Universität Bremen, Fachbereich Physik / Elektrotechnik

SchroedingerSolver user's guide

final project as part of the lecture

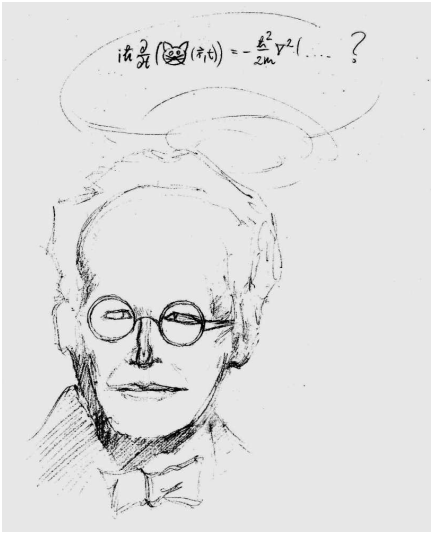
"scientific programming"

authors:
Alexander Erlich (alexander.erlich@gmail.com)
Andreas Krut (andreas.krut@gmx.de)

supervisor:
Dr. Bálint Aradi

project's launchpad website:
`https://launchpad.net/schroedingersolver`

also available on:
`www.airlich.de`



Bremen, September 2009

What SchroedingerSolver does...

SchroedingerSolver computes numerical solutions of SCHRÖDINGER's equation for the stationary case (i.e., no time dependency). You can feed an arbitrary one-dimensional potential into the solver, along with information about the observed interval and discretization. The program interpolates the potential and solves Schrödinger's equation numerically in order to obtain an arbitrary number of wave functions, as well as their corresponding energy levels. All of the results are broken up in output files which can easily be displayed graphically. Additionally, a Matlab routine is provided for the purpose of obtaining a neat plot of the results. The program SchroedingerSolver is written entirely in Fortran and uses several LAPACK routines.

Prequisites

In order to compile SchroedingerSolver, you will require a FORTRAN compiler. LAPACK and BLAS are also required, as some of their routines are called from within the program. The program was written on Ubuntu 8.04 LTS ("Hardy Heron") using the gfortran compiler. In order to install the necessary packages from the Ubuntu repositories, repeat the following steps:

1. `sudo apt-get install gfortran` will take care of the gfortran compiler
2. `sudo apt-get install liblapack-dev` is a package which is essential for the linking process. Installing it will take care of all necessary dependencies. Note that liblapack allone won't work, it will install the libraries, but linking will remain impossible.
3. `sudo apt-get install bzr` is optional as you can simpliy compile the files from the project homepage.
4. `sudo apt-get install grace` will install the graphical tool xmgrace which you need for make-testing the program
5. `sudo apt-get install doxygen` for creating source code documentation files (latest stable version for Ubuntu 8.04 is <1.5.6, so no FORTRAN optimization!). Simply navigate to the branch's main folder and call `$ doxygen`. Note: this has not been tested sufficiently: HTML and LaTeX documentation may not contain all of the documentation which is available in the f90 source code files.

How to get the branch

`bzr branch lp:schroedingersolver` will download the main bazaar branch (provided that bazaar has been installed according to step 3).

Compile and execute

Here is a quick overview of what the make file can do. Some more information is available in the input output examples section where you'll find step by step instructions on how to create plots which are shown there.

```
[how to compile]
just type 'make' and have fun!

[make commands]
-clean          removes *.o and *.mod files only
-realclean      calls 'clean', also removes project files *.dat
                and the program 'sglSolver'
-testclean      cleans the *.dat and *.error files in the test-dirs

-load_pottopf1  loading example file pottopf1
-load_pottopf2  loading example file pottopf2
-load_pottopf3  loading example file pottopf3
-load_harmoszi  loading example file harmoszi

-test_pottopf1  testing example file pottopf1
-test_pottopf2  testing example file pottopf2
-test_pottopf3  testing example file pottopf3
-test_harmoszi  testing example file harmoszi

All of the following use xmgrace for plotting (see prequisites).
Unless xmgrace is installed, there will be no plots, but the
program will remain stable.

-test           makes a test for four examples
-test_lite      makes a less accurate, but faster test for the same
                four examples
-solveSGL       executes the program using the current input file
                and plots the results

-plot_discrpot  plots the discretized potential
-plot_wfuncs    plots all eigenfunctions
-plot_ewfuncs   plots all eigenfunctions with the corresponding
                eigenvalue added to the function's y values
```

User input - editing the `schrodinger.inp` file

The user input is done entirely in the `schrodinger.inp` file. Here's the main frame of the file. If the user input is incorrect, the program will stop and refer to the very line where the problem occured.

