RAUTO: Running AUTO more efficiently

Frank Schilder

12th April 2007

Copying

Contents

1 Introduction

Rauto is a command line interface to the software package AUTO and might be of interest to expert as well as novice users. In fact, if you are new to AUTO you might find it easier to start straight away with rauto. The intention of this software package is to make life easier for all users of AUTO. Rauto was tested with AUTO97, but should run with all later versions as well.

Installation

If not already done so set the environment variable `AUTO_DIR` to the location of AUTO. Then run

```
make prefix=install_dir
make install
```

where *install_dir* is the directory under which you want to install rauto (default: *install_dir* = `/usr/local`). This will install all files of rauto in sub-directories of *install_dir* as follows:

install_dir`/bin`	:	scripts and binaries
install_dir`/share/rauto`	:	examples and auxiliary files
install_dir`/share/doc/rauto`	:	documentation (PDF and HTML)

Make sure that *install_dir*`/bin` is in the search path on your system. You can remove rauto from your system by executing the commands

```
cd install_dir/share/rauto
make uninstall
```

If you are not using a make program that supports GNU make's `-W` option, you may need to use '`make cmddep=off`' instead of '`make`'.

Internationalization

When running rauto on a localized system, that is, the environment variable LANG is set to something else than 'C', you may encounter an error message like

```
rauto/cusp> rauto cusp LPf muf -IRS 2 -ISW 2
rauto: *** number of runs=1
rauto: *** starting run LPf: run 1 of 1, IRS=2
printls: numeric format of language 'de_DE.UTF-8' not supported
printls: set the environment variable LANG to C before running printls
rauto: shell returned with error, offending command was:
printls muf 2 > l.LPf
```

To avoid subtle errors due to different notational conventions for numerical values in different languages every AWK script shipped with rauto checks whether the number format under the current locale is understood by AUTO. If this test fails the script tries to change the locale to 'C'. Unfortunately, the current standard of AWK does not require an implementation to support changing the locale at run-time. Hence, most of the time this will fail. For the moment you have to change the locale of your session manually by typing '`setenv LANG C`' if you are using the csh or tcsh, and '`export LANG=C`' if you are using the sh or bash command interpreter.

The name of the game

Every name has to mean something, so what is the story of rauto? Well, performing computations with AUTO is not exactly easy. It is, to pick a random example, more complicated than removing all your files with '`rm -rf ${HOME}`'. For every computation one has to edit or create a so-called

constants file and AUTO itself is executed with a number of (rather obscure?) @-commands, which the author found almost impossible to remember when to use in what form.

The basic idea behind rauto is to hide all this nastiness. It combines a number of tasks into a single invocation of the command rauto and stores the data of your computations in an organized manner. It comes with a number of powerful tools for post-processing your data. It makes using AUTO as easy as removing all your files. But I'm digressing. The name rauto does make some sense, in particular, if you are familiar with some subtleties of the German language. It can be read as 'Run AUTO!' (as in 'Run Forest, run!')

Notation

We use an abbreviated Backus Naur form for describing the syntax of commands. As usual, | denotes alternative elements, [ ] encloses optional elements, ( ) groups elements and ... denotes zero or more repetitions of the previous element. Metasymbols are typeset in *italic font* and literal input in `teletype font`, except in running text, where we try to avoid excessive changes of font.

Contact

Frank Schilder, Department of Mathematics, University of Surrey, Guildford, UK
(f.schilder@surrey.ac.uk)

2 The 4.5 minutes guide to AUTO

Experienced users of AUTO most probably know about an interesting phenomenon. It takes less that five minutes to explain AUTO to novice users, while it seems almost impossible to start with the manual. The intention of this section is to provide more or less this 'less than five minutes' introduction.

If you are familiar with AUTO you may skip this section and start reading with Section 'Running rauto' on page 10. The following description of AUTO will be on the version AUTO97. There has been a number of changes in AUTO2000, but we stick to AUTO97 here, because the basic principle of use did not change and AUTO97 still seems to be a quite popular version. Rauto should work as well with AUTO2000. A new version AUTO07 is under development and we will upgrade rauto and this description sometimes after it became the new official version (more precisely, after the author decides to switch to AUTO07).

AUTO is a software package for computational bifurcation analysis of generic dynamical systems. The most commonly used features are the computation of families of equilibrium points and periodic solutions of autonomous ODEs

$$\dot{x} = f(x, \mu), \qquad x \in \mathbb{R}^n, \quad \mu \in \mathbb{R}^m \tag{1}$$

and their local bifurcations, and in this tutorial we will focus on problems of this type. AUTO offers substantially more functionality, but the principal of use remains always the same. Please see the reference manual [1] for more information.

The first you need to know is that AUTO is neither a stand-alone program nor a library, it is something in between. AUTO is an almost complete program that comes as a library. It is incomplete because it naturally lacks the subroutines defining your right-hand-side of ODE (1) and it has the form of a library because that's the easiest way to deal with this situation. However, you don't need to worry too much about these things, because rauto takes care of the messy part. It is just a little helpful to know that running rauto involves compiling and linking of Fortran files and that you might be confronted with error messages from the Fortran compiler. But we are running ahead, let's start at the beginning.

BR	PT	TY	LAB	PAR(2)	L2-NORM	U(1)	PAR(1)
1	1	EP	1	0.00000E+00	0.00000E+00	0.00000E+00	1.00000E+00
1	14	LP	2	3.84900E-01	5.77351E-01	-5.77351E-01	1.00000E+00
1	20		3	1.68950E-01	9.01430E-01	-9.01430E-01	1.00000E+00
1	39	UZ	4	-9.99998E-01	1.32472E+00	-1.32472E+00	1.00000E+00
1	40		5	-1.09746E+00	1.34710E+00	-1.34710E+00	1.00000E+00
1	50	EP	6	-2.07958E+00	1.53463E+00	-1.53463E+00	1.00000E+00

Figure 1: Screen output of a typical AUTO run.

The philosophy of AUTO

In order to perform computations with AUTO you need to set up an *equations file* and a *constants file*. The equations file is a Fortran source code file where you specify the right-hand side of your ODE (1) and an initial solution for some parameter values. The equations file offers further functionality, which is explained in the reference manual [1]. The naming convention for the equations file is *name*`.f`, where *name* is a short unique identifier for your problem. The constants file is a plain ASCII text file with name `r.`*name* and contains values for certain parameters of AUTO's algorithms as well as some constants of your problem, for example, the dimension of ODE (1). The easiest, most typical, and recommended way to create these two files is to copy an example from this manual or from [1], and to modify the equations and constants file of this copy. There are even commands for this: rdm (rauto) and @dm (AUTO [1]).

After setting up all files we can start computations with AUTO. A single computation with AUTO is called a *run*. In each run a part of a branch is computed, that is, a part of a parameter dependent family of equilibrium points or periodic solutions. The computation of a complete bifurcation diagram normally consists of multiple runs. We will refer to runs starting at the initial solution specified in the equations file as *initial runs*. Subsequent runs starting at solutions computed in a previous run will be called *restarted runs*. There is no difference in the actual computations involved, but the command line of rauto differs slightly for both cases.

AUTO can perform a stability analysis during continuation. If this feature is enabled AUTO will detect and report a number of so-called *special points*. Most commonly, these are bifurcation points along a branch at which the solutions change stability. Other special points are points along a branch at which a user defined condition is satisfied. Figure 1 shows a typical output of an AUTO run. The first column `BR` is the number of the branch, which we will ignore here. Column `PT` is the number of the solution along the branch, that is, the continuation step in which the solution was computed. Column `TY` indicates the type of the point. If this column is empty the point is called a *regular output point*. All other points are special points. For example, LP marks a limit point or saddle-node bifurcation point, and UZ is a point at which the user defined condition PAR(2) $= -1$ is satisfied (within numerical accuracy). Table 1 gives a complete overview over all special point types that AUTO can detect.

Column `LAB` is a unique *label* that is assigned to any solution that is written to the permanent output of AUTO. These labels are used to identify a *restart solution* for initializing a restarted run. Often this will be a special point, in which case we perform *branch switching* to a family of solutions branching off a bifurcation point. For example, at a Hopf bifurcation point HB of an equilibrium point we can switch to the continuation of the periodic solutions emerging from this point.

The first column after `LAB` is the *primary continuation parameter*, and the column `L2-NORM` is the Euclidian norm (equilibrium points) or $\mathcal{L}_2$ norm (periodic solutions) of the solution. The contents of all other columns depends on the actual continuation and will be indicated by an appropriate caption. In Figure 1 these two additional columns are the first coordinate and another free parameter of ODE (1).

BP	(1)	Branch point (algebraic systems)
LP	(2)	Fold (algebraic systems)
HB	(3)	Hopf bifurcation
	(4)	User-specified regular output point
UZ	(-4)	Output at user-specified parameter value
LP	(5)	Fold (differential equations)
BP	(6)	Branch point (differential equations)
PD	(7)	Period doubling bifurcation
TR	(8)	Torus bifurcation
EP	(9)	End point of branch; normal termination
MX	(-9)	Abnormal termination; no convergence

Table 1: Solution Types; table taken from [1].

The equations file

If you are new to AUTO you might also be new to Fortran. Therefore, this section will equip you with the necessary basics to get started without having to consult other sources about Fortran first. We would like to note here again that the more recent version AUTO2000 allows you to define your ODEs in C. However, it is useful to know about the Fortran version, because all examples in this package are in Fortran, existing codes of ODEs in Fortran will continue to be around for some while, and a number of users actually prefers Fortran over C. Our description will be on the quite old Fortran 77 standard. Newer versions of Fortran are much more flexible, but these are not yet widely available, in particular, not in open source implementations like the popular g77. Furthermore, the restrictions imposed by the Fortran 77 standard are negligible here. At the end of the day all we have to do is to fill in the body of two subroutines.

The equations file *name*`.f` must define all of the Fortran subroutines

Subroutine	:	Defines
FUNC	:	right-hand side of ODE (1),
STPNT	:	an *a-priory* known initial or seed solution,
BCND	:	boundary conditions,
ICND	:	integral conditions,
FOPT	:	objective function,
PVLS	:	user-defined output.

Here, we are only concerned with the first two subroutines, for the others we use default implementations that do nothing. Detailed descriptions of all subroutines, including the empty ones, can be found in the equation files provided with the examples of `rauto`. Figure 2 shows a sample implementation of the two subroutines FUNC and STPNT. FUNC evaluates the right-hand side of the one-dimensional ODE $\dot{x} = f(x, \lambda, \mu) := \mu + \lambda x - x^3$. The state vector x is in the input array U, the system parameters (λ, μ) are in the input array PAR, and the vector $f(x, \lambda, \mu)$ is written to the output array F. All other arguments are (normally) unused. STPNT initializes the state variables and parameters with the initial equilibrium solution $x = 0$ at $\lambda = 1$ and $\mu = 0$. The argument NDIM is (normally) unused.

Let us explain step by step the format of the source code in Figure 2. Fortran is not case sensitive, so `fortran`, `Fortran` and `fORTRAN` are all the same. Fortran source code consists of *statements* and *comments*. A statement has one or more lines, an *initial line* and optional *continuation lines*, which allow splitting of longer statements over several lines. Comments have one or more *comment lines*. Each type of line has its own format. Initial and continuation lines contain the program statements and consist of four fields, each starting and ending at specific character positions (columns):

```
CLLLLC<--                    statements go here                        -->

      SUBROUTINE FUNC(NDIM,U,ICP,PAR,IJAC,F,DFDU,DFDP)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION U(NDIM), PAR(*), F(NDIM), ICP(*)

      F(1) = PAR(1) * U(1) - U(1)*U(1)*U(1) + PAR(2)

      RETURN
      END

      SUBROUTINE STPNT(NDIM,U,PAR)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION U(NDIM), PAR(*)

      PAR(1) = 1.0D0
      PAR(2) = 0.0D0

      U(1) = 0.0D0

      RETURN
      END
```

Figure 2: Sample subroutines FUNC for the ODE $\dot{x} = \mu + \lambda x - x^3$ and STPNT for the starting point $\lambda = 1$, $\mu = 0$ and $x = 0$.

