Root Rational Fraction program, version 4.1

A calculator program and subroutine library for working with numbers that are square roots of rational fractions

This program was originally written by Anthony Stone and Charles Wood, and is intended primarily for the calculation of Wigner 3j, 6j and 9j symbols, which occur in the theory of angular momentum, and for performing elementary arithmetic with them. Further details of the original Fortran77 implementation may be found in Stone and Wood (1980). The current Fortran95 source code is available at www-stone.ch.cam.ac.uk/pub/rrf-4.1.tar.gz as a compressed tarfile (about 25 kbytes). As well as providing a stand-alone calculator program, it provides a library of Fortran 90 subroutines that manipulate numbers which can be represented as square roots of rational fractions — root-rational-fraction or RRF objects.

Version 4 of the program comprises Fortran modules `rrf_module` and `wigner`, together with a program RRFcalc that reads input a line at a time and treats it as commands for a calculator working in Reverse Polish Notation. It is convenient for interactive use, but it can be used non-interactively by supplying a file of commands as standard input. See §1 for details. The modules can be used to build other programs that need to manipulate Wigner coefficients or other RRF objects.

1 RRF calculator program

The program `rrfcalc` is designed to simplify quantum mechanics calculations involving angular momenta, which usually involve Clebsch–Gordan coefficients or Wigner 3j, 6j or 9j coefficients. The 3j coefficients for low angular momentum values are tabulated in various places but often in inconvenient forms, while 6j tabulations are limited and harder to fins, and the huge number of 9j symbols makes tabulation impractical. Calculation of 6j and 9j coefficients from the standard formulae in terms of 3j coefficients is tedious and error-prone. The `rrfcalc` program provides the values directly and allows straightforward computation of quite complicated formulae to give an exact numerical result.

`rrfcalc` operates in Reverse Polish Notation, which may be unfamiliar but is quite easy to use. It deals with a stack of numbers, initially empty. Each operation either puts a new number on the top of the stack, or operates in some way on the top number, for example by modifying it or displaying it, or combines the top two numbers in some way and replaces them by the result. It is normally used interactively, but can be used non-interactively by supplying a file of commands as standard input.

After each line of input has been processed, the top item on the stack is displayed, unless the last command already displayed it. The commands available are:

integer	read the integer (which must be non-negative) and put it on the stack. To get a negative number on the stack, you need to read in a positive number and use CHS to change the sign.
`PP` *rrf*	read the rest of the line as a root-rational-fraction in power-of-prime form (See Appendix A), and put it on the stack.
`+`	Add: replace the top two elements on the stack by their sum.
`-`	Subtract (top-of-stack from the one below).
`*`	Multiply
`/`	Divide (top-of-stack into the one below)
`POWER` n	raise to power n (n = integer or integer/2).
`SQRT`	= `POWER 1/2`. Note that it is an error to take the square root of a number which already contains half-odd-integer powers.
`CHS`	change the sign of top-of-stack
`SWOP, SWAP`	exchange the top two items on the stack.
`STO` n	Store top-of-stack in memory n ($1 < n < 25$). The item is not removed from the stack.
`RCL` n	Recall from memory n to top-of-stack.
`V`	Verify (display) top-of-stack.
`LIST`	List the entire stack using the current default verification mode. Top-of-stack is shown first.
`VP`	Verify top-of-stack in power-of-prime notation
`VI`	Verify top-of-stack as (a/b)*sqrt(c/d), where a, b, c and d are integers (standard default).
`VF`	Verify top-of-stack as a floating-point number.
`VMP`	Change default verification mode to power-of-prime.
`VMI`	Change default verification mode to (a/b)*sqrt(c/d). This is the usual default.
`VMF`	Change default verification mode to floating-point.
`0J, 3J, 6J, 9J`	Calculate a Wigner $3j$, $6j$ or $9j$ symbol and put it on the stack. See below.
`CG`	Calculate a Clebsch–Gordan coefficient and put it on the stack. See below.
`POP`	Delete the top member from the stack.
`PUSH`	Push the stack down, copying the top member. PUSHing an empty stack inserts zero.
`CLR, CLEAR`	Clear the stack.
`PROMPT` *string*	Set prompt string. The string is output when the program is ready for the next line of commands. The default string is ‘:’. A null string may be set.
`ECHO+`	Reflect each input line before obeying any commands.
`ECHO-`	Do not reflect input lines (default).
`(`	Ignore rest of line.
`?`	Give a summary of the available commands.
`QUIT, Q`	Exit the program.

