AT

Diego Bravo Estrada

An Eclectic Mathematical Software Tool

1 Introduction

AT was built as a general purpose software tool for solving common exercises of mathematical (and related) educational programs.

It consists of a general purpose Java mathematical library (resembling a subset of Apache Commons Math), and two frontends: the standard desktop (anywhere Java runs) and an Android "app". This document[1] describes the frontend language common to both environments (the "AT language" from here on.)

The AT language gets its syntax mainly from an Octave subset (or Matlab) and some bits from the C family.

Its aim was originally to develop some basic linear algebra functions with emphasis on exact arithmetic. Eventually it included basic symbolic power (specially for calculus operations); several numerical methods for root finding, ODEs, interpolation; and finally very basic graphical capabilities.

The "AT" name is a homage to Alan Turing.

1.1 Running AT

1.1.1 Desktop Computer[2]

It does require Java 1.7 or higher. From the command line just type:

```
java -cp at1.jar:automata.jar dbe.at.app.console.ConsoleMain
```

1.1.2 Android Device[3]

The mobile version does require Android 6 or higher. In that case, just click the icon to start.

[1] Updated for the AT release 11677.

[2] We don't provide installation instructions. The user should be able to install Java and put the AT jar files into a suitable folder.

[3] Again, no installation instructions are provided. The user should be able to install an APK file in spite of vendor restrictions.

1.2 Getting Help

The “help” command provides information about the available functionality which is distributed into several categories (like linear algebra, calculus, etc.) For help on a specific category, use "help <category>”; for example, “help la” does provide help about linear algebra.

2 AT Numeric Elements

The numeric elements in AT have several kinds, which include: integer, rational, real, ratroot and complex (and some more with special purpose.) Use the typeinfo() function to get information about the type of an element.

2.1 Integers

Plain integers of arbitrary size[4]. When a matrix is created, by default it is populated with integer zeroes. For example:

```
AT> ? 41+24;
65
AT> ? typeinfo(-900);
integer
AT> k=[1,3;4,5];
AT> ? typeinfo(k);
matrix
AT> ? typeinfo(k(1,2));
integer
```

Note the “?” (print) command used to display the results.

The following example shows the syntax for specifying numbers on different radix:

```
AT> ? 456(11)+412(12);
7a7(12)
AT> ? 13+100011001(2);
294
AT> ? 0x412+152;
0x4aa
AT> ? 412(16)+152(10);
0x4aa
```

As shown, the hexadecimal numbers have the special syntax (0x...) which is standard in the literature. As a general rule, the results of the operations are promoted to the higher radix. Use the integer() function to convert any number (of any type) to integer, and to force a radix conversion (defaults to 10):

[4] Internally implemented by Java BigInteger.

```
AT> a=0x12a + 1000;
AT> ? a;
0x512
AT> ? integer(a);
1298
AT> ? integer(a,11);
a80(11)
```

2.2 Rationals

These corresponds to the fractions[5].

```
AT> a=1/7;
AT> b=3/997;
AT> ? a+b;
1018/6979
AT> ? typeinfo(-3/4);
rational
```

Rationals numbers are automatically converted to their irreducible form.

2.3 Reals

Floating point numbers[6]. When combined with rationals, the result gets promoted to the real type.

```
AT> ? 1.3+4/5;
2.10000
AT> // show type promotion
AT> ? typeinfo(1.3+4/5);
real
AT> ? 1 + 2e-5;
1.00002
```

Note the insertion of comments on lines beginning with '//'. These are specially useful in external files with AT commands.

2.4 RatRoot

This is an attempt to work with "exact" numbers for problems where rationals are not enough. This type represents numbers with the form:

$$\sum_{i=1}^{n} a_i \sqrt[r_i]{R_i}$$

[5] Internally implemented as a pair of Java BigIntegers.

[6] Internally stored with Java "double"

where a_i is a non-zero rational, r_i is a natural (from 1), R_i is a natural (from 1) or a negative integer if $r_i = 2$ (in such case it is a complex number.) Note that zero is an special case where the sum does not contain any term. Also, the case $r_i = 1$ corresponds to a Rational.