Columns	:	Field name
1 through 5	:	statement label,
6	:	continuation indicator,
7 through 72	:	statement,
73 to end of line	:	comment (optional).

Initial lines have an optional statement label, which is a unique number that can be used to refer to it. The continuation indicator is blank and the columns 7 through 72 contain program statements. The statement line can end before column 72, but *everything typed from column 73 until the end of line will be treated as a comment, that is, ignored.* A **continuation line** continues a Fortran statement that needs to be split over more than one line. A continuation line has a blank statement label. The continuation indicator contains any Fortran character other than blank or the digit 0. This indicator might be used to count the continuation lines. At least 6 continuation lines must be supported by any Fortran compiler. The statement and comment fields are as for initial lines. A **comment line** serves solely the purpose of commenting a program and does not affect execution. A comment line is either an entirely blank line or a line beginning with a `C`, `c` or an asterisk `*`. The first line in the source code of Figure 2 is a comment line indicating the fields of initial and continuation lines. All examples that come with `rauto` contain such comment lines at appropriate locations to make typing easier.

The first statement line 'SUBROUTINE ...' defines a subroutine with name FUNC and a list of formal parameters, which are:

Input arguments		
NDIM	:	dimension of the ODE (normally unused)
U	:	state or phase-space variables
ICP	:	list of active continuation parameters (normally unused)
PAR	:	system parameters
IJAC	:	Jacobian computation flag (normally unused)
Output arguments		
F	:	right-hand side of ODE
DFDU	:	Jacobian (normally unused)
DFDP	:	parameter derivatives (normally unused)

The data type of the formal parameters is not part of the subroutine definition. The body of a subroutine consists of a list of statements that is terminated with an END statement.

The second statement line 'IMPLICIT ...' sets the *assumed data type* for variables beginning with a certain letter to double precision. By default the data type of variables with a name beginning with I, J, K, L, M or N is integer, and the type of all other variables is real (single precision). The data type of variables can also be declared explicitly with a type statement, for example, 'DOUBLE PRECISION FOO' declares a variable FOO of type double precision. Type statements inside the body of a subroutine affect the assumed data type of its arguments. Consequently, NDIM, ICP and IJAC are of type integer and U, PAR, F, DFDU and DFDP of type double precision. All computations in AUTO use double precision, so watch out for unwanted conversions. Avoid using single precision functions and constants, because this may lead to poor convergence behavior of AUTO's algorithms. Literal double precision constants are of the form <*integral-part*>.<*fractional-part*>`D[+|-]`<*exponent*>. Either the integral or the fractional part may be missing. Note the D in the exponential notation, `0.1` and `1.0e-1` are single precision constants. Fortran provides all mathematical standard functions, like trigonometric functions and exponentials. The double precision functions begin with a D, for example, `DSIN(X)` computes the sine of x in double precision.

The 'DIMENSION ...' statement declares a number of arrays. The dimension of an array can be given explicitly in the form `[`*lb*`:]`*ub*, where *lb* is the lower and *ub* the upper bound. If the lower bound is omitted, indexing starts with 1. If an asterisk `*` is given for the upper bound of the last dimension, the array is a so-called *assumed-size* array with an unspecified upper bound. However, since Fortran does not check array bounds all declarations in Figure 2 are equivalent. Arrays can be multi-dimensional. Array elements are referenced with round brackets as in `U(1)`.

At the next statement line starts the core part of the subroutine FUNC, namely, the evaluation of the right-hand side of the ODE $\dot{x} = f(x, \lambda, \mu) := \mu + \lambda x - x^3$ at some point x. Our ODE is one-dimensional and depends on two parameters, hence, we identify U(1)=x, PAR(1)=λ and PAR(2)=μ. The statement line 'F(1) = ...' evaluates $f(x, \lambda, \mu)$ and assigns the resulting value to the array element F(1). In general, we have U(1)=x_1, ..., U(NDIM)=x_n and PAR(1)=λ_1, ..., PAR(`*`)=λ_m and the components of $f(x, \lambda, \mu)$ are written to F(1),..., F(NDIM). The maximal dimension of an ODE is NDIM=12 by default. This limit can be changed; see [1] for more details. The parameters that can be used in FUNC are PAR(1), ..., PAR(9), parameters PAR(I) with indices I $\geq$ 10 are used internally by AUTO. These internal parameters may be used as continuation parameters or for output purposes. Most commonly, this will be the period of a periodic solution stored in PAR(11) or the rotation number along a locus of Neimark-Sacker points in PAR(12).

The arithmetic expressions that can be used to compute the right-hand side are almost as in other programming languages, just take care of the data types. Fortran provides all mathematical standard functions, the binary arithmetic operators `+`, `-`, `*` and `/`, the unary sign operators `+` and `-`, as well as the exponentiation operator `**`. The assignment operator is `=`, multiple assignments as in `a=b=c` are not allowed.

```
1 1 0 1              NDIM,IPS,IRS,ILP
2 2 1                NICP,(ICP(I),I=1,NICP)
5 4 3 2 1 0 0 0      NTST,NCOL,IAD,ISP,ISW,IPLT,NBC,NINT
200 -2. 2. 0 100     NMX,RL0,RL1,A0,A1
20 0 2 8 5 3 0       NPR,MXBF,IID,ITMX,ITNW,NWTN,JAC
1.e-6 1.e-6 0.0001   EPSL,EPSU,EPSS
0.01 0.005 0.1 1     DS,DSMIN,DSMAX,IADS
0                    NTHL,((I,THL(I)),I=1,NTHL)
0                    NTHU,((I,THU(I)),I=1,NTHU)
0                    NUZR,((I,UZR(I)),I=1,NUZR)
```

Figure 3: Sample constants file.

The body of the subroutine FUNC is terminated with the two consecutive statements RETURN and END. The subroutine STPNT (STarting PoiNT) defined below FUNC initializes U and PAR with an equilibrium point or periodic solution of the ODE at certain parameter values, here $x = 0$, $\lambda = 1$ and $\mu = 0$. It has only one input argument NDIM (normally unused) and two output arguments U and PAR. Note that the initial point must not be a bifurcation point.

The constants file

The constants file is a plain text file containing values for some problem constants, for example, the dimension of the ODE and parameters influencing the performance of AUTO's algorithms. Figure 3 shows a sample constants file, which is split optically into two parts. The left-hand block of numbers are the actual constants and the right-hand block contains commenting text. The format of these comments reflects the formatted READ statements used to read in the individual lines. For example, AUTO uses the Fortran statement

```
READ(2,*,END=5) NDIM,IPS,IRS,ILP
```

to read in the first line of a constants file. When executed, this example statement reads the full first line and assigns the first four values to the variables listed after the closing bracket. Any surplus data, in our case the comments, is ignored. The data type of the constants is integer if the first letter is I, J, K, L, M, or N, and double precision otherwise. The data in the constants file is interpreted according to the data type of the constant. Therefore, `1.e-6` *in the constants file is a double precision number,* while it would be of single precision in Fortran source code.

In the order they appear in the constants file the AUTO constants are: **NDIM** is the dimension of the ODE. **IPS** sets the solution type you want to compute, 1=equilibrium points of ODEs and 2=periodic solutions of ODEs. **IRS** is the label of the restart solution, that is, the solution you want to start your continuation with. If IRS>0 use solution with label IRS of a previous run, if IRS=0 use solution set by subroutine STPNT. **ILP** switches limit point detection on (ILP=1) or off (ILP=0).

NICP is the number of free continuation parameters. The continuation of equilibrium points requires at least one free parameter (NICP≥1) and the continuation of periodic solutions two (NICP≥2). **ICP** is an array of dimension NICP and stores the indices of the free continuation parameters. The first parameter PAR(ICP(1)) is the *primary continuation parameter.* For the continuation of periodic solutions a *secondary continuation parameter* must be present. The natural choice is the period stored in PAR(11). It is sometimes useful to include more free parameters than necessary, which results in additional output. This is called *parameter overspecification* and at most seven parameters may be specified; see [1] for more details.

NTST is the number of mesh intervals used for computing periodic solutions. This should be set to a moderate value (5≤NTST≤20) and only be increased if necessary. **NCOL** is the number of collocation points per mesh interval. Default is NCOL=4, which should normally not

be changed. **IAD** controls the mesh adaptation during continuation. The distribution of mesh points is adapted every IAD continuation steps if IAD>0, or kept fixed if IAD=0 (adaptation off). **ISP** controls the detection of special points (bifurcation points). ISP=0 disables detection of special points. With ISP=1 branch points of equilibria, with ISP=3 branch points of periodic solutions and with ISP=2 all special points will be detected (equilibria and periodic solutions). **ISW** indicates if branch-switching shall be performed. The normal setting is ISW=1. If IRS is the label of a Hopf bifurcation (HB) branch point (BP) or a period-doubling bifurcation (PD) then setting ISW=−1 will start to continue the branch of solutions branching off the special point. If IRS is the label of a fold (LP), Hopf (HB), period doubling (PD) or torus bifurcation point (TR) then IRS=2 will start a codimension-one continuation of a locus of such points. An additional free continuation parameter must be present for such continuations. **IPLT** selects the *principal solution measure* shown in column 5 of AUTO's output; see Figure 1. The normal setting is IPLT=0 and is rarely changed. **NBC** is the number of boundary conditions and **NINT** the number of integral conditions when AUTO is used for the continuation of solutions of general two-point boundary value problems, which we are not concerned with here. We use the setting NBC=NINT=0.

NMX is the maximal number of continuation steps. AUTO will stop with EP after at most NMX continuation steps have been performed. **RL0** and **RL1** are the lower and upper limit for the primary continuation parameter. AUTO will stop with EP whenever the primary continuation parameter leaves the interval [RL0, RL1]. **A0** and **A1** are bounds for the principal solution measure. AUTO will stop with EP whenever the principal solution measure leaves the interval [A0, A1].

NPR controls the amount of data saved during continuation. AUTO will save the solution and full restart data every NPR continuation steps in addition to all solutions at special points. NPR should not be too small to avoid excessive data storage. A value around NMX/10 is usually a good choice. **MXBF** controls automatic branch switching for equilibrium point continuation, which allows to some extend the automatic computation of a bifurcation diagram. The normal setting is MXBF=0. **IID** controls the amount of diagnostic output generated by AUTO. This is useful for debugging purposes. The diagnostic output is written to the file `data/d.`*run*, where *run* is the name of the run. The normal setting is IID=2. **ITMX** is the maximum number of iterations allowed for locating special points. The normal setting is ITMX=8. **ITNW** is the maximum number of combined Newton-Chord iterations for the correction step of the continuation method. The normal setting is ITNW=5, but may need to be increased for difficult problems. **NWTN** is the maximum number of full Newton steps of the combined Newton-Chord iterations. The normal setting is NWTN=3. This may need to be increased for difficult problems. **JAC** is a flag indication whether the user provides derivatives in the subroutine FUNC. The normal setting is JAC=0 (no derivatives provided). If set to JAC=1 the user must provide derivatives with respect to state variables and parameters.

