from the fully relativistic run, to be kept frozen in the subsequent (if any) first-order perturbation calculation.
Iwcor=number of core orbitals used to construct the core density and core potential, that are output on TAPE12. So, here you specify the relativistic core.

5 Xalph, Xlatt, Rnuc
Xalph= Exchange parameter in the Xalpha formalism.
Xlatt=Coulomb tail parameter
Rnuc=size of nuclear radius, in bohr. If set to 1.0 or larger, it is recomputed as 0.0000208*Anuc**(1/3)

6 For each orbital shell:
N, L, J, E, Z, D
N, L, J = The usual orbital quantum numbers. J is used only for relativistic runs.
E = Initial estimate of orbital energy, in atomic units.
Z = Number of electrons on the shell
D = Initial estimate of the error in the orbital energy

7 Icorp, Npcl, Demp, Peps
Icorp = If 1 (one): Do a first order perturbation calculation after the fully relativistic run. This option plays no role in the current application for ADF.
Npcl = Maximum number of cycles in the perturbation calculation.
Demp = Damping factor in the perturbation iterations
Peps = Convergence criterion in the perturbation iterations.