The square roots are expressed in AT with the syntax "`RT(rad)`"; and n-th order radicals use the "`RT([n]rad)`" notation. For example, to divide the following two RatRoots (effectively rationalizing the expression):

$$\frac{\sqrt[3]{5}+1}{\sqrt{3}+\sqrt{7}-2}$$

we proceed as follows:

```
AT> ? (RT([3]5)+1)/(RT(3)+RT(7)-2);
```

Which results in:

$$-\frac{1}{4}+\frac{\sqrt{3}}{6}+\frac{\sqrt{21}}{12}-\frac{\sqrt[3]{5}}{4}+\frac{\sqrt[6]{675}}{6}+\frac{\sqrt[6]{231525}}{12}$$

and can be verified by reversing procedure; i.e. multiplying the quotient by the divider:

```
AT> ? (-1/4+1/6*RT(3)+1/12*RT(21)-1/4*RT([3]5)+1/6*RT([6]675)+
--> 1/12*RT([6]231525))*(RT(3)+RT(7)-2);
```

Exact complex numbers can be represented by RatRoots; for example:

```
AT> ? (3+8i)*(1/3*RT(2)-RT(7)i);
```

which returns:

$$\frac{8\sqrt{2}}{3}i-3\sqrt{7}i+\sqrt{2}+8\sqrt{7}$$

2.5 Complex

Represents a pair of floating point reals, subject to the complex field rules. For example:

```
AT> ? (3.0+8i)*(1/3*RT(2)-RT(7)i);
22.5802-4.16602i
```

Compare with the last RatRoot example: the "3.0" forces the floating point representation, which in turn promotes the RatRoots to Real or Complex as needed.

2.6 Basic Operations

The usual operations (addition, subtraction, product, division) are provided by the standard operators +, -, *, and /. For integers, the “modulo” operation is provided by the mod() function:

```
AT> ? mod(36,5);
1
AT> ? mod(-12,5);
3
```

2.6.1 Power

The usual “^” operator is provided for standard power:

```
AT> ? 4^0.5;
2.00000
AT> ? 2^0.5;
1.41421
AT> ?(3+4i)^(3+4i);
-2.99799+0.623785i
```

2.7 Type Conversions

The numbers can be converted between numeric types; this often works well when going to “supertypes”; for example:

```
AT> ? real(1/3);
0.333333
```

In reverse we could be lucky when going to rationals from reals:

```
AT> a = sqrt(5)/7.0;
AT> ? a;
0.319438
AT> b = a / sqrt(5);
AT> ? b;
0.142857
AT> ? rational(b);
1/7
```

but not always:

```
AT> a=(3.0+8i)*(1/3*RT(2)-RT(7)i);
AT> ? a;
22.5802-4.16602i
AT> ? ratroot(a);
-8324688183610/1998236361309i+1104734509655545/48924869264617
```

Type information can be extracted with typeinfo():

```
AT> ? typeinfo(0.142857);
real
AT> ? typeinfo(31/34);
rational
AT> ? typeinfo(4+RT([3]4));
ratroot
AT> ? typeinfo(4.0-9i);
complex
```

2.8 Output Formatting

The `sprintf()` function allows the formatting of the numbers. For example:

```
AT> ? sprintf("K = %.10g [+/- %e]", real(1/7), real(5/306));
K = 0.1428571429 [+/- 1.633987e-02]
```

Note that integers and rationals (and the coefficients of RatRoots) are backed by Java BigIntegers, so their allowed format conversions are limited to %d, %o, %x, %X. The reals are backed by doubles (and the complex by two doubles), so their allowed format conversions are %e, %E, %g, %G, %f, %a, %A.

A format conversion applied on a matrix is in turn applied to all the matrix elements: make sure they are of a type compatible with the conversion format, or force the type of the elements of the matrix. For example, the following fails:

```
AT> ? sprintf("%5d", [1.0 , 44 , 21/4]);
ERROR: Invalid format conversion for matrix element
```

Forcing to integer elements, the format is valid now:

```
AT> ? sprintf("%5d", integer([1.0 , 44 , 21/4]));
    1      44       5
```

Please check `https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html#syntax` for more information.