Example user input file `schrodinger.inp`

```
2.0                          # mass
-2.0   2.0   1999            # xMin xMax nPoint
xMin   xMax  nPoint 1 15     # firstEigVal    lastEigVal
linear                       # interpType
6                            # nInterp
-2.0   0.0
-0.5   0.0
-0.5   -10.0
0.5    -10.0
0.5    0.0
2.0    0.0
```

Some explainations

The user provides an interval [`xMin`, `xMax`] on which a discretization is to be computed. The length of one interval of discrtization (from here on called `delta`) is

```
delta=(xMax - xMin)/(nPoint-1)
```

As already mentioned in the introduction, one of the most important features of the program is to interpolate the potential which is provided by the user (in the shape of the x, y coordinates of `nInterp` interpolation points). The interpolation type `interpType` can either be the string `'linear'` or `'polynomial'` (a string which does not begin with either `linear` or `polynomial` will cause an error and terminate the program).

Another feature of the program is the calculation of the eigenvalues and the eigenvectors of the Hamiltonian. The Hamiltonian can be expressed as a tridiagonal matrix (see [1] for its explicit form and some more mathematical explainations). If the Hamiltonians dimensions are `nPoint` $\times$ `nPoint`, the `LAPACK` routine `dsteqr` which is used for eigenvalue/eigenvector computation will return `nPoint` eigenvalues. Naturally, you wouldn't want all of the eigenvalues and the corresponding eigenfunctions to be saved. Those for saving are determined by `firstEigValue` and `lastEigValue`, first being the smallest.

Some details on discretization

It is not a simple issue how to deal with very close points. If two points are closer to each other than the discretization step width, this makes the interpolation routines inoperable. If a special routine (`pointsTooClose` in `misc.f90`) finds two points to be too close, it moves the left one to the left (by exactly one disc. step width). Then the search for too close points is re-run, starting with the utmost left points, and so on. The idea behind this is that one point can never be moved past another (if it were so, it would signify that the program would have missed two points which are truly too close to each other. An

honest boy always splits from his *left* girlfriend *first* and *then* has himself approached by the *right* girl.)

It's like Newton's pendulum: The momentum is distributed from the right to the left (or vice versa) via the spheres while the spheres never change their order.

Note that `tooClosePoints` is preceded by a bubblesort routine. Bubblesorting causes the interpolation points to be sorted by their x values. But if two points have the same x value, bubblesort will not change their order.

You can enter your points in any order (bubblesort will sort them), but those points which have identical x values *absolutely need to be given in the correct order* (bubblesort will not change their order - that is, among each other).

If `tooClosePoints` needs to move some interpolation points, a warning is displayed (WARNING INTERMOVE). Note that `tooClosePoints` is followed by a routine called `check_intervals`. If `check_intervals` finds the intervals to to be inconsistent (for which *one* of the reasons might be the newton pendulum effect), it stops the program. It also does make a reference to WARNING INTERMOVE, making it a lot easier to trace the reason for program termination (pendulum effect or simply wrong user input?).

