



Scientific Computing & Modelling

Utility Programs

ADF Program System

Release 2004.01

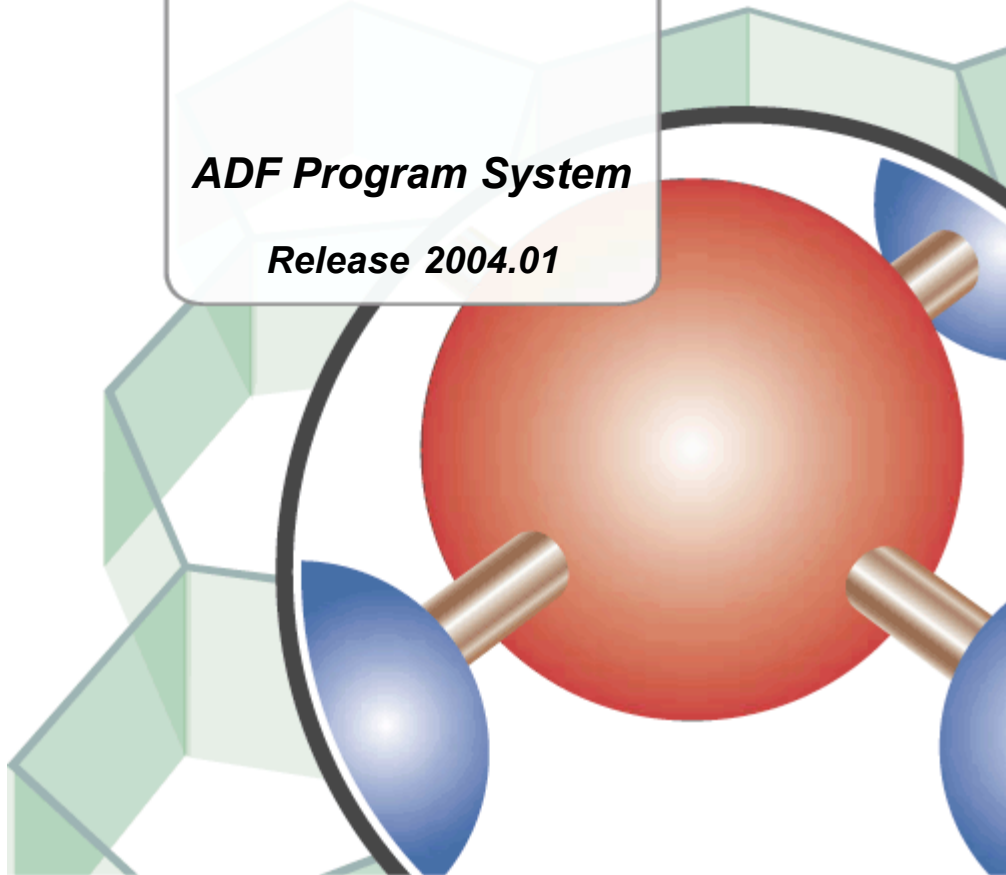


Table of Contents

Utility Programs

Introduction

KF command line utilities

pkf

cpkf

dmpkf

udmpkf

Dirac: Relativistic Potentials

Implied options

Input

Introduction

The ADF package contains two main programs ADF and BAND for calculations on, respectively, molecules and periodic systems and a suite of auxiliary utility and property programs, which are used to prepare special input data or to process results. This manual describes the usage of the utility and other programs. Consult also the *User's Guide*. You should be familiar with the *User's Guide* anyway before reading this *Utilities* document because some concepts from the *User's Guide* will be assumed known here. Among these are the use, format, and general types of keywords applied in the input files as well as in the input for some of the utilities. Most of the utility programs relate only to ADF, in the current version of the package.

The utilities are installed automatically when the ADF package is installed. See the *Installation Manual*.

Some of the files that will be mentioned are KF files. A KF file (Keyed-File) is a special type of Direct-Access Fortran file used in programs of the ADF suite. They have a keyword driven organization and can easily be processed with the KF-utilities that come with the ADF package.

Overview

The utilities discussed in this manual are:

- KF utilities: UNIX 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.
- *bob*: a program which provides a graphical user interface to browse through ADF standard ('print') output files.

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. The conversion back and forth between binary and ASCII may be necessary when a binary KF file generated on a particular platform is to be used on another platform that is not binary compatible. In such case: convert to ASCII, transfer to the other platform and transfer back to binary. Although a bit tedious, it occasionally is the only way to avoid recomputing a TAPE21 result file only because you need it on another machine.

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.

If a CorePotentials file is needed for an *adf* calculation with the (scalar) relativistic option, the simplest approach is 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, 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 X-alpha formalism.

Xlatt=Coulomb tail parameter

Rnuc=size of nuclear radius, in bohr. If set to 1.0 or larger, it is recomputed as $0.0000208 \cdot \text{Anuc}^{1/3}$

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.