2.9 Strings

The string type is used to represent plain texts. They can be created by text surrounded by double quotes, or from the `string()` function:

```
AT> k="this is a string";
AT> ? "=>" + substr(k,5,9) + "<=";
=>is a<=
AT> ? strlen(string(99^99));
198
```

2.10 Logical Values

Logical/boolean elements represent the "`true`" and "`false`" values.

```
AT> a = logical(true);
AT> b = logical(false);
AT> c = a && b;
AT> ? typeinfo(c);
logical
AT> d = a || b;
AT> ? typeinfo(d);
logical
AT> ? c;
false
AT> ? d;
true
AT> ? !d;
false
```

The standard comparison operators produce logical values:

```
AT> ? 34-4 == 30;
true
AT> ? 34-4 == 31;
false
AT> ? 34-4 >= 100;
false
AT> ? 34-4 <= 100;
true
AT> ? 10 > 100;
false
AT> ? 100 > 10;
true
AT> ? 34+4 > 35;
true
AT> ? 34-4 > 35;
false
```

The `true/false` values are translated as `1/0` integers on conversions:

```
AT> ? 5 + (34-4 <= 100);
6
AT> ? 5 + (34-4 >= 100);
5
```

2.11 Debugging

In order to get more information about the calculation process, the "debugging" flag may be enabled:

```
AT> ? debug();
false
AT> ? expand(@(x){(x+1)^3});
@(x){ 1+3*x+3*x^2+x^3; }
AT> debug(true);
AT> ? debug();
true
AT> ? expand(@(x){(x+1)^3});
DEBUG: Called expand with measure-level=12
DEBUG: Expansion iteration 1
DEBUG: Expansion iteration 2
DEBUG: Expansion iteration 3
DEBUG: Expansion iteration 4
DEBUG: Expansion interrupted since no change on step 4
@(x){ 1+3*x+3*x^2+x^3; }
AT> debug(false);
AT> ? expand(@(x){(x+1)^3});
@(x){ 1+3*x+3*x^2+x^3; }
```

Note that the debugging flag is a new addition which currently is not useful for most functions; future versions will improve this situation.

3 Linear Algebra

3.1 Dealing with Matrices

A matrix may be created using a syntax which follows Octave. For example:

```
AT> x=[2,3,1/3;3,1,-1;-3,1/10,RT(2)];
```

defines the matrix:

$$x = \begin{bmatrix} 2 & 3 & \frac{1}{3} \\ 3 & 1 & -1 \\ -3 & \frac{1}{10} & \sqrt{2} \end{bmatrix}$$

The usual operations are available using the standard operators: addition, product by a scalar, matrix product, etc; other ones are provided via additional functions.

Unlike the rest of the AT elements, the matrix is a mutable object: we can reset its elements to any value at any time. Its dimensions are fixed, though.

An element can be extracted or set with the syntax “`name(row,column)`” where `row` and `column` are indexed from 1. A colon (wildcard) symbol means a full row or a full column. Also, with the syntax “`name(row,column,num-rows,num-cols)`” a submatrix can be extracted, with the wildcard meaning “up to the end” for the two last arguments. For example, with the previous matrix:

```
AT> ? x(3,2);
1/10
AT> ? x(1,:);
2       3       1/3
AT> ? x(:,1);
2
3
-3
AT> ? x(2,1,2,2);
3    1
-3   1/10
AT> x(1,1)=100;
```

after the last command, the matrix is now:

$$x = \begin{bmatrix} 100 & 3 & \frac{1}{3} \\ 3 & 1 & -1 \\ -3 & \frac{1}{10} & \sqrt{2} \end{bmatrix}$$

The syntax `name(idx)` works in the same way for row and column matrices. For non row/column matrices, the `idx` index runs over all the matrix elements starting with all the elements in the first row, following with all the second row elements, and so on:

```
AT> a=magic(3);
AT> ? a;
8       1       6
3       5       7
4       9       2
AT> ? a(2);
1
AT> ? a(6);
7
AT> a(6)=100;
AT> ? a;
8       1       6
3       5       100
4       9       2
```

In order to join two matrices, use the syntax `m1 || m2` and `m1 && m2` for horizontal and vertical concatenation, respectively (the `hjoin()` and `vjoin()` functions do the same.)