Input/output examples

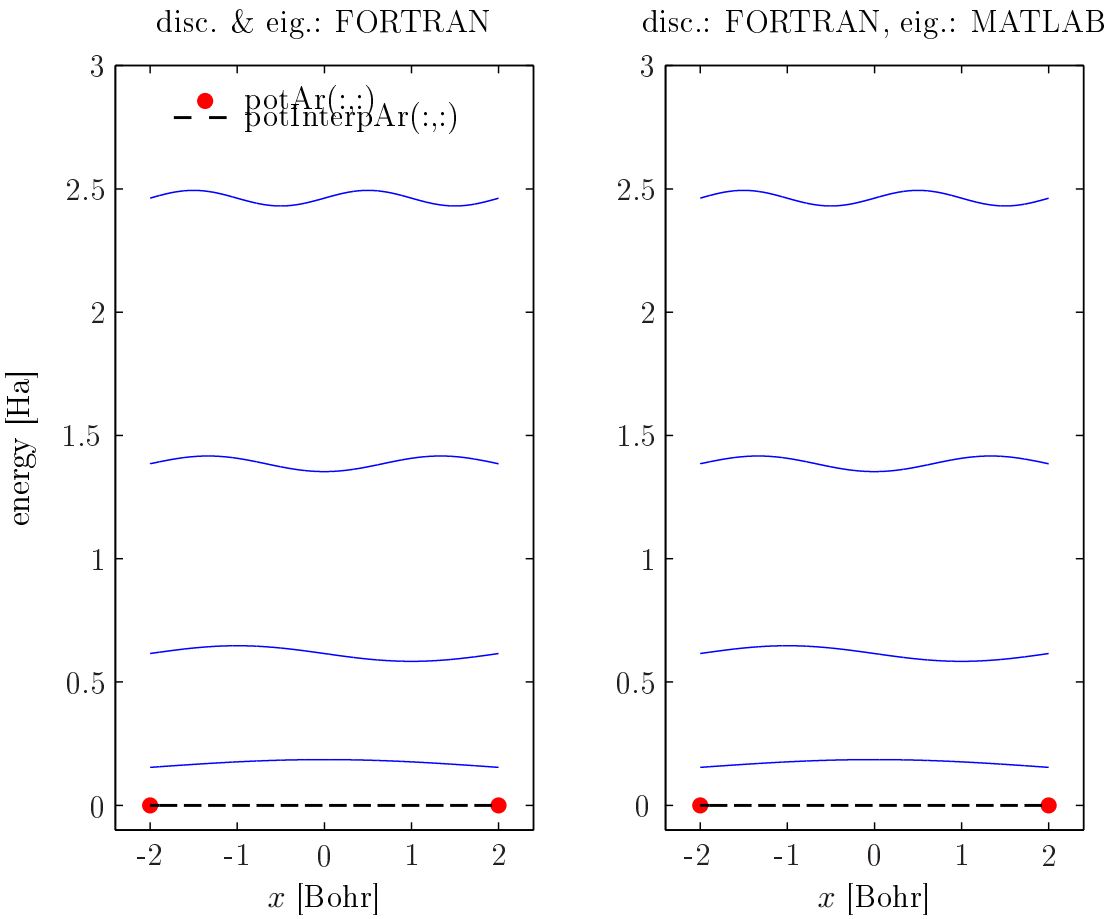
All of the below plots have been created with the `MATLAB` script importfile. But why are there two plots accompanying each example? What `schroedinverSolver` does, in a nutshell, is computing the discretization of the user-given potential and writing it into a file (discrpot.dat), and then use this file in order to compute the eigenvalues/eigenvectors of the $H\Psi = E\Psi$ system, and write those of them into a corresponding file (wfuncs.dat and ewfuncs.dat, respectively).

The first part, writing discrpot.dat, is always done by `schroedingerSolver`. But the second part can rather easily be done by `MATLAB` as well, provided that discrpot.dat and `schrodinger.inp` are available. So the pair of plots which goes with each example shows the very same discretization of the potential (originating in discrpot.dat), but different eigenvalues/eigenvectors, computed by `schroedingerSolver` (left plot) and `MATLAB` (right plot). The `MATLAB` routine importfile which creates the pair of plots was used for debugging purposes in the course of the development.

But if `MATLAB` is not available or not desired (or both), the very same plots can be created using xmgrace very easily. The following commands will do the job: `make plot_discrpot` for for plotting discrpot.dat, `make plot_wfuncs` for the eigenvectors on their own and `make plot_ewfuncs` for the the sum of eigenvalue and corresponding eigenvector.

Example 1: infinite potential well

```
2.0                     # mass
-2.0    2.0     1999    # xMin  xMax    nPoint
1       15              # firstEigVal   lastEigVal
linear                  # interpType
2                       # nInterp
-2.0    0.0             # potAr(:,2)
2.0     0.0
```