EPSL Convergence criterion for equation parameters. AUTO will stop with MX if the error in the parameters becomes larger than EPSL. The normal setting is EPSL=$10^{-7}\ldots10^{-6}$. **EPSU** Convergence criterion for state variables. AUTO will stop with MX if the error in the state variables larger than EPSU. The normal setting is EPSU=$10^{-7}\ldots10^{-6}$. **EPSS** Convergence criterion for locating special points. The normal setting is EPSU=$10^{-4}\ldots10^{-3}$. EPSS should be $100\ldots1000$ times greater that EPSL, EPSU. AUTO behaves a bit erratic when this criterion cannot be satisfied. Sometimes it will stop with MX and sometimes it will resume the continuation without mentioning. Check the diagnostic output file `data/d.`*run* for possible special points that went unnoticed.

DS is the initial continuation step size. This value should not be too large and must satisfy DSMIN $\leq$ |DS| $\leq$ DSMAX. A negative value reverses the direction of continuation. **DSMIN** is the minimal allowed continuation step size. AUTO will stop with MX if the correction step fails with |DS|=DSMIN. **DSMAX** is the maximal allowed continuation step size. AUTO will not

use larger steps. Note that increasing DSMAX also increases the risk of missing special points. **IADS** controls adaptation of the continuation step size. The continuation step size is adapted every IADS continuation steps, or kept fixed if IADS=0 (adaptation off).

The last three entries all have the same syntax and read values into arrays. The first constant **NTHL**, **NTHU** or **NUZR** appears on a separate line and specifies the number of subsequent array entries to read in. After this line follow precisely NTHL, NTHU or NUZR lines containing two values: an index I and a real number X. **THL** is an array defining weights for individual equation parameters and **THU** for individual state variables. I is the index of the parameter or variable and X is the new weight assigned to it. By default, all weights are 1. These weights are used in the computation of the distance between two points in combined state and parameter space and affect the continuation step size. This functionality is sometimes used to exclude the period from the computation of the continuation step size by setting THL(11)=0. **UZR** is a list of parameter values for which the solution shall be saved. A label will be assigned to such solutions and they can be used as initial solutions of restarted runs. I is the index of the parameter and X is the value for which the solution shall be saved whenever PAR(I)=X holds during continuation. If I is negative then AUTO will save the solution and stop with EP if PAR(|I|)=X holds. The maximum number of user output points is normally limited to 20. This limit can be changed; see [1] for more details.

A one-page overview of all AUTO constants with typical values is given in Appendix B and a full description can be found in the manual of AUTO [1].

3 Running rauto

The easiest way to get started with rauto is to look at the examples provided with the package. We explain rauto's typical work cycle using the cusp normal form $\dot{x} = \mu + \lambda x - x^3$ as a simple example. Create a directory where you would like to do your computations, for example, `rauto` and change to this directory. Then type '`rdm cusp`'. This will create the sub-directory `cusp`. Change to the directory cusp and type '`ls`'. You should now see something like

```
frank/tmp> mkdir rauto
frank/tmp> cd rauto/
tmp/rauto> rdm cusp
tmp/rauto> ls
cusp
tmp/rauto> cd cusp
rauto/cusp> ls
cusp.f  demo  r.cusp  scandemo
```

The *equations file* is `cusp.f` and the *master constants file* is `r.cusp`. Note that the names of these two files follow the pattern *name*`.f` and `r.`*name*, which is required by rauto. We ignore the files `demo` and `scandemo` for the moment.

In our *initial runs*, that is, runs starting at the solution provided in STPNT, we are going to compute a branch of equilibrium points with respect to the parameter μ. These computations will start at the point $x = 0$, $\lambda = 1$ and $\mu = 0$. Before you continue:

- Take a look at the equations and master constants file.
- Sketch the μ-x-bifurcation diagram for $\lambda > 0$.

In our equations file U(1) = x, PAR(1) = λ and PAR(2) = μ. The constants file is preset for a continuation in $\mu \in [-2, 2]$ for μ increasing. The basic call to rauto for performing an initial run is

`rauto` [`-h`] *name run*

where *name* is the name of the problem (cusp), and *run* is a unique name we assign to the computation or branch for later reference. Note that rauto and all its auxiliary programs have a -h switch, which will print a usage message and exit. Since we are going to perform a continuation for increasing μ, we call our first run `muf` for 'μ forwards'. To run this computation simply type '`rauto cusp muf`'. You should see something like

```
rauto/cusp> rauto cusp muf
rauto: *** number of runs=1
rauto: *** starting run muf: run 1 of 1, IRS=0
Starting cusp ...

  BR    PT  TY LAB     PAR(2)         L2-NORM          U(1)         PAR(1)
   1     1  EP   1   0.00000E+00   0.00000E+00   0.00000E+00   1.00000E+00
   1    14  LP   2   3.84900E-01   5.77351E-01  -5.77351E-01   1.00000E+00
   1    20       3   1.68950E-01   9.01430E-01  -9.01430E-01   1.00000E+00
   1    40       4  -1.19337E+00   1.36827E+00  -1.36827E+00   1.00000E+00
   1    49  EP   5  -2.07783E+00   1.53434E+00  -1.53434E+00   1.00000E+00

 Total Time    0.240E-01
cusp ... done
```

This will create the directory `data`, which stores the results of all rauto computations.

Rauto comes with a number of tools for processing data obtained through rauto computations. The most elementary of these are `lsruns`, `printbd` and `rmruns`. Lsruns will print a list of runs, printbd will re-print rauto's output of a run and rmruns will delete all files of all runs or of the runs given on the command line. Try '`lsruns`' and '`printbd muf`'. A more sophisticated tool is `plotbd`. Plotbd will plot a bifurcation diagram using all runs specified on the command line. For the moment we have only muf, so try '`plotbd muf`'. If you are a fan of black-and-white bifurcation diagrams, try '`plotbd -bw muf`'. You should see a gnuplot window with a plot similar to Figure 4. The horizontal axis is μ and the vertical axis is $\|x\|_2$, which corresponds to columns 5 and 6 of rauto's output during continuation. You may specify an argument of the form *xaxis*:*yaxis* to plot other columns, where *xaxis* and *yaxis* are the numbers of columns to be plotted as the x- and y-axis. Counting starts with column PAR as number 1. Try to plot x versus μ.

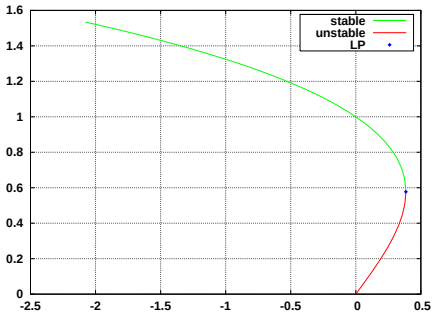


Figure 4: `plotbd muf`

In a subsequent initial run we are now going to compute the missing part of the branch computed so far by continuation from the same starting point, but in the other direction. Instead of changing DS in the master constants file, we rather tell rauto on the command line to start a continuation in the other direction. Rauto allows to set almost any AUTO constant to a value different from that in the master constants file by using a command of the extended form

`rauto` *name run* (`-`*auto_constant value*) ...

Auto_constant is the name of any AUTO constant that is of type integer or double precision and *value* is the new value of appropriate type that shall be assigned to it. However, the constant DS is special in two aspects. When reading a constants file rauto ignores the sign and initially sets DS to the absolute value of the number found in the constants file. A negative value for DS must be given on the command line. To simplify matters DS also accepts the *direction flags* `+` and `-` on the command line instead of a floating point number. If you just want to start a continuation in one or the other direction and are happy with the absolute value of DS in the constants file then specify a direction flag. In our example, to start a continuation for decreasing

μ type '`rauto cusp mub -DS -`'. Note the change of the run name to mub for 'μ backwards'. You should see something like

```
rauto/cusp> rauto cusp mub -DS -
rauto: *** number of runs=1
rauto: *** starting run mub: run 1 of 1, IRS=0
Starting cusp ...

  BR    PT  TY LAB    PAR(2)        L2-NORM         U(1)        PAR(1)
   1     1  EP   1   0.00000E+00   0.00000E+00   0.00000E+00   1.00000E+00
   1    14  LP   2  -3.84900E-01   5.77351E-01   5.77351E-01   1.00000E+00
   1    20       3  -1.68950E-01   9.01430E-01   9.01430E-01   1.00000E+00
   1    40       4   1.19337E+00   1.36827E+00   1.36827E+00   1.00000E+00
   1    49  EP   5   2.07783E+00   1.53434E+00   1.53434E+00   1.00000E+00

 Total Time    0.800E-02
cusp ... done
```

This will create a second run, type '`lsruns`' to see a list of all runs. You may plot a combined bifurcation diagram of both runs by typing '`plotbd 1:3 muf mub`', or, the more advanced command '`plotbd 1:3 ‘lsruns‘`' making use of *command substitution* by the shell. The latter technique is very convenient for plotting combinations of many runs.

Typing rauto commands on the command line quickly becomes tiring, in particular, if you need to set many AUTO constants. The recommended way of using rauto is to step by step add rauto commands to a shell script. This not only allows you to collect all data about your computations in a single file, it also facilitates recomputing runs when necessary and massive computations involving large numbers of runs. Let us illustrate this style of working with the example cusp at hand. Type '`rmruns`' and answer 'yes' when asked for confirmation:

```
rauto/cusp> rmruns
Remove the following runs?
 mub       muf
rmruns: (yes/no) yes
rauto/cusp> lsruns
lsruns: no matching runs
```

Create the file '`cuspdemo`' and make it executable:

```
rauto/cusp> touch cuspdemo
rauto/cusp> chmod +x cuspdemo
rauto/cusp> ls -l cuspdemo
-rwx------  1 frank users 0 2007-04-06 19:30 cuspdemo
```

Yes, I was typing this over Easter. Now open the file cuspdemo with your favorite text editor (or use '`cat > cuspdemo`'), and write

```
#!/bin/sh
set -e
rauto cusp muf
rauto cusp mub -DS -
```

This is a simple shell script for the command interpreter sh. The command '`set -e`' makes sure that the execution of the script stops whenever rauto encounters an error. Save the script (or press `^D`) and type '`./cuspdemo`'. This will re-compute the runs muf and mub and produce the same output as before. Now type '`./cuspdemo`' again. This time you should see something like

```
rauto/cusp> ./cuspdemo
rauto: *** number of runs=1
rauto: *** starting run muf: run 1 of 1, IRS=0
rauto: *** run muf up to date, nothing done
rauto: *** number of runs=1
rauto: *** starting run mub: run 1 of 1, IRS=0
rauto: *** run mub up to date, nothing done
```

Rauto employs a number of tests to check whether a run is up to date or needs recomputation. This way you can (and should) add commands for new computations to the end of the list of rauto commands in a shell script and then simply re-execute the script. If you later change some constants of earlier computations rauto will recompute only the runs that might be affected by these changes. Rauto has a '`-f`' switch for forcing a recomputation in the impossible event that these tests fail. Play around with this example, change some constants and watch what happens.

The preceeding discussion should have equipped you with enough information to get started with rauto, in particular, if you are already an experienced user of AUTO. You might want to jump ahead to the command reference and start working with rauto and its many utility programs. With the subsequent examples we demonstrate how to execute restarted runs, select restart solutions, perform branch switching, start from numerical data and we explain rauto's constants inheritance mechanism. These examples also illustrate some advanced features, which include using command substitution, variable expansion and arithmetic expansion together with rauto as well as rauto's multiple run mode for so-called scans for creating 3d-bifurcation diagrams. A section with detailed descriptions of these and even more fancy features is in preparation.