Following Octave, there exist the ’`.*`’ and ’`.^`’ operators. The first one allows the by-element product of two matrices (which must have the same dimensions.) On the other side, the power-by-element is applied to a matrix and a -typically real- number (exponent), and produces a new matrix obtained by powering each element of the first one with the exponent.

```
AT> ? [2,3;4,5;4,-1].*[0,0;1,-1;100,0];
0    0
```

```
4     -5
400   0

AT> ? [2,3;4,5;4,-1].^2;
4     9
16    25
16    1
```

3.2 Solving Linear Equations

The `solve()` function finds the solution of system of linear equations. For example, the following system (with a unique solution) is solved below:

$$\begin{cases} x + y = 10 \\ x - y = 2 \end{cases}$$

```
AT> ? solve([1,1,10; 1,-1,2]);
6
4
```

Note that `solve()` is able to use the extended matrix:

$$A = \begin{bmatrix} 1 & 1 & 10 \\ 1 & -1 & 2 \end{bmatrix}$$

But also as the pair:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \; ; \; B = \begin{bmatrix} 10 \\ 2 \end{bmatrix}$$

```
AT> ? solve([1,1; 1,-1],[10; 2]);
6
4
```

3.2.1 Indeterminate Solutions

For systems with indeterminate solutions the “indet” option allows their extraction. For example, the following the system can be solved as shown below:

$$\begin{cases} x + y + z = 10 \\ x - y + 5z = 20 \end{cases}$$

```
AT> ? solve([1,1,1,10; 1,-1,5,20]);
ERROR: System does not have unique solution - try 'indet' option
AT> ? solve([1,1,1,10; 1,-1,5,20],"indet");
0
5
5
```

If more solutions are needed, just pass a natural number to the "indet" option:

```
AT> ? solve([1,1,1,10; 1,-1,5,20],"indet=1");
1
13/3
14/3
AT> ? solve([1,1,1,10; 1,-1,5,20],"indet=2");
2
11/3
13/3
```

3.3 Eigenvalues and Eigenvectors

Use the eig() function in order to get them in a column matrix:

```
AT> ? eig([1,2;-2,5]);
3
3
```

Here we have the eigenvector “3” with algebraic multiplicity two (hence two rows.) Providing the matrix and one of the eigenvalues, we get the corresponding eigenvectors as columns in a matrix:

```
AT> ? eig([1,2;-2,5], 3);
1
1
```

3.4 Mapping Elements

The map() function applies a function to each of the matrix elements:

```
AT> A=magic(3);
AT> ? A;
8       1       6
3       5       7
4       9       2
AT> ? map(A, @(x){2*x});
16      2       12
6       10      14
8       18      4
```

3.4.1 Translating matrix elements

The map() function may be used to get a new matrix with the elements translated; just apply the translation functions:

```
AT> a=rand(4,2)*100;
AT> ? a;
61.7437 11.2668
75.6517 31.0725
90.5139 96.4488
36.0076 51.9893
AT> ? map(a,integer);
61      11
75      31
90      96
36      51
AT> ? map(magic(3),real);
8.00000 1.00000 6.00000
3.00000 5.00000 7.00000
4.00000 9.00000 2.00000
```

3.5 Matrices as Elements

A matrix is considered as another special "numeric" type, so it can be inserted in another matrix:

```
AT> A=[[2,3],[5,12,6]];
AT> ? A;
Element (1,1):
2       3
Element (1,2):
5       12      6
```

3.6 Type Conversions

The type conversion functions are available for matrices:

```
AT> ki=magic(4);
AT> ? ki;
16   2    3     13
5    11   10    8
9    7    6     12
4    14   15    1
AT> kr=real(ki);
AT> ? kr;
16.0000 2.00000 3.00000 13.0000
5.00000 11.0000 10.0000 8.00000
9.00000 7.00000 6.00000 12.0000
4.00000 14.0000 15.0000 1.00000
```

The conversion to the “real” type could be necessary for some functions which make a big number of operations with the matrix elements (like the statistical ones.) In some cases, the conversion is implicit (for example, for the approximation the solutions of differential equations.)