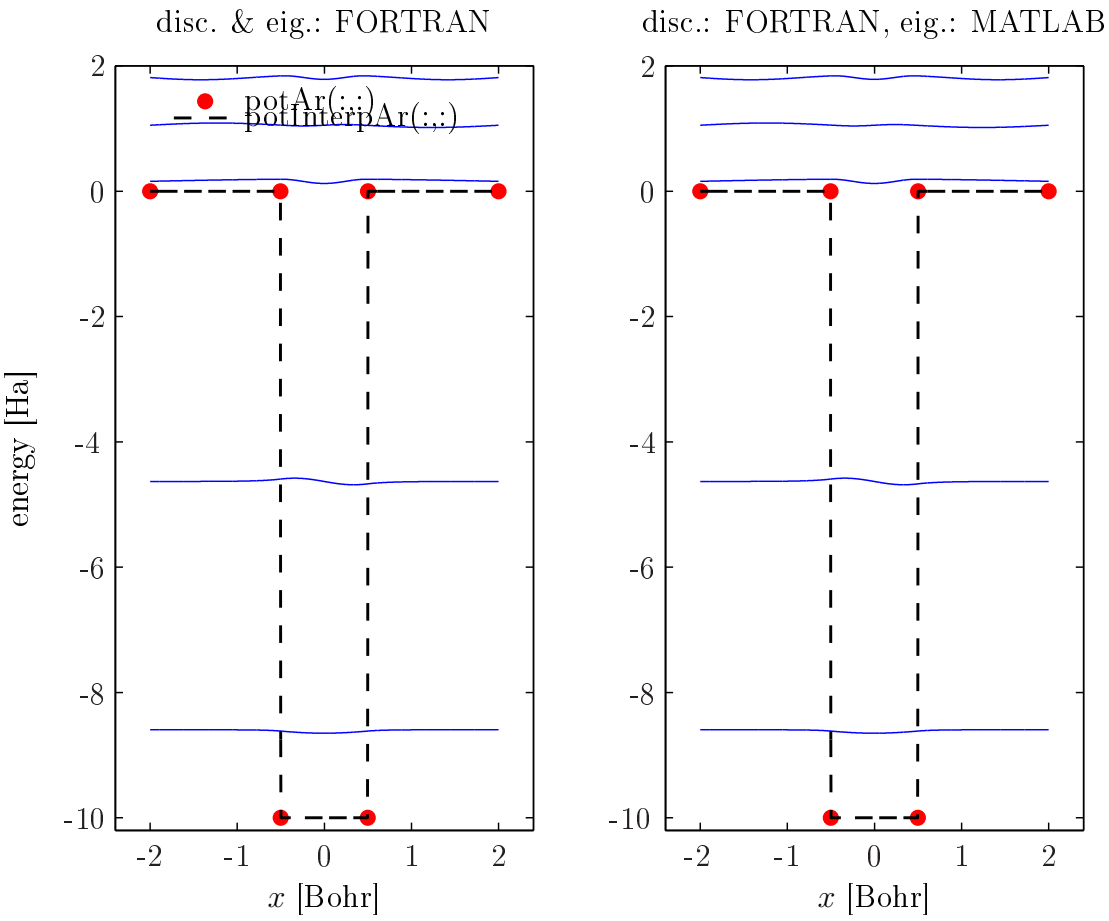
Commands to create this

```
$ make
$ make load_pottopf
./schroedingerSolver
cd documentation
MATLAB >> importfile
```

For `xmgrace` plots, see the instructions on the `make` file above.

Example 2: finite potential well

```
2.0                     # mass
-2.0    2.0     1999    # xMin  xMax    nPoint
1       15              # firstEigVal   lastEigVal
linear                  # interpType
6                       # nInterp
-2.0    0.0             # potAr(:,2)
-0.5    0.0
-0.5    -10.0
0.5     -10.0
0.5     0.0
2.0     0.0
```