Demo cusp: restart labels, branch switching and scans

To create a working copy of the demo cusp change to a directory where you would like to do your computations, for example, `${HOME}/rauto`. Then type '`rdm cusp`'. This will create the sub-directory `cusp`, which contains the files `cusp.f`, `r.cusp`, `demo` and `scandemo`. Change to the directory cusp.

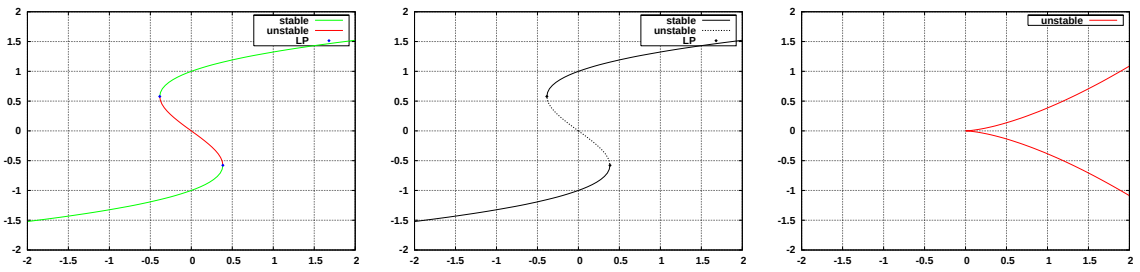
The equations file `cusp.f` defines the right-hand side of the ODE $\dot{x} = \mu + \lambda x - x^3$ with the identifications U(1) $= x$, PAR(1) $= \lambda$ and PAR(2) $= \mu$. The master constants file is `r.cusp` and the files `demo` and `scandemo` are two executable shell scripts for the command interpreter bash serving as demonstration programs. The first script `demo` illustrates branch-switching and executes the rauto commands

```
rauto cusp muf
rauto cusp mub -DS -

rrun=muf
rauto cusp LPf $rrun -IRS $(splabs $rrun LP 1 1) -ISW 2
rauto cusp LPb $rrun -IRS $(splabs $rrun LP 1 1) -ISW 2 -DS -
```

The first two commands perform initial runs in forward (muf) and backward (mub) direction starting at the initial solution given in STPNT. During both runs a limit point (fold or saddle-node bifurcation) is detected; see Figures 5 (a) and (b). In the subsequent runs we compute the locus of limit points in the two parameters λ and μ starting at the limit point found in run muf; see Figure 5 (c). This is called *fold continuation*. To perform such a *restarted run* we use an rauto command of the form

`rauto` *name run rrun* `-IRS` *rlab* [(*-auto_constant value*) ...]

Here, *rrun* is the name of the *restart run* during which the limit point was detected and *rlab* is the *restart label* that was assigned to it by AUTO (column LAB). *Rrun* and *rlab* together uniquely specify a *restart solution*.



(a) `plotbd 1:3 "[-2:2]" "[-2:2]" muf mub`

(b) `plotbd -bw 1:3 "[-2:2]" "[-2:2]" muf mub`

(c) `plotbd 4:1 "[-2:2]" "[-2:2]" LPf LPb`

Figure 5: Bifurcation diagrams of the example cusp. Initial runs in (μ, x) coordinates in (a) color and (b) black-and-white and (c) the fold curve in (λ, μ) coordinates with a cusp point at $(0, 0)$.

If you look at the output of run muf you will see that the restart label of the limit point is 2, so a call like '`rauto cusp LPf muf -IRS 2 -ISW 2`' would do. However, specifying restart labels explicitly is not recommended as we normally can't anticipate what label will be assigned and the labels assigned may change when changing AUTO constants. We would rather like to tell rauto to 'start at the limit point found in run muf'. This functionality is provided by the utility program `splabs` (special point labels). An invocation of splabs has the form

`splabs` *run type* [*begin* [*end*]]

Type is the type of the special point, here LP. The list of labels to be selected can be restricted to a subset by specifying a range of indices with *begin* and *end*. Try to select the labels of all, the first and the last end point (EP) of run muf with splabs. To substitute the output of splabs for *rlab* we use a simple form of command substitution.

Since we want to change the type of continuation from equilibrium point to fold continuation we have to perform *branch switching* by setting the AUTO constant ISW to 2. Note that ISW is a special constant in the sense that rauto always initializes ISW to 1 (normal setting) regardless of its value in a constants file. Values other than 1 must be specified on the command line.

The second script `scandemo` demonstrates the so-called *multiple run mode* of rauto for performing so-called *scans*. It executes the rauto commands

```
rauto cusp muf -UZR 2 -1
rauto cusp mub -DS -

rrun=muf
rauto cusp LPf $rrun -IRS $(splabs $rrun LP 1 1) -ISW 2
rauto cusp LPb $rrun -IRS $(splabs $rrun LP 1 1) -ISW 2 -DS -

rrun=muf
rauto cusp laf $rrun -IRS $(splabs $rrun UZ 1 1) -ICP 1 2 \
  -UZR 1 -2:0.1:2
rauto cusp lab $rrun -IRS $(splabs $rrun UZ 1 1) -ICP 1 2 \
  -UZR 1 -2:0.1:2 -DS -

rrun=laf
rauto cusp mu%02dff $rrun -IRS $(splabs $rrun UZ) -ICP 2 1
rauto cusp mu%02dfb $rrun -IRS $(splabs $rrun UZ) -ICP 2 1 -DS -
rrun=lab
rauto cusp mu%02dbf $rrun -IRS $(splabs $rrun UZ) -ICP 2 1
rauto cusp mu%02dbb $rrun -IRS $(splabs $rrun UZ) -ICP 2 1 -DS -
```

The first four commands are almost as before. However, the first run muf is recomputed, because a so-called *user output point* is defined with the option '`-UZR 2 -1`', which can be read as 'save restart data whenever PAR(2) assumes the value -1'. The general form of defining user output points is

`-UZR` [(*inum drange*) ...]

drange : *dnum* | *begin*[:*step*]:*last*

Inum is the index of a parameter and *drange* is either a double precision number or a list of double precision numbers evenly distributed within the interval [*begin*, *end*] with spacing *step,* which is 1.0 by default.

The fifth and sixth run start at this user output point and perform an equilibrium point continuation in λ for $\mu = -1$. These runs produce a large number of further user output points for different values of λ. Note that you may need to recompile AUTO with NUZRX set to a sufficiently large value. The user points computed in these runs serve as starting points for a sequence of continuations in μ, scanning a two-dimensional manifold of equilibrium points above the (λ, μ) parameter plane; see Figure 6. This sequence of continuations is executed with the last four commands, where we use the multiple run mode of rauto.

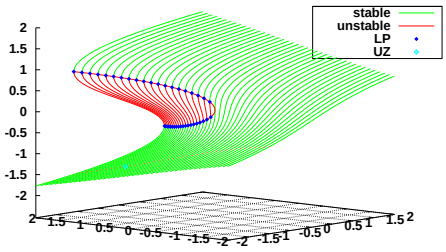


Figure 6: `plotbd -i 1:4:3 "[-2:2]" "[-2:2]" ‘lsruns "mu*"‘ LPf LPb`

In multiple run mode, that is, when a list of more than one restart label for the same restart run is specified the run name must contain a printf %d format specifier of the general form `%`[`0`[*width*]]`d`. The optional `0` forces zero-padding and *width* sets an optional field width for printing a zero-padded decimal integer number. Rauto will compute a *qualified run name* with the actual restart label substituted according to the format specifier. This creates a unique name for each run of a scan. Use lsruns as shown in the caption of Figure 6 to pass all qualified run names matching a certain pattern to plotbd. Note that this example makes use of parameter overspecification. All runs plotted in Figure 6 used the option '`-ICP 2 1`' which includes both parameters in the output data such that these runs can easily be combined in a single plot.

Demo duff: harmonic forcing and starting from numerical data

To create a working copy of the demo duff change to a directory where you would like to do your computations, for example, `${HOME}/rauto`. Then type '`rdm duff`'. This will create the sub-directory `duff`, which contains the files `duff.f`, `duff.dat`, `r.duff` and `demo`. Change to the directory duff.

The Duffing equation

$$\ddot{x} + \alpha x + \lambda \dot{x} + \varepsilon x^3 = A \cos \omega t$$

considered in this example is not autonomous, but periodically forced with period of $T = 2\pi/\omega$. However, the forcing is harmonic with a single frequency, which allows to transform this equation into an autonomous equation by augmenting it with an asymptotically stable harmonic oscillator. This procedure leads to the four-dimensional first order system

$$\begin{aligned} \dot{x} &= y, \\ \dot{y} &= Av - \alpha x - \lambda y - \varepsilon x^3, \\ \dot{u} &= u + \omega v - u(u^2 + v^2), \\ \dot{v} &= -\omega u + v - v(u^2 + v^2), \end{aligned}$$

which is implemented in the equations file `duff.f`. The file `duff.dat` contains a numerical solution of this system over approximately one period for the parameter values $A = 0.4$, $\omega = 1.0$, $\lambda = 0.04$, $\alpha = -0.2$ and $\varepsilon = 8/15$.

The master constants file is `r.duff`, which is set up for a continuation in PAR(2) $= \omega \in [0, 3]$. The file `demo` is an executable shell script for the command interpreter bash serving as a demonstration program, which executes the `rauto` commands

```
rfcon duff

rauto duff omf dat -IRS 1
rauto duff omb dat -IRS 1 -DS - -NMX 300

rauto duff po1 omb -IRS ‘splabs omb BP 1 1‘ -ISW -1 -NTST 30 -DSMAX 0.05

rauto duff po2 po1 -IRS ‘splabs po1 PD 1 1‘ -ISW -1 \
  -NTST 60 -DS 0.01 -DSMAX 0.02 -NMX 1000
rauto duff po3 po2 -IRS ‘splabs po2 PD 1 1‘ -ISW -1 \
  -NTST 100 -DS 0.001 -DSMAX 0.005 -NMX 4000
```

The first command performs the only initial run using the utility program `rfcon`. It converts the numerical data from file `duff.dat` into the internal data format used by AUTO. In general, running rfcon requires the three files *name*`.f`, *name*`.dat` and `r.`*name*. The function STPNT in the equations file *name*`.f` must initialize the free parameters. The file *name*`.dat` contains columns with sufficiently dense data points in t, U(1), ..., U(NDIM). The file `r.`*name* provides values for the dimension NDIM and certain discretization constants. The numerical data is always converted into a solution with label 1 of the run ‘dat’.

The subsequent commands in the script perform in this order

- continuations forwards and backwards in ω,
- branch switching at a branch point (BP),
- branch switching at period-doubling points (PD),

computing the first two branches of a period-doubling sequence; see Figures 7 (a) and (b). Use the command ‘`plotsol` *run* [*lab* ...]’ to plot all or selected solutions of a run as shown in Figures 7 (c) and (d).

Demo roess: constants inheritance

To create a working copy of the demo roess change to a directory where you would like to do your computations, for example, `${HOME}/rauto`. Then type ‘`rdm roess`’. This will create the sub-directory `roess`, which contains the files `roess.f`, `r.roess` and `demo`. Change to the directory roess.