The full input for the `0J, 3J, 6J, 9J` and `CG` commands takes the form:

`0J` j_1 j_2 j_3

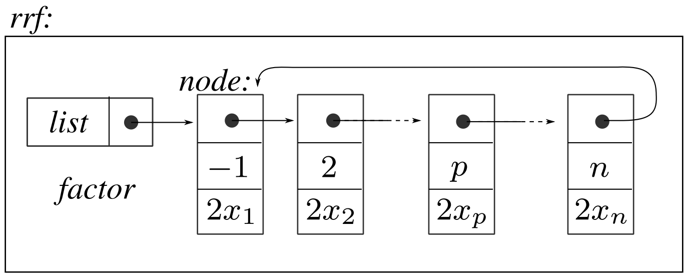

Figure 1: Structure of a root-rational-fraction (RRF) number.

`3J` j_1 j_2 j_3 m_1 m_2 m_3
`6J` j_1 j_2 j_3 l_1 l_2 l_3
`9J` a b c d e f g h i
`CG` j_1 j_2 J m_1 m_2 M
where each argument is integer or integer/2 (integers only for `0J`) and all arguments must appear on the same line separated by one or more spaces. `0J` calculates a $3j$ symbol for which all m values are zero. `CG` calculates the Clebsch–Gordan coefficient $\langle j_1\ j_2\ m_1\ m_2|J\ M\rangle$.

2 RRF routine library

The RRF calculator is suitable for simple one-off calculations, but some calculations need a special program. The RRF routine library is used by the calculator in the background, and is available for general use.

Each RRF (root-rational-fraction) is referred to via a Fortran variable of `type(rrf)`. Such a variable contains a pointer to a `type(node)` variable `list`, which points to the head of a circular list in which the RRF is held in power-of-prime form. (See Figure 1.) Each node in the list describes a factor $(p_n)^{x_n}$, where p_n is a prime number and x_n is an integer or half-odd-integer. The list only includes primes p_n for which the x_n is non-zero. The node actually stores the integer $2x_n$. The head node contains the 'prime' -1, and its integer and half-odd-integer powers provide sign factors ± 1 and $\pm i$. The RRF also contains an integer `factor` which may hold an intermediate multiplying factor that has not yet been factorized into a product of primes. Such intermediate numbers can get extremely large in complicated manipulations, so it is recommended to use the gfortran compiler, which provides 16-byte integers (up to 38 decimal digits). Other compilers, limited to 8-byte integers (18 decimal digits) will however be adequate for normal use. The program checks for integer overflow and should stop rather than give erroneous results.

All arithmetic and i/o manipulations are carried out by special routines provided in the rrf module, and the list elements are not available directly to a program that uses `type(rrf)` variables. Mutiplication of RRFs is straightforward — essentially just a matter of merging the power-of-prime lists, adding the x_n values for each prime p_n. Addition is much more complicated, because each RRF has to be expressed as a power-of-prime list describing the highest common factor of the RRFs in the sum, each multiplied by an integer factor. These integer factors are then added together, factorized into a power-of-prime list, and multiplied by the common factor. When several RRFs are added sequentially, the intermediate factors can get very large, and include very large prime factors, but the final result in angular-momentum calculations contains only relatively small prime factors, usually no larger than the sum of all the angular momenta involved plus 1.

Further details of the program, in its original Fortran 77 form, may be found in Stone and Wood (1980). The current version of the program comprises a Fortran module `rrf_module.F90` containing all the basic routines, a module `wigner.F90` containing routines to calculate 3j, 6j and 9j coefficients, the calculator program `rrfcalc.F90`, and programs `test3j.f90`, `test6j.f90` and `test9j.f90` which carry out various tests of the Wigner routines. Also included are the programs `realcg.F90`, which generates tables of coupling coefficients, analogous to Clebsch–Gordan coefficients, for coupling spherical tensors expressed in real form (see Stone (2013)), and `C6coeffs.F90`, which generates Fortran code for calculating dispersion coefficients (up to C_{12}, not just C_6) from local distributed polarizabilities at imaginary frequency.

3 RRF library routines

3.1 Initialization routine

```
subroutine init_rrf([quiet[,primes]])
```

This must be called before any of the other routines. It sets up a table of primes and initializes the table of factorials. `quiet` is an optional logical argument, which if true suppresses the initial program banner. `primes` is an optional integer argument, which if present specifies the number of primes to be generated in the table of primes. Default 20000.