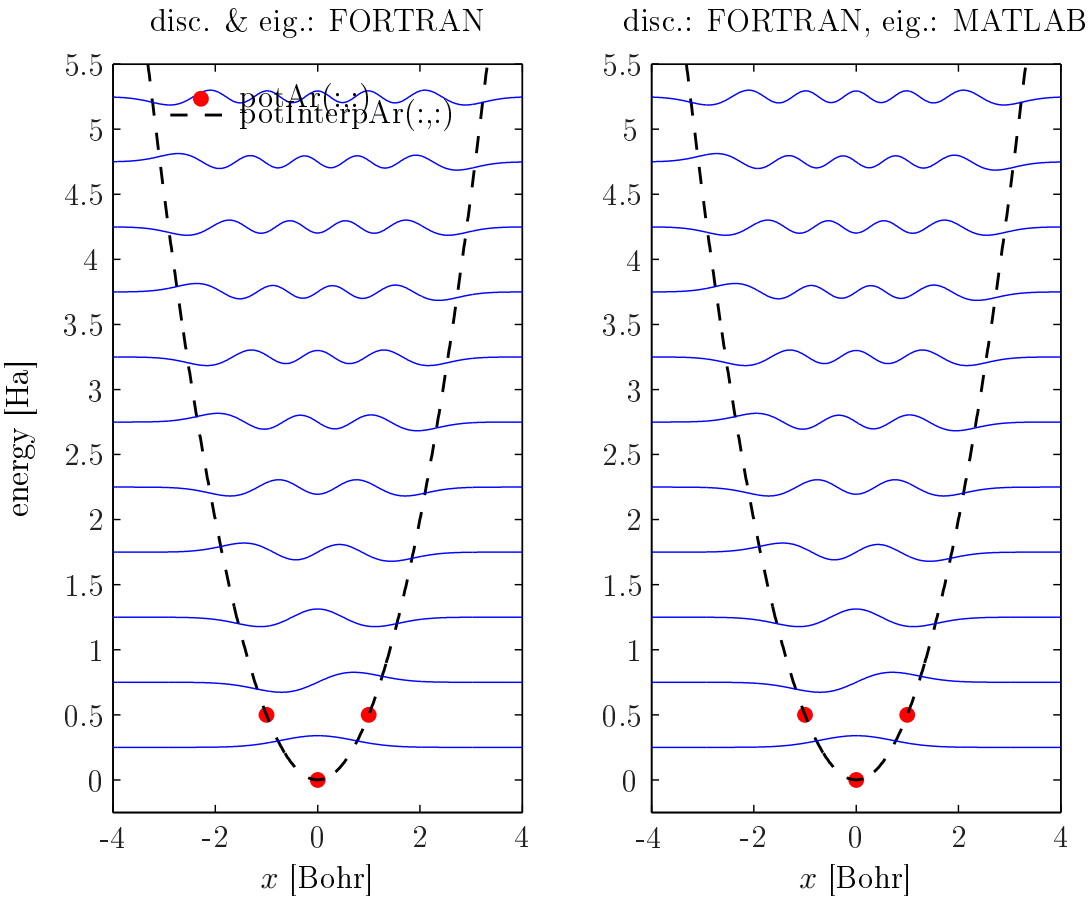
Commands to create this

```
$ make
$ make load_pottopf2
./schroedingerSolver
cd documentation
MATLAB >> importfile
```

For `xmgrace` plots, see the instructions on the `make` file above.

Example 3: harmonic oscillator potential

```
4.0                     # mass
-10.0   10.0    1999    # xMin  xMax    nPoint
1       15              # firstEigVal   lastEigVal
polynomial              # interpType
3                       # nInterp
-1.0    0.5             # potAr(:,2)
0.0     0.0
1.0     0.5
```