There are many more linear algebra functions, which can be queried with ‘‘help la”.

4 Working with Functions

The syntax:

$$@(arg1[,arg2...])\{body\}$$

allows the definition of a function. For example:

```
AT> f=@(t){3*exp(-t)};
```

defines the function $f(t) = 3e^{-t}$. This can be evaluated using the standard syntax:

```
AT> ? f(1);
1.10364
```

Functions of multiple variables can be defined simply by extending the argument list. For example:

```
AT> f=@(x,y){x^2+y^2};
```

An important concept is that functions are considered an special numeric elements, so they can be combined using the standard operations. For example:

```
AT> f1=@(x,y){x^2+y^2};
AT> f2=@(x,y){x-y};
AT> f3=f1*f2+f1;
AT> ? f3;
@(x,y){ x^2+y^2+(x-y)*(x^2+y^2); }
```

4.1 Expansion

Use the `expand()` function to “expand” and hopefully simplify the algebraic expressions of a function.

```
AT> ? expand(@(x){sin(2*x)*cos(3*x)+sin(2*x)*cos(x)});
@(x){ 1/2*sin(5*x)+1/2*sin(3*x); }
AT> ? expand(@(x){((x+1)^7-(x-1)^7-2)/x^2});
@(x){ 14*x^4+70*x^2+42; }
```

5 Calculus Tools

We support the usual functions: `sin()`, `cos()`, `tan()`, `atan()`, `sinh()`, `cosh()`, `tanh()`, `atanh()`, `log()`, `exp()`, `sqrt()`, `erf()`. Other derived functions (like cotangent, secant, etc.) are not included.

5.1 Derivatives

5.1.1 Symbolic Derivation

Use the `symder()` for symbolic derivation. A second argument signals the derivation variable; by default it is the first in the argument list.

```
AT> f=@(x){x^2-cos(3*x)}; ? symder(f);
@(x){ 2*x+3*sin(3*x); }
AT> fmul=@(x,y){x^4+y^5-2};
AT> ? symder(fmul);
@(x,y){ 4*x^3; }
AT> ? symder(fmul,"y");
@(x,y){ 5*y^4; }
```

Look up the `jacobian()` function to calculate such important matrix in a symbolic way.

If the function references another function, this will be automatically included in the derivation:

```
AT> g=@(x){7*x^7-cos(5*x);};
AT> f=@(x){x^2-cos(3*x)+2*g};
AT> ? symder(f);
@(x){ 2*x+3*sin(3*x)+98*x^6+10*sin(5*x); }
```

5.1.2 Numerical Derivation

The `diff5p()` function uses the "5-point" approximation for the derivative. Provide a function, a point for calculation, and optionally the approximation interval (defaults to $h = 10^{-6}$.)

```
AT> f=@(x){x*exp(x)};
AT> ? diff5p(f,2.0,0.1);
22.1670
AT> ? diff5p(f,2.0);
22.1672
```

5.2 Integration

5.2.1 Symbolic Integration

The `symint()` provides symbolic integration. Note that currently it is very basic, but solves some common (and tedious) "integration by parts" expressions.

For example, to integrate:

$$\int cos^4(2x) + x^3e^{-x}dx$$

```
AT> ? symint(@(x){cos(2*x)^4+x^3*exp(-x)});
@(x){ 1/64*sin(8*x)+3/8*x+1/8*sin(4*x)-1*exp(-1*x)*x^3-
        3*exp(-1*x)*x^2-6*exp(-1*x)*x-6*exp(-1*x); }
```

5.2.2 Numerical Integration

Here we provide several methods. See the help of the functions `trapz()`, `simpson()`, `romberg()`, `quad()`, `quadv()` for the supported algorithms. For example, to integrate with 10 steps via Simpson, followed by 4th level Romberg the expression:

$$\int_0^2 \sqrt{1+x^2}dx$$

```
AT> f = @(x){sqrt(1.0+x*x)};
AT> ? simpson(f,0.0,2.0,10);
2.95788
AT> ? romberg(f,0.0,2.0,4);
2.95788
```