3.2 Housekeeping routines

```
subroutine clear(k)
```

Deallocate RRF `k`. This should always be used to release the space occupied by temporary variables before exit from subroutines – otherwise the list space will rapidly become full. There is no automatic garbage collection.

```
subroutine copy(k1,k2)
```

Set RRF `k2` equal to `k1`. The Fortran assignment statement `k2 = k1` can also be used; it is overloaded to do the right thing.

```
subroutine rename(k1,k2)
```

Set RRF `k2` equal to `k1` and clear `k1`.

```
subroutine exch(k1,k2)
```

Exchange the RRFs `k1` and `k2`. Use exch or rename rather than copy where possible.

```
subroutine unit(k)
```

Set `k` to the RRF representation of unity (not the same as $k = 1$).

`function iszero(k)`

Returns the value .true. if RRF `k` represents zero and .false. otherwise.

`function nonzero(k)`

Returns the value .true. if `k` represents a nonzero value and .false. otherwise.

`subroutine chsign(k)`

Change the sign of `k` (multiply by -1).

`subroutine conjugate(k)`

Replace `k` by its complex conjugate.

3.3 Arithmetic routines

`subroutine mult(k1,k2,m)`

Multiply `k1` by `k2**m`. `k1` and `k2` must be different variables. Division is achieved by having $m < 0$.

`subroutine multi(k1,i,m)`

Multiply RRF `k1` by `i**m`, i, m integers.

`subroutine setf(n,k)`

Set RRF k to the value of $n!$. The factorial table, which initially contains only 0! and 1!, is extended if necessary.

`subroutine multf(k,n,m)`

Multiply RRF k by $(n!)^m$. The factorial table is extended if necessary.

`subroutine power(k1,m)`

Replace `k1` by `k1**m`.

`subroutine add(k1,k2)`

Add `k2` to `k1`, leaving `k2` unchanged. `k1` and `k2` must be different RRF variables. An additional restriction, which is in practice not a limitation at all, is that the result of the addition must itself be expressible as an RRF. Thus, for example, an attempt to add $\sqrt{3}$ to $\sqrt{5}$ will provoke an error message. See Note 2.

`subroutine subtr(k1,k2)`

Subtract `k2` from `k1`.

`logical function equal(k1,k2)`

The logical function `equal` has the value `.true.` if `k1` and `k2` represent the same number, `.false.` otherwise. The form `k1 == k2` can also be used when `k1` and `k2` are both RRFs, as can `k1 /= k2` to test for inequality.

`subroutine root(k)`

Replace `k` by its square root. This will provoke an error if there are any primes with half-odd-integer exponents in `k` already.

3.4 Conversion routines

These routines convert RRFs to ordinary Fortran variables, or to values in an A1 character buffer, and vice versa, or print out a character representation of an RRF.

```
subroutine int_to_rrf(i,k)
```

Express integer I as RRF K.

```
subroutine pp_to_rrf(m,n, np, k)
```

Read an RRF from the first N characters of the A1 character buffer M. The detailed syntax of the RRF representation is given in Appendix A..

```
subroutine rrf_to_4i(k, i1,i2,i3,i4)
```

Express RRF k as (`i1`/`i2`)sqrt(`i3`/`i4`). `i1` and `i3` may be negative. `i2` and `i4` are always positive, unless integer overflow occurs, in which case zero is returned in `i2`, and `i1`, `i3` and `i4` contain rubbish.

```
subroutine char4i(k, m,m1,m2)
```

Convert the rrf K to 4-integer form as characters in the buffer M of length M2, starting at position `m1`. `m1` is updated to point at the next available position in the buffer. If integer overflow occurs, 12 asterisks are returned.

```
subroutine rrf_to_real(k,a,n)
```

Express RRF `k` as `a*i**n`, where `a` is double precision and the integer `n` has the value 0 or 1 according as `k` is real or imaginary.

```
subroutine rrf_to_char(k, m, m1,max, np)
```

Express RRF `k` in A1 character form in the buffer `m(max)`, starting at position `m1`. The sign term is output as `+`, `-`, `+i` or `-i`; the exponents of the first `NP` primes follow, and further terms appear as '. *primeêxp*'. `NP` may be zero. A value of zero, +1 or -1 is output as `0`, `+1` or `-1` in the sign position. An error message is printed if there is not enough space in the buffer. `m1` is updated to point at the next available position in the buffer.