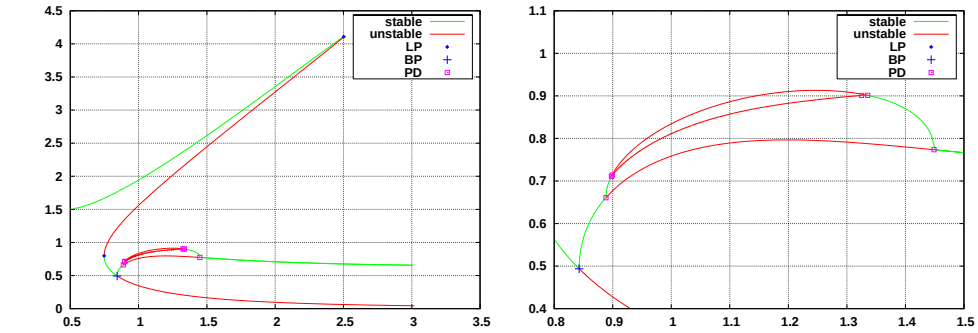
The equations file `roess.f` defines the right hand side of the Rössler equation

$$\begin{aligned} \dot{x} &= -y - z, \\ \dot{y} &= x + \alpha y, \\ \dot{z} &= \beta + z(x - \gamma). \end{aligned}$$

and provides an initial equilibrium point in STPNT for $\alpha = 0.005$, $\beta = 0.25$, $\gamma = 6.2$. The master constants file `r.roess` is set up for a continuation in PAR(1) $= \alpha \in [0, 0.4]$. The file `demo` is a script for the command interpreter bash and demonstrates `rauto`’s mechanism of *constants inheritance.*

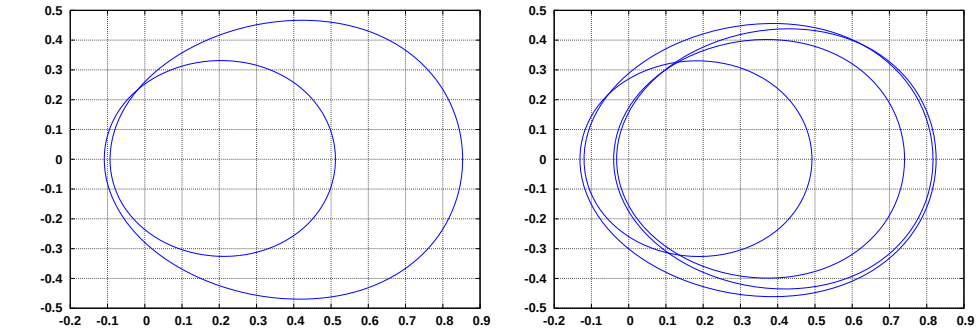
Constants inheritance creates a dependency of a restarted or *child run* on its restart or *parent run.* The objective is to trigger a recomputation of a child run whenever its restart solution in the parent run might have changed, including changes in the discretization. This feature is essential, for example, for

- adding commands to a script (avoid unnecessary recomputations),



(a) `plotbd 1:3 ‘lsruns "om*" "po*"‘`

`plotbd 1:3 "[0.8:1.5]" "[0.4:1.1]" ‘lsruns "om*" "po*"‘` (b)



(c) `plotsol po2 17`

`plotsol po3 31` (d)

Figure 7: Bifurcation diagram (a) of the Duffing equation and an enlargement (b) around a period-doubling sequence. The period-doubled (c) and quadrupled (d) solution around $\omega \approx 1.02$.

- scans (number and location of UZR restart solutions may change) and
- switching between coarse and fine grids (changing master constants file only).

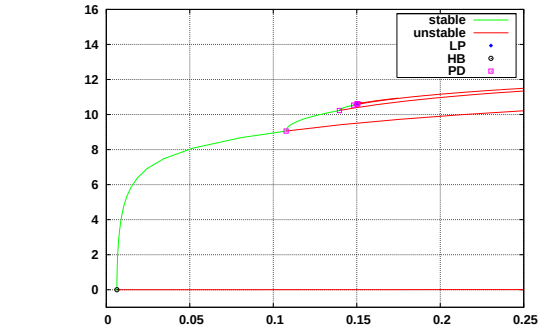
It is also useful in situations where you want to change a certain AUTO constant depending on its value for the parent run as the demonstration program illustrates. Use the command ‘`printac` *run ac_name*’ to print the value that was assigned to AUTO constant *ac_name* during run *run*. For the experts: note that it is indeed more natural to use shell scripts rather than the command make as we have a top-down dependency and make is designed to resolve bottom-up dependencies. Furthermore, the dependency check employed by rauto is more accurate, because it is based on the contents of certain files as opposed to using just the modification date. The only rule imposed on shell scripts is that the command for a child run must be executed after the command of its parent run.

The demonstration program computes a variable number of steps of a period-doubling cascade occurring in Rössler's equation. It uses a loop over the index of the step and makes full use of constants inheritance for doubling the number of mesh points and reducing the initial, minimal and maximal step size by a factor of 4.67 every iteration. The full command for calling the demonstration program is ‘`demo` [*steps*]’, where the optional argument *steps* specifies how many steps of the period-doubling sequence shall be computed, default is 3. The author was able to go up to *step* = 7, which seems to be the accuracy limit of AUTO. Note that you might need to recompile AUTO with NTSTX and NCOLX set to sufficiently high values; see [1] for details. The demonstration program executes the rauto commands

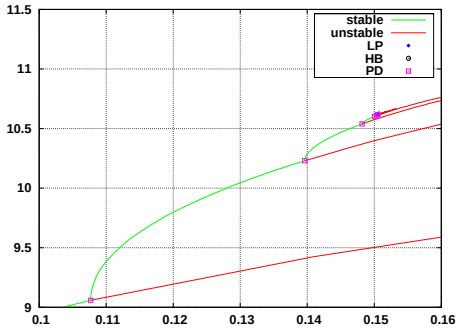
```
rauto roess alf

rrun=alf
```

```
rauto roess po1 $rrun -ICP 1 11 -IRS ‘splabs $rrun HB‘ \
  -DS 0.1 -DSMAX 0.5 -ISW -1 -IPS 2 -NPR 1

for (( i=2 ; $i <= ${1:-3} ; ++i )) do

  # names of run and restart run
  run=po$i
  rrun=po$(( $i - 1 ))

  # read step sizes from restart run
  h0=$(printac $rrun DS )
  hmin=$(printac $rrun DSMIN )
  hmax=$(printac $rrun DSMAX )

  # reduce step sizes and double NTST
  h0=$( dc -e "8 k $h0 4.67 / p" )
  hmin=$( dc -e "8 k $hmin 4.67 / p" )
  hmax=$( dc -e "8 k $hmax 4.67 / p" )
  NTST=$(( 2*$(printac $rrun NTST) ))

  # print current settings and perform run
  echo "NTST=$NTST, DS=$h0, DSMIN=$hmin, DSMAX=$hmax"
  rauto roess $run $rrun -ICP 1 11 -IRS ‘splabs $rrun PD‘ \
    -DS $h0 -DSMIN $hmin -DSMAX $hmax -NTST $NTST \
    -ISW -1 -IPS 2 -NPR 5
done
```

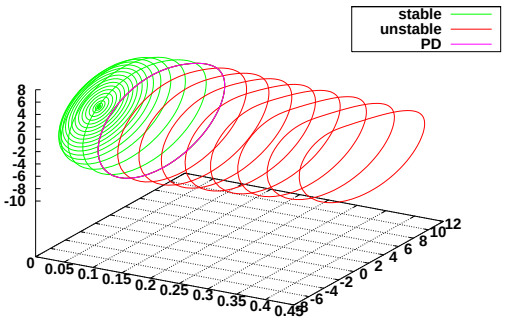
Note the use of *default value substitution* `${1:-3}`, *command substitution* `$(...)` and *arithmetic expansion* `$((...))`. Use the commands `plotbd` and `plotsol` as shown in Figure 8 to plot the bifurcation diagram and families of solutions.



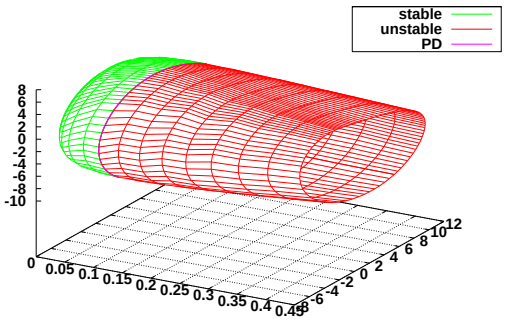
(a) `plotbd 1:3 "[0:0.25]" "[-1:16]" ‘lsruns "*"‘`



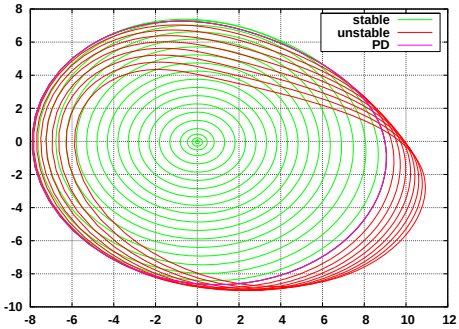
`plotbd 1:3 "[0.1:0.16]" "[9:11.5]" ‘lsruns "*"‘` (b)



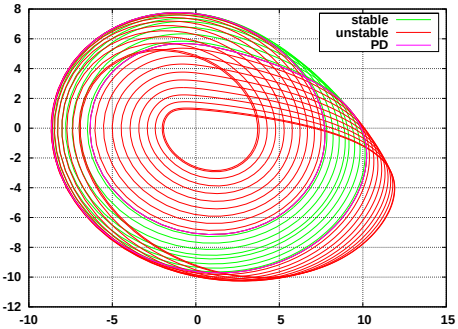
(c) `plotsol -bl 2 0:2:3 po1`



`plotsol 0:2:3 po1` (d)



(e) `plotsol po1`



`plotsol po2` (f)

Figure 8: Bifurcation diagram (a) of the Rössler system and an enlargement (b). The family of period-one solutions emerging in a Hopf-bifurcation shown as individual solutions (c) and as a two-dimensional manifold (d) in (α, x, y) space. Projection of the period-one (e) and the period-doubled (f) solutions onto the (x, y) plane.

A Command reference

All utility programs that are part of the package rauto, except rfcon, accept filenames and standard input as well as run names. Hence, they can be used independent of rauto, for example, to process computational results from the pre-rauto era. If you are not interested in rauto you might still want to have a look at the tools provided.

Rauto - execute AUTO continuation runs.

Rauto is a command line interface to AUTO.

Usage

`rauto` [`-h`|`-f`] *name run* [*restart_run* `-IRS` *irange* ...] [*options* ...]

options : (`-`*iopt inum*) | (`-`*dopt dnum*) | *spopt* | *lopt*
iopt : `NDIM` | `IPS` | `ILP` | `NTST` | `NCOL` | `IAD` | `ISP` | `ISW`
| `IPLT` | `NBC` | `NINT` | `NMX` | `NPR` | `MXBF` | `IID` | `ITMX`
| `ITNW` | `NWTN` | `JAC` | `IADS`
dopt : `RL0` | `RL1` | `A0` | `A1` | `EPSL` | `EPSU` | `EPSS` | `DSMIN` | `DSMAX`
spopt : (`-ICP` *inum* ...) | (`-DS` (*dnum*|`+`|`-`))
lopt : ((`-THL`|`-THU`) (*inum dnum*) ...) | (`-UZR` (*inum drange*) ...)
irange : *inum* | *inum*`:`[*inum*`:`]*inum*
drange : *dnum* | *dnum*`:`[*dnum*`:`]*dnum*

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-f` : Force execution of computation even if the data seems up-to-date.

Arguments

name : Base name of the equation file *name*`.f`.
run : Unique name (identifier) of the run to be performed.
restart_run : Name of the run that contains the restart solutions.

In single-run mode (one restart label) *run* may contain, and in multiple-run mode (two or more restart labels) *run* must contain a %d format specifier *f* : `%`[`0`[*width*]]`d`. This specifier is replaced with the restart label of the current run. *Width* sets the minimum field width and the initial '`0`' forces zero-padding. For example, '`%03d`' will result in '007' for restart label 7. The run name resulting from this substitution is called the *qualified run name*, for example, '`agent007`' is the qualified run name of the run name '`agent%03d`' with restart label 7.