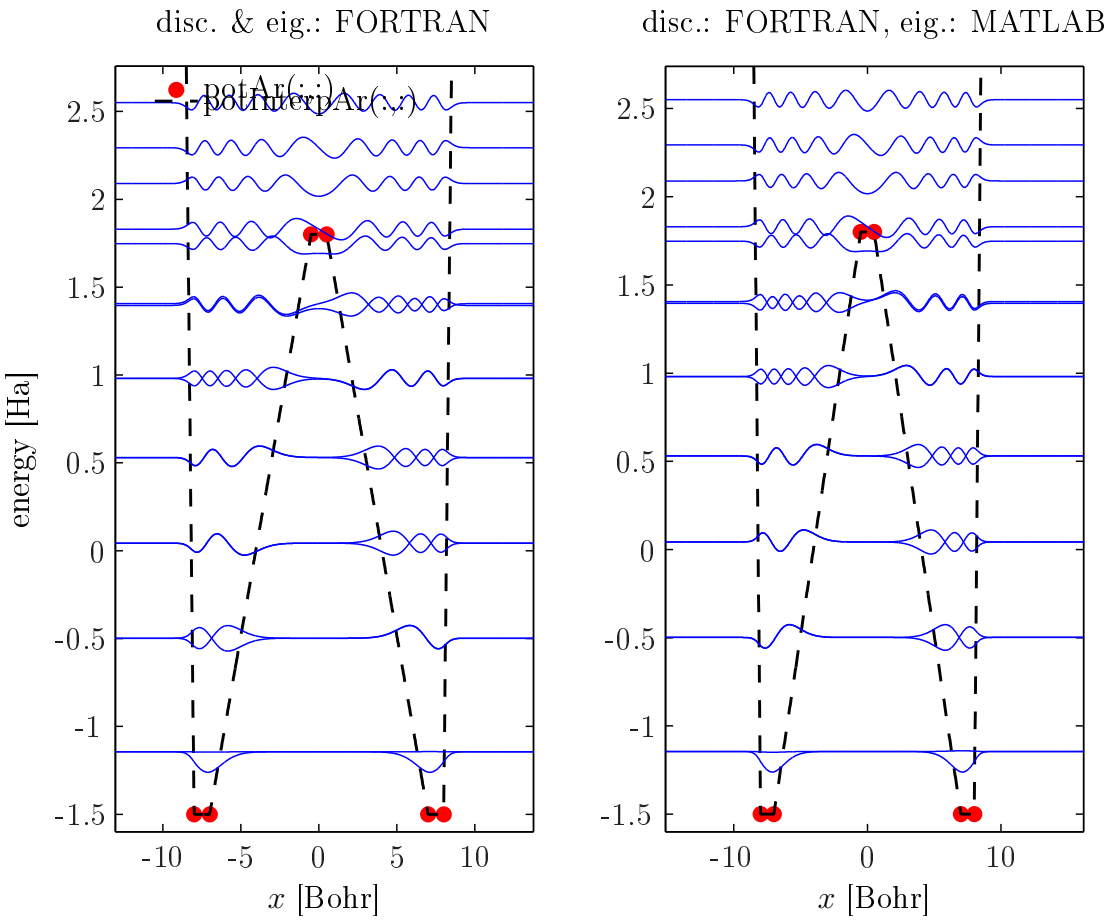
Commands to create this

```
$ make
$ make load_harmOszi
./schroedingerSolver
cd documentation
MATLAB >> importfile
```

For `xmgrace` plots, see the instructions on the `make` file above.

Example 3: double potential well

```
2.0                     # mass
-20.0   20.0    1999    # xMin  xMax    nPoint
1       15              # firstEigVal   lastEigVal
linear                  # interpType
8                       # nInterp
-20.0   100.0           # potAr(:,2)
-8.0    -1.5
-7.0    -1.5
-0.5    1.8
0.5     1.8
7.0     -1.5
8.0     -1.5
20.0    100.0
```



Commands to create this

```
$ make
$ make load_pottopf3
./schroedingerSolver
cd documentation
MATLAB >> importfile
```

For `xmgrace` plots, see the instructions on the `make` file above.

A few notes, hints, remarks, and sort of an outlook

A programming project may have, roadmaps, release dates, hand-in dates etc., but it can never be finished. There are a couple of things which would certainly be worth doing. A lot of interesting things can be found in the documentation of the repositories. For example, editing the schrodinger.inp file in the editor kate caused a lot of trouble until the problem was discovered: kate sometimes suppresses a line terminator close to the end of a file, which e.g. emacs or gedit don't (see [2] for comp.lang.fortran discussion). Another interesting journey was the so called cherrypick-merge which bazaar is able to perform, but which is not documented as a merge in the repository and which is therefore withdrawn from the bzr help merge page, but which nevertheless works perfectly well (this is discussed in greater detail in revno 13, also see [3]). And last, but certainly not least, for the purpose of dealing with unsorted interpolation points given by the user, and algorithm was developed which the author is especially proud of (discussed in detail in revno 15) and which is somewhat similar to Newton's pendulum. All of this would have been a pleasure to discuss and to illustrate in this documentation, and certainly there are spots where the source code might shortened, generalized, modularized, documented more extensively or more concisely, but our experience is that source code is always *not quite perfect*.

References

[1] Bálint Aradi: Projektspezifikation – Abschlussprojekt für Wissenschaftliches Programmieren. contained in the repository.

[2] Alexander Erlich: *How to read the last line before the EOF is reached?* discussion in **comp.lang.fortran** started on 09.09.2009, see
`http://groups.google.com/group/comp.lang.fortran/browse_thread/thread/cc3157a233bf`

[3] John A Meinel: *'bzr help merge' should describe merging a single file.* Launchpad bug report #81758, see
`https://bugs.launchpad.net/bzr/+bug/81758`