3.5 Angular momentum functions (module wigner)

```
subroutine threej(j1,j2,j3, m1,m2,m3, x, k)
subroutine sixj(j1,j2,j3, l1,l2,l3, x, k)
subroutine ninej(a,b,c, d,e,f, g,h,i, x, k)
```

These routines set the appropriate vector coupling coefficient in the RRF `k`. `x` is a Fortran integer which should have the value 1 or 2; every angular momentum argument (`j1` etc.) is interpreted as j_1 or $j_1/2$ according as x = 1 or 2.

```
subroutine three0(j1,j2,j3, k)
```

A more efficient equivalent of `threej(j1,j2,j3, 0,0,0, 1, k)`.

`subroutine regge3(jp1,jp2,jp3, jm1,jm2,jm3, k)`

An alternative way to get a 3j coefficient. `jp1` $= j_1 + m_1$, `jm1` $= j_1 - m_1$, etc., so all arguments are integers even if the quantum numbers are not. Since the 3-j routine needs $j1 + m1$ etc. anyway, this is actually a more efficient way to define the coefficient required.

`subroutine CG(j1,j2,J, m1,m2,M, x, k)`

Set the RRF `k` to the Clebsch–Gordan coefficient $\langle j_1 j_2 m_1 m_2 | JM \rangle$.

3.6 Error handling

There is a variable `hard_errors` in `rrf_module` which is initially set true. With this setting most errors cause the program to stop after printing an error message. If it is set false, errors are soft, and cause the variable `error` in `rrf_module` to be set true after printing the error message. If another error occurs while `error` is true, the program will treat it as a hard error even if the variable `hard_errors` is false. If `error` is set false after an error, the program will continue, but results should be treated with suspicion.

3.7 Notes

- The program should not be used unless exact results are essential, as is it inevitably much slower than ordinary arithmetic. In particular, addition may be very slow indeed — much slower than multiplication.

- The embargo on adding numbers if the result cannot be expressed as an RRF will seem a very severe restriction. In fact, the program has been used for several fairly elaborate calculations, involving heavy use of 3j, 6j and 9j coefficients, without ever coming up against the restriction. I would be interested to learn of a serious problem in which it is an important restriction. It could be removed in principle, but would involve a complete rewrite of the program.

- Machine-dependent values are set in `rrf_module.F90`, and should be checked before compiling the program.

- The angular momentum routines all use factorials. The largest available value is 200!, which should suffice for most purposes. An error message is printed if this limit is exceeded, in which case it is necessary to recompile with a larger value for the parameter `MAXFCT` in `rrf_module.F90`.

- The program has been fairly thoroughly tested and has been used successfully on a number of problems, but some bugs doubtless remain. I should appreciate being told about them.

A Power of Prime input and output

Numbers to be read in power-of-prime form must be written as
sign $x(2)$ $x(3)$ $x(5)$ $\ldots$ $x(P_{\mathrm{NP}})$ [*prime*[`^`*exp*] | `n!`[`^`*exp*]] . . .

The sign factor must be present (even for positive numbers). It is followed by exponents for the first `NP` primes; `NP` is specified as an argument to the `PP_TO_RRF` routine, and may be zero. The remaining elements of this expression are optional, as indicated by the square brackets (which should not themselves appear). Each element represents a factor in the required number; that is, the elements are multiplied together to arrive at the result. Numbers output in power-of-prime form appear in the same format. Examples are given below.

sign is either 0 (denoting a zero value) or `+`, `[+]1`, `[+]i`, `-[1]`, or `-i`. The sign term of a positive number may not be omitted.

prime`^`*exp* represents a factor comprising the specified prime number *prime* raised to the power *exp*. E.g. `5^2` represents the factor 25. The exponent may be omitted if it is 1.

`n!^`*exp* is read as *n*! raised to the exponent given. The exponent may be omitted if it is 1.

Prime and factorial terms may occur in any order and may be interspersed with each other. Neither may contain embedded blanks, but they are separated by blanks or period, '.'. The' `^`' introducing an exponent may be omitted if the first character of the exponent is `+` or `-`. Exponents have the form `[+|-]nnn..[/2]`.

A.1 Examples

Using `NP = 0`, the line
`+ 2^-1/2 . 3^1/2 . 5+1/2 . 7-1/2 . 4!`
would be read as $24\sqrt{15/14}$. Using `NP = 4` the same number could be input as
`+ -1/2 1/2 1/2 -1/2 4!`
or as
`+ 5/2 3/2 1/2 -1/2`

References

Stone, A. J. (2013) *The Theory of Intermolecular Forces*, Oxford University Press, Oxford, 2nd edn.

Stone, A. J. and Wood, C. P. (1980) 'Root-rational-fraction package for exact calculation of vector coupling coefficients,' *Computer Phys. Comm.* **21**, 213.