Options

Options is a list of AUTO constants preceeded with a '`-`'. The meaning of these constants is briefly explained in Section 2 and Appendix B and in detail in the AUTO manual [1]. Upper- and lower case are equivalent and can be mixed. The majority of options requires an integer argument *inum*, or, a floating point argument *dnum*, which will be assigned to the corresponding AUTO constant. Exceptions to this rule are `-IRS`, the special options `-ICP` and `-DS`, and the list options `-THL`, `-THU` and `-UZR`.

IRS expects a list of restart labels and accepts integer numbers and/or integer ranges. The notation of ranges follows the matlab syntax *irange* : *begin*:[*step*:]*end*, that is, '`1:5`' expands to '`1 2 3 4 5`' and '`1:2:5`' to '`1 3 5`'. ICP expects a list of indices of continuation parameters. Note that you must *not* give the number of parameters. The initial continuation step size DS

accepts the two direction flags '`+`' and '`-`' besides an explicit floating point value. Specifying a direction flag will set the direction of continuation to forward or backward, respectively, without changing the absolute value of the initial step size.

THL and THU follows a list of pairs consisting of an integer *index* and a floating-point value *weight*. *Index* is the index of a parameter or of a state variable and *weight* is the scaling factor that will be applied to this variable in the arc-length computation. UZR expects pairs consisting of an integer *index* and a floating-point value *uzr_value* similarly to THL and THU. However, ranges of floating-point values may be given for *uzr_value*. The syntax is identical to integer ranges. For example, the pair '`1 1:0.5:2`' expands to the three-element list of pairs '`1 1.0 1 1.5 1 2.0`'.

Detailed description

Rauto combines into a single invocation a number of basic steps that are necessary to perform a run with AUTO, which makes it easier to execute and organize large numbers of runs. Before using rauto you need to prepare an equations file *name*`.f`, defining the right-hand side of your system and an initial solution, and the *master constants file* `r.`*name* with initial values for all AUTO constants; see Section 2 for details. If you want to start from numerical data you also have to provide the file *name*`.dat` that contains points along a solution; see the description of `rfcon` for details.

AUTO computations are divided into two phases: a few initial runs of rauto with restart label IRS and *restart_run* omitted, and subsequent restarted runs where *restart_run* and restart labels are specified. When starting from numerical data only a single initial run is performed with `rfcon`. At this point we describe initial runs when using rauto, an initial run using `rfcon` is explained in the command description of `rfcon`. On startup of an initial run all AUTO constants are initialized from the master constants file `data/r.`*name* and from the constants file `data/r.`*restart_run* for a restarted run, which means that a restarted run inherits all constants from its restart run. There are three exceptions to this rule: ISW is always set to 1, DS is always initialized with a positive value and the user list UZR is always set to an empty list, only UZR points given on the command line will have the desired effect. After initialization rauto checks for which AUTO constants new values were given on the command line and substitutes all of these. Rauto then enters a loop over all restart labels that were specified on the command line, or IRS=0 for an initial run.

For each restart label rauto checks if the files `data/r.`*qrun* and `data/l.`*qrun* already exist, where *qrun* is the qualified run name; see Section Arguments above. Existence of these files indicates that this particular run has been computed before. Three tests are performed to decide whether or not it is necessary to rerun the computation. The computation is assumed to be up-to-date if the equations file is older than `data/r.`*qrun*, if all AUTO constants are identical to the ones in `data/r.`*qrun* and if the output of '`printls` *restart_run restart_lab*' is identical to `data/l.`*qrun*. If one of these tests fails or if `data/r.`*qrun* does not exist, then the computation is executed. Otherwise, a message indicating that the run is up-to-date is printed on the screen. Since AUTO shows a certain lack of communication skills rauto employs a heuristic test to verify hat the computation was successful. If this test fails an error message is printed. On success rauto saves all computed data and creates or updates the files `data/r.`*qrun* and `data/l.`*qrun*.

Return values

Rauto returns 0 on success and 1 if an error occurred or the `-h` switch was given. In the case that an error occurs during a run of AUTO, or if the run of AUTO is interrupted, the file `data/r.`*qrun* will be deleted, which schedules the interrupted run for recomputation.

Use 'set -e' in sh or bash scripts to stop a sequence of computations as soon as an error occurs.

Bd2dat - convert and combine branch data.

Bd2dat converts branch data data into a plain text format and supports splitting the data into stable and unstable parts as well as extraction of special points. The output format follows gnuplot's 'plot data-file' convention and the splitting of files allows stability plots as well as marking special points.

Usage

`bd2dat` [`-h`|`-a`] [`-bl` *n*] (*run*|*pfile*|`-`) [*ofile1* [*ofile2*]] [(`-`*sptype spfile*) ...]

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-a` : Append data to output files, do not overwrite.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.
ofile1 : Write or append stable part or all to file *ofile1*.
ofile2 : Write or append unstable part to file *ofile2*.
sptype : `LP` | `BP` | `HB` | `PD` | `TR` | `UZ` | `RO` | `MX`
spfile : Write or append special points of selected type to file *spfile*.

Options

`-bl` *n* : # of Blank lines to separate branches, 1=mesh data, 2=individual.

Detailed description

Bd2dat converts, combines and splits branch data from AUTO output format into a plain ASCII format that can be used as input to gnuplot's 'plot data-file' commands. It either reads the data from the run with qualified run name *run*, from an explicitly specified file *pfile*, or, from standard input if '`-`' was given. The last variant is used when filtering branch data with filterbd; see the description of filterbd below. The data of the branch is written or appended to file *ofile1*, depending on whether bd2dat was invoked with the switch `-a` or not. If both, *ofile1* and *ofile2* are present, then bd2dat performs a stability splitting, that is, the stable parts of a branch are written or appended to *ofile1* and the unstable parts to *ofile2*. If no output file is given, then no output of branch data is produced. If an option of the type `-`*sptype spfile* was specified, bd2dat extracts a list of special points of type *sptype* to file *spfile*.

Bd2dat splits the branch data of a single run into records. A record represents a connected component of a single branch, in the simplest case it is the entire branch (stability splitting off). These records can be combined in files (switch `-a`) to make plotting easier and will be separated by a number of blank lines as specified with the option `-bl`, default is 2, that is, gnuplot will interpret each record as an individual curve. The same applies to the special points files, where a record is a complete list of special points of a particular type along the branch, however, stability splitting is never performed for records of special points.

In the case that you want to plot a surface made up from several branches, make sure that each record has the same number of points and use `-bl 1`, gnuplot will then interpret the resulting output as mesh data. For example, if you switch detection of special points off, then the branch data will contain exactly NMX points, if the run did not terminate prematurely with MX. Several runs with the same value for NMX might then be combined for a surface plot.

Return values

Bd2dat returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Fam2dat - convert and combine families of solutions.

Fam2dat converts the data of families of solutions into a plain text format. The output format follows gnuplot's 'plot data-file' convention.

Usage

`fam2dat` [`-h`|`-a`] [*options* ...] *source* [*ofile1* [*ofile2*]] [(`-`*sptype spfile*) ...]

options : (`-bl` *n*) | (`-p` *col*)
source : *run* | (*pfile*|`-` *qrun*|*qfile*)

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-a` : Append data to output files, do not overwrite.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.
qrun : Read solutions from run with qualified run name *qrun*.
qfile : Read solutions from `q.*`-file *qfile*.
ofile1 : Write or append stable solutions or all to file *ofile1*.
ofile2 : Write or append unstable solutions to file *ofile2*.
sptype : `LP` | `BP` | `HB` | `PD` | `TR` | `UZ` | `RO` | `MX`
spfile : Write or append solutions at special points to file *spfile*.

Options

`-bl` *n* : # of Blank lines to separate solutions, 1=mesh data, 2=individual.
`-p` *col* : Prepend solution data with values from column *col* of the branch data, column PAR is *col*=1.

Detailed description

Fam2dat is the counterpart of bd2dat for solutions and we focus here on the differences between these two tools; see also the description of bd2dat. Fam2dat converts, combines and splits families of solutions from AUTO output format into a plain ASCII format that can be used as input to gnuplot's 'plot data-file' commands. In order to do so fam2dat requires *two* input files, one file with the branch data and one with the solutions of the same run. In the simplest case only the qualified run name *run* is specified and fam2dat uses the files of that run. If you want to use a different source of branch data, for example, the output of filterbd, then you must also specify a source of the solution data, either a qualified run name *qrun*, or, a `q.*`-file name *qfile*.

Fam2dat allows to include an arbitrary number of values of selected columns from the branch data into the solution data. This enables the plotting of families of solutions versus the continuation parameter. For example, a call like

```
fam2dat -p 1 ...
```

will include the continuation parameter in the solutions, that is, the output will have the columns PAR T U(1) ... U(NDIM).

Return values

Fam2dat returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Filterbd - filter branch data.

Filterbd removes rows from branch data, for which either a column becomes a constant multiple of π, or, the points have solution labels outside a specified range.

Usage

`filterbd` [`-h`] (*run*|*pfile*|`-`) (`-tr` *col*) | (`-l` *begin* [*end*])

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.

Options

`-tr` *col* : Filter TR curve with respect to values in column *col*, `PAR` is column 1.
`-l` *begin* [*end*] : Print all points between labels in the range [*begin*,*end*].

Detailed description

Filterbd has two principal operation modes: filtering curves of Neimark-Sacker bifurcations and cutting branch data. The computation of a locus of Neimark-Sacker bifurcations (torus bifurcations) sometimes jumps onto a period-doubling or a fold curve. This can be detected by including PAR(12) into ICP, because [PAR(12) mod π] = $2\pi\varrho$, where ϱ is the rotation number of the emerging invariant circle. If PAR(12) becomes a constant multiple of π, then we approach a 1/1 or 1/2 resonance where the Neimark-Sacker curve generically ends, but not necessarily the continuation. Filterbd replaces those rows of a bifurcation diagram with a single empty line for which this condition is satisfied, eliminating the segments of a Neimark-Sacker continuation that are not part of the actual Neimark-Sacker curve. Use a call like

`filterbd` *run* `-tr` *col* `| bd2dat - ...`

for correct conversion of Neimark-Sacker curve data.

In its second form filterbd can be used for cutting branch data. Filterbd will print all points that have or follow a solution point with label *begin* up to and including a point with solution label *end*. If *end* is omitted, then all points until the end of the branch data, starting with the point with label *begin* will be printed. This function can be used to print only a section of branch data:

`filterbd` *run* `-l` *begin* [*end*] `| printbd -`

or to preprocess branch data for fam2dat. For example, to extract all labeled solutions up to, and including the first period-doubling point use (in bash syntax):

`filterbd` *run* `-l 1 $(splabs` *run* `PD 1 1) | fam2dat -` *run* `...`

All utilities that come with `rauto`, including filterbd, are able to handle output generated by filterbd. Hence, it is possible to filter multiple times in one pipeline.

Return values

Filterbd returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Lsruns - list runs.

Lsruns prints on screen a list of all runs that have been computed.

Usage

`lsruns` [`-h`] [`"`*pattern*`"` ...]

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

pattern : List all runs that match the shell pattern *pattern*.

Environment

Lsruns honors the environment variables LSRUNS_COLS and LSRUNS_FW, which influence the formatting of lsruns' output. LSRUNS_COLS sets the maximal number of columns per line (default is 6) and LSRUNS_FW sets the field width per run name (default is 10).

Detailed description

Lsruns prints a list of runs that match a shell pattern, that is, a pattern containing the wildcard characters '`*`' and '`?`'; see the manual of [t]csh for what else might be used. Note that a pattern must be enclosed in double-quotes to prevent premature expansion by the shell. If no pattern is given, then all runs will be printed.

Return values

Lsruns returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Plotbd - plot bifurcation diagram.

Plotbd plots bifurcation diagrams with gnuplot.

Usage

`plotbd` [`-h`|`-i`] [*options ...*] [*cols*] [*ranges*] (*run*|*pfile*|`-`) ...

options	:	`-bw` | (`-k` *name*) | (`-eps` *eps-name*)
cols	:	*x*`:`*y*[`:`*z*]
ranges	:	`"`*xrange*`"` [`"`*yrange*`"` [`"`*zrange*`"`]]
(*x*|*y*|*z*)*range*	:	`[`*a*`:`*b*`]`

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-i` : Interactive, do not exit from gnuplot, but prompt for input.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.

Options

`-bw`	:	Plot black and white.
`-k` *name*	:	Keep, save gnuplot file as *name*`.gp` and data files as *name*`*.dat`, `*` = `s`, `u`, `LP`, `BP`, `HB`, `PD`, `TR`, `UZ`, `RO`.
`-eps` *eps-name*	:	Plot to eps-file *eps-name*`.eps` instead of screen.

Detailed description

Plotbd uses gnuplot to plot a bifurcation diagram using the branch data from the runs or files specified on the command line. Exactly one '`-`' may be given to indicate standard input. A two or three dimensional bifurcation diagram is plotted depending on how many columns have been specified, default are the two columns 1 (PAR) and 2 (L2-NORM). The definition of columns and plotting ranges follows gnuplot's syntax, that is,

```
plotbd 1:8:2 "[-1:1]" "[]" "[1:10]" ...
```

uses columns 1, 8 and 2 as x, y and z axis and defines a plotting range for each coordinate, where `"[]"` stands for 'automatic.' If you call plotbd with the `-i` switch gnuplot will prompt for input. You can then enter any gnuplot commands, for example, to change the plotting ranges or the view. Refresh the graphics with the gnuplot command `replot`. If mouse support is enabled you can zoom in or rotate the graphics interactively.

Plotbd uses green lines for stable and red lines for unstable parts of a branch. If you prefer a more traditional form of stability plot, then the option `-bw` will create a black-and-white plot with solid lines for stable and dashed lines for unstable parts.

The option `-eps` *eps-name* will direct the output to the encapsulated postscript file *eps-name*`.eps`. Note that *eps-name* may contain sub-directories, but should not contain the extension `.eps`. The figures created by plotbd are intended for quick inspection of computation results and for supporting the selection of restart points, but not for publication (except this one). However, the intermediate files created by plotbd may serve as a starting point for producing publishable pictures. If you specify the option `-k` *name*, then all intermediate files will be

saved as files with a name matching *name*`.*`. Therefore, *name* should point to a sub-directory and should not contain any extension. For example,

```
plotbd -k bddata/bd -eps pics/bd ...
```

will save all intermediate files to the sub-directory bddata, which must be created before calling plotbd. The most important file in this set is `bddata/bd.gp`, which is the gnuplot file containing instructions for creating the data-files and, more importantly, the eps-file `pics/bd.eps`. You should move this gnuplot file to a different location to prevent accidental overwriting, and edit it to accommodate your ideas. Simply call '`gnuplot` *gnuplot-file*' to update `pics/bd.eps`.

Note that plotbd is quite powerful when used together with lsruns. A command like

```
plotbd 1:8:2 `lsruns "mu*"` ...
```

will create a three-dimensional bifurcation diagram using the data from all runs with qualified run names that match the pattern `mu*`.

Return values

Plotbd returns 0 on success and 1 if an error occurred or the `-h` switch was given. Error checking is quite sloppy. Plotbd will fail if called with incorrect options/arguments, but the issued error message might not be terribly helpful.

Plotsol - plot solutions.

Plotsol plots solutions with gnuplot.

Usage

`plotsol` [`-h`|`-i`] [*options ...*] [*cols*] [*ranges*] *FPM*|*SPM*

options	:	`-bw` | (`-k` *name*) | (`-eps` *eps-name*) | (`-bl` *n*)
cols	:	*x*`:`*y*[`:`*z*]
ranges	:	`"`*xrange*`"` [`"`*yrange*`"` [`"`*zrange*`"`]]
(*x*|*y*|*z*)*range*	:	`[`*a*`:`*b*`]`
FPM	:	*run* | (*pfile*|`-` *qrun*|*qfile*)
SPM	:	(*qrun*|*qfile*|`-`) *lab ...*

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-i` : Interactive, do not exit from gnuplot, but prompt for input.

Arguments

run	:	Read branch data and solutions of run with qualified run name *run*.
qrun	:	Read solutions of run with qualified run name *qrun*.
pfile	:	Read branch data from `p.*`-file *file*.
qfile	:	Read solutions from `q.*`-file *file*.
`-`	:	Read branch data (*FPM*) or solutions (*SPM*) from standard input.
lab	:	Plot solutions with labels *lab ...*

Options

`-bw`	:	Plot black and white.
`-k` *name*	:	Keep, save gnuplot file as *name*`.gp` and data files as *name*`*.dat`, `*` = `s`, `u`, `LP`, `BP`, `HB`, `PD`, `TR`, `UZ`, `RO`.
`-eps` *eps-name*	:	Plot to eps-file *eps-name*`.eps` instead of screen.
`-bl` *n*	:	# of Blank lines to separate solutions, 1=mesh data, 2=individual.

Detailed description

Plotsol has two principal operation modes: a *family plotting mode* that is used if *no* solution labels are specified, and a *solutions plotting mode* that is used if solution labels *are* present on the command line.

In family plotting mode plotsol is similar to MatCont's function plotcycle. It uses gnuplot to plot a family of solutions from the run with qualified run name *run*. Input data may also be specified by providing either two filenames or a file name and a run name. The first file must be a `p.*`-file containing branch data and may be set to standard input using '`-`' as file name. The second file must be a `q.*`-file containing solution data. A stability splitting using fam2dat is performed and the continuation parameter is included in the data as column 0. By default a plot of column 3 versus column 2, that is, U(2) versus U(1) is shown. The selection of columns and plotting ranges follows gnuplot's syntax, that is,

```
plotsol 0:2:3 ...
```

will plot the projection of a solution to its first two components (U(1),U(2)) versus the continuation parameter. Use filterbd to plot only part of a family, for example, in bash notation

```
filterbd run -l 1 $(splabs run PD 1 1) | plotsol 0:2:3 - run ...
```

will plot a family of periodic solutions from the first labeled solution up to and including a solution at the first period-doubling bifurcation. If the parameter (column 0) is given for the x, y, or z axis, then the family of solutions is plotted as a surface, otherwise, the solutions are plotted as individual curves. This behavior can be changed with the option `-bl` n; see section 'Common switches/options' below.

Plotsol uses green lines for stable, red lines for unstable and various other colors for special solutions. If you prefer a more traditional form of stability plot, then the option `-bw` will create a black-and-white plot with solid lines for stable, dashed lines for unstable and different markers for special solutions.

In solutions plotting mode plotsol is similar to PLAUT's 2d and 3d plotting modes. It uses gnuplot to plot solutions from the run with qualified run name *qrun*, from `q.*`-file *qfile*, or from standard input if '`-`' was given. The input must be solution data. In bash notation a command like

```
plotsol run $(splabs run UZ)
```

will plot all solutions at UZR output points from run with qualified run name *run*. By default a plot of column 3 versus column 2, that is, U(2) versus U(1) is shown. The selection of columns and plotting ranges follows gnuplot's syntax, that is,

```
plotsol 1:2 "[0:0.5]" ...
```

will plot U(1) versus time over half a period.

Common switches/options. If you call plotsol with the `-i` switch gnuplot will prompt for input. You can then enter any gnuplot commands, for example, to change the plotting ranges or the view. Refresh the graphics with the gnuplot command `replot`. If mouse support is enabled you can zoom in and rotate the graphics interactively.

The option `-bl` n will override any default interpretation of the plotting data to: mesh data if $n = 1$, or, individual curve data if $n \geq 2$. Default is $n = 2$ except if the parameter (column 0) is given for one of the x, y or z axis in family plotting mode, in which case the default is $n = 1$.

The option `-eps` *eps-name* will direct the output to the encapsulated postscript file *eps-name*`.eps`. Note that *eps-name* may contain sub-directories, but should not contain the extension `.eps`. The figures created by plotsol are intended for a quick overview over the solutions, but not for publication (except this one). However, the intermediate files created by plotsol may serve as a starting point for producing publishable pictures. If you specify the option `-k` *name*, then all intermediate files will be saved as files with a name matching *name*`.*`. Therefore, *name* should point to a sub-directory and should not contain any extension. For example,

```
plotsol -k bddata/pos -eps pics/pos ...
```

will save all intermediate files to the sub-directory bddata, which must be created before calling plotsol. The most important file in this set is `bddata/pos.gp`, which is the gnuplot file containing instructions for creating the data-files and, more importantly, the eps-file `pics/pos.eps`. You should move this gnuplot file to a different location to prevent accidental overwriting, and edit it to accommodate your ideas. Simply call '`gnuplot` *gnuplot-file*' to update `pics/pos.eps`.

Return values

Plotsol returns 0 on success and 1 if an error occurred or the `-h` switch was given. Error checking is quite sloppy. Plotsol will fail if called with incorrect options/arguments, but the issued error message might not be terribly helpful.

Printac - print AUTO constant.

Printac prints the value of the specified AUTO constant on screen.

Usage

`printac` [`-h`] (*run*|*rfile*|`-`) *ac_name*

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

run	:	Read constants of run with qualified run name *run*.
rfile	:	Read constants from `r.*`-file *rfile*.
`-`	:	Read constants from standard input.
ac_name	:	`NDIM` | `NTST` | `NCOL` | `NMX` | `NPR` | `RL0` | `RL1` | `A0` | `A1`
		| `EPSL` | `EPSU` | `EPSS` | `DS` | `DSMIN` | `DSMAX`

Detailed description

Printac prints on screen the value that was assigned to an AUTO constant during the run with qualified run name *run*. This might be useful for initializing certain computations, for example, if you want to compute a cascade of period-doubling bifurcations. If you use the bash, then a command like

```
rauto name run rrun -IRS $(splabs rrun PD 1 1) \
   -NTST $(( $(printac rrun NTST)*2 )) -ISW -1 ...
```

will initialize the computation of a doubled solution with twice the number of mesh points of the single solution. Note that you might also want to decrease the initial step size. You need to use the desktop calculator dc for that (again bash notation):

```
h0=$(printac rrun DS)
h0=$(dc -e "8 k 0 1 $h0 * 4.67 / p")
rauto ... -DS $h0 ...
```

Set DSMIN to a sufficiently small value. Note that we have to go to some length here to deal with negative values of DS.

Return values

Printac returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Printbd - print branch data.

Printbd prints the branch data of the specified run on screen.

Usage

`printbd` [`-h`] (*run*|*pfile*|`-`)

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.

Detailed description

Printbd prints on screen the branch data of the run with qualified run name *run* in a format similar to the output generated by AUTO during a computation. The listing includes all labeled solutions.

Return values

Printbd returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Printcols - print list of columns in branch data.

Printcols prints a list of columns of the branch data of the specified run on screen.

Usage

`printcols` [`-h`] (*run*|*pfile*|`-`)

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.

Detailed description

Printcols prints on screen a list of columns of the branch data of the run with qualified run name *run* in the format '*number* `:` *heading*'. Number can be used to refer to the column with heading *heading* wherever a column can be passed to an rauto command on the command line, for example, to the plotting utilities plotbd and plotsol.

Return values

Printcols returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Printls - print labeled solution entry.

Printls prints the branch data entry of a labeled solution.

Usage

`printls` [`-h`] (*run*|*pfile*|`-`) *lab*

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.
lab : Print entry of solution with label *lab*.

Detailed description

Printls prints on screen the entry that was created for a labeled solution in the branch data of the run with qualified run name *run*. This function is used internally by rauto.

Return values

Printls returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Printsp - print special points.

Printsp prints a list of special points of specified type on screen.

Usage

`printsp` [`-h`] (*run*|*pfile*|`-`) [*type*]

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.
type : `LP` | `BP` | `HB` | `PD` | `TR` | `UZ` | `RO` | `MX`

Detailed description

Printsp prints on screen a list of special points of the run with qualified run name *run* in a format similar to the output generated by AUTO during a computation. If *type* is omitted, then printsp behaves exactly like printbd – it is just about a microsecond slower.

Return values

Printsp returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Rdm - copy rauto demos.

Rdm creates a copy of an rauto demo in the local directory.

Usage

`rdm` [`-h`] *name*

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

name : Name of the demo.

Detailed description

Rdm creates a copy of the demo *name* in the local directory. A sub-directory with name *name* is created and all files of the demo are copied to this sub-directory. The demos that come with rauto are discussed in Section 3. Use the AUTO command @dm to copy demos that come with AUTO.

Return values

Rdm returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Rfcon - convert numerical data to run 'dat'.

Rfcon performs an initial run with numerical data.

Usage

`rfcon` [`-h`] *name*

Switches

`-h` : Display a usage message and exit with nonzero return value.

Arguments

name : Base name of the equations file *name*`.f`.

Detailed description

Rfcon performs an initial run with a numerically given start solution for your problem. In order to use rfcon you need to prepare an equations file *name*`.f` and a constants file `r.`*name* as explained in Section 2, as well as the data file *name*`.dat`. The first column of the data file is time and the subsequent columns are the values of the state variables in the order U(1),...,U(NDIM). The data must be given for a full period of a periodic solution, or over a full time interval for boundary value problems. In either case, PAR(11) will be initialized with PAR(11) $= t_{\text{end}} - t_{\text{begin}}$, where t_{begin} and t_{end} are the time values in the first and last row of your data file. Hence, it can be used for explicit rescaling of time. The initial parameters for your problem are defined as usual in the subroutine STPNT in the equations file.

Rfcon produces a run with name 'dat' containing one solution with label 1 that can be used to initialize subsequent restarted runs. The constants file of this run will be a copy of the master constants file `r.`*name*. It is not possible to specify new values for AUTO constants on the command line.

Return values

Rfcon returns 0 on success and 1 if an error occurred or the `-h` switch was given. Note that a return with success does not mean that the conversion was successful. Rfcon does not verify that the numerical data is indeed a suitable start solution for your problem.

Rmruns - remove runs.

Rmruns removes runs from your data.

Usage

`rmruns` [`-h`|`-i`|`-f`] ["*pattern*" ...] | (`-ra` *run* [*lab* ...])

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-i` : Interactive, prompt before removal of any matching run.
`-f` : Force, do not list or prompt.
`-ra` : Make `rmruns` behave symmetric to a call of rauto.

Arguments

pattern : Remove all runs that match the shell pattern *pattern*.
run : Name of run to remove, may contain one %d format specifier.
lab : Remove runs with restart labels *lab* ...

Environment

Rmruns honors the same environment variables as lsruns, namely LSRUNS_COLS and LSRUNS_FW, which influence the formatting of rmruns' intermediate output. LSRUNS_COLS sets the maximal number of columns per line (default is 6) and LSRUNS_FW sets the field width per run name (default is 10).

Detailed description

Rmruns has two calling forms, one symmetric to lsruns and the other symmetric to rauto (switch `-ra`). In the first form a list of shell patterns is specified, that is, patterns containing the wildcard characters '`*`' and '`?`'; see the manual of [t]csh for what else might be used. Note that a pattern must be enclosed in double-quotes to prevent premature expansion by the shell. Rmrun will delete any run with a qualified run name that matches one of these patterns. If no pattern is given, then rmruns will delete all runs.

In the second form a run name *run* and a list of restart labels is given. Rmruns will expand *run* to a list of qualified run names and delete all these runs. That is, if you want to delete all runs created with a command like

`rauto` *name run* `... -IRS 1 2 3 ...`

then

`rmruns` *run* `1 2 3 ...`

will do so.

Rmruns knows that loosing precious data might be a painful experience and will kindly ask for your permission before sending any run to the digital Hell. Anything except a 'yes', including 'Yes' and 'YES' will be interpreted as no. If you specify the `-i` switch rmruns will ask for each matching run separately, but here a 'y' is also accepted as a yes and you can exit rmruns with answering 'a' or 'abort'. This way you don't need to worry about too complicated patterns. However, if you are one of those guys who never make mistakes, then the `-f` switch is for you.

Return values

Rmruns returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Sol2dat - convert and combine solutions.

Sol2dat converts solution data into a plain text format. The output format follows gnuplot's 'plot data-file' convention.

Usage

`sol2dat` [`-h`|`-a`] [`-bl` *n*] (*run*|*qfile*|`-`) *ofile lab* ...

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-a` : Append data to output files, do not overwrite.

Arguments

run : Convert data of run *run*, may contain one %d format specifier.
qfile : Convert data from `q.*`-file *qfile*.
`-` : Convert data from standard input.
ofile : Write or append solutions to file *ofile*.
lab : Convert solutions with restart labels *lab* ...

Options

`-bl` *n* : # of Blank lines to separate solutions, 1=mesh data, 2=individual.

Detailed description

Sol2dat converts and combines solutions from AUTO output format into a plain ASCII format text file that can be used as input to gnuplot's 'plot data-file' commands. It either reads the data from the run with qualified run name *run*, from an explicitly specified file *qfile*, or, from standard input if '`-`' was given. The solutions with the specified labels are written or appended to file *ofile*, depending on whether sol2dat was invoked with the switch `-a` or not. The individual solutions will be separated by a number of blank lines as specified with the option `-bl`, default is 2, that is, gnuplot will interpret each solution as a separate curve. This is similar to the 2d and 3d plotting modes of AUTO's plotting utility plaut.

Return values

Sol2dat returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Splabs - print labels of special points.

Splabs prints a list of labels of special points of a selected type of a run. It is most useful for stable selection of restart labels of restarted runs.

Usage

`splabs` [`-h`|`-l`] (*run*|*pfile*|`-`) *type* [*begin* [*end*]]

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-l` : Interpret *begin* and *end* as labels, not as indices.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.
type : Print labels of special points of type *type*.
begin : Index of first label to print.
end : Index of last label to print.

Detailed description

Stable selection of restart labels really is an issue with AUTO and splabs is here to help. The author made the experience of changing labels the first time when repeating on an INTEL CPU a lengthy computation that involved many runs and was initially performed on a SUN SPARC workstation. The small differences in the floating point units were enough for a quite frustrating surprise. The usual "solution" is to set NPR=NMX, but this is not always desirable.

A more elegant and very robust solution is to select labels of a specific type and this is exactly what splabs does. In many cases this will even work if the equations or some parameters are changed slightly. A simple use of splabs selects all solution labels of type *type* of a run with qualified run name *run*. A subset of labels can be selected by giving values for *begin* and *end*, which will select all labels at positions $begin \leq i \leq end$ from the list of all labels. If *end* is omitted, then all labels at positions greater than or equal to *begin* will be selected. It is possible, but rarely helpful to select empty sets.

A more advanced application is to select all solution labels of a certain type between two labels of solutions of some different type. A natural candidate for the latter are limit points (LP) and end points (EP). For example, to select all UZR points between the first and second limit point use (in bash syntax)

```
-IRS $( splabs -l rrun UZ $(splabs rrun LP 1 2) )
```

Note the use of the `-l` switch in the outer call to splabs. This is needed for correct interpretation of the values returned by the inner call to splabs, which returns labels, not indices.

It is possible to do more complicated things using the abilities of arithmetic expansion provided by the bash. However, this might be unstable, that is, may lead to different or unexpected results when changing parameters or AUTO constants etc. The usage shown in the example above is probably the best way for robust and portable selection of restart labels.

Return values

Splabs returns 0 on success and 1 if an error occurred or the `-h` switch was given.

Sppars - print parameter values of special points.

Sppars prints a list of values from a specified column of branch data of a run for special points of a selected type.

Usage

`sppars` [`-h`|`-l`] (*run*|*pfile*|`-`) *type* [*col* [*begin* [*end*]]]

Switches

`-h` : Display a usage message and exit with nonzero return value.
`-l` : Interpret *begin* and *end* as labels, not as indices.

Arguments

run : Read branch data of run with qualified run name *run*.
pfile : Read branch data from `p.*`-file *pfile*.
`-` : Read branch data from standard input.
type : Print labels of special points of type *type*.
col : Select value from column *col*, `PAR` is column 1 (default).
begin : Index of first label to print.
end : Index of last label to print.

Detailed description

Sppars behaves in exactly the same way as splabs, except that it prints the values of column *col* of the branch data instead of the solution labels of special points. The column PAR is *col*=1 and column LAB is *col*=0.

Bugs

Is there any application?

Return values

Sppars returns 0 on success and 1 if an error occurred or the `-h` switch was given.

B AUTO constants

NDIM	Problem dimension
IPS	Problem type; 0=AE, 1=FP(ODEs), -1=FP(maps), 2=PO, -2=IVP, 4=BVP, 7=BVP with Floquet multipliers, 5=algebraic optimization problem, 15=optimization of periodic solutions
IRS	Start solution label
ILP	Fold detection; 1=on, 0=off
NICP	Continuation parameters
NTST	# mesh intervals
NCOL	# collocation points
IAD	Mesh adaption every IAD steps; 0=off
ISP	Bifurcation detection; 0=off, 1=BP(FP), 3=BP(PO,BVP), 2=all
ISW	Branch switching; 1=normal, -1=switch branch (BP, HB, PD), 2=switch to two-parameter continuation (LP, HB, TR)
IPLT	Select principal solution measure
NBC	# boundary conditions
NINT	# integral conditions
NMX	Maximum number of steps
RL0, RL1	Parameter interval $RL0 \leq \lambda \leq RL1$
A0, A1	Interval of principal solution measure $A0 \leq \|\cdot\| \leq A1$
NPR	Print and save restart data every NPR steps
MXBF	Automatic branch switching for the first MXBF bifurcation points if IPS=0, 1
IID	Control diagnostic output; 1=little, 2=normal, 4=extensive
ITMX	Maximum # of iterations for locating special solutions/points
ITNW	Maximum # of correction steps
NWTN	Corrector uses full newton for NWTN steps
JAC	User defines derivatives; 0=no, 1=yes
EPSL	Convergence criterion for equation parameters
EPSU	Convergence criterion for solution components
EPSS	Convergence criterion for special solutions/points
DS	Start step size
DSMIN, DSMAX	Step size interval $DSMIN \leq h \leq DSMAX$
IADS	Step size adaption every IADS steps; 0=off
NTHL, NTHU	list of parameter and solution weights
NUZR	list of values for user defined output

See the documentation of AUTO [1] for a detailed description.

References

[1] E. Doedel, A. Champneys, T. Fairgrieve, Y. Kuznetsov, B. Sandstede, and X. Wang. Auto97: Continuation and bifurcation software for ordinary differential equations (with HomCont). Technical report, Concordia University, 1997. URL: `http://cmvl.cs.concordia.ca/auto/`.