Multiple integrals are provided via `dblquad()` and `triplequad()`. For example, to integrate with a net of 3x3x5 steps:

$$\int_{0.1}^{0.5} \int_{x^3}^{x^2} e^{\frac{y}{x}} dy\ dx$$

```
AT> f = @(x,y){exp(y/x)};
AT> y1 = @(x){x*x*x};
AT> y2 = @(x){x*x};
AT> ? dblquad(f, 0.1, 0.5, y1, y2, 3, 3, 5);
0.0333056
```

5.3 Ordinary Differential Equations

5.3.1 Symbolic ODEs

Single linear ODE

To solve:

$$a_0(t)y + a_1(t)y' + ... + a_n(t)y^{(n)} = b(t)$$

subject to:

$$y(t_0) = y_0,\ y'(t_0) = y'_0,\ ...\ y^{(n-1)}(t_0) = y_0^{(n-1)}$$

Use the `symodel()` function, passing the $a_i(t)$ functions (coefficients) in a column matrix, and the initial conditions in another one. For example, to solve:

$$y'' - 4y' - 12y = 3e^{5t}$$

$$y(0) = \frac{18}{7}\ ;\ y'(0) = -\frac{1}{7}$$

```
? symodel(@(t){3*exp(5*t)}, [-12; -4; 1], [18/7; -1/7]);
@(t){ -3/7*exp(5*t)+exp(6*t)+2*exp(-2*t); }
```

Linear Constant Coefficients ODE System Here we solve the system:

$$X'(t) = AX(t) + b(t)$$

Where $X(t)$ is a column vector of functions; subject to $X(t_0) = X_0$.
Use the `symodesys()` function. For example to solve:

$$\begin{array}{rcl} x' = & 4x - y + z & -(t+1)^2 \\ y' = & -x + 3y - 2z & +2t^2 + t + 15 \\ z' = & x - 2y + 3z & -3t^2 + t - 10 \end{array}$$

Subject to $x(0) = 3;\ y(0) = -7;\ z(0) = -2$, we proceed as follows:

```
AT> A=[4,-1,1;-1,3,-2;1,-2,3];
AT> b1 = @(t){-(t+1)^2};
AT> b2 = @(t){2*t^2+t+15};
AT> b3 = @(t){-3*t^2+t-10};
AT> b=[b1;b2;b3];
AT> x0=[3;-7;-2];
AT> ? symodesys(A,b,0,x0);
@(t){ 32/27*exp(6*t)+76/27*exp(3*t)+4/9*t+-1; }
@(t){ -32/27*exp(6*t)+38/27*exp(3*t)+-2*exp(t)+-1/9*t+-47/9; }
@(t){ 32/27*exp(6*t)+-38/27*exp(3*t)+-2*exp(t)+t^2+1/9*t+2/9; }
```

So we got the solutions x(t), y(t) and z(t) in a column matrix.

5.3.2 Numerical ODEs

Single ODE The traditional Runge-Kutta method is provided with the `oderk()` function. For example, to solve the ODE in the interval $[0, 2]$:

$$y' = y - t^2 + 1$$

$$y(0) = 0.5$$

```
AT> f = @(t,y){y-t*t+1};
AT> ? oderk(f, 0, 2, 0.5, 10);
0.00000 0.500000
0.20000 0.829293
0.40000 1.21408
0.60000 1.64892
0.80000 2.12720
1.00000 2.64082
1.20000 3.17989
1.40000 3.73234
1.60000 4.28341
1.80000 4.81509
2.00000 5.30536
```

Please check the `oderkf()`, `oderk2l()`, `oderk2n()` functions for more variants. The basic Euler method is provided by `odeeuler()`, and the Adams-Moulton multistep algorithm is provided by `odeam2()`, `odeam3()` and `odeam4()`.

ODE System Here we solve the general system:

$$\begin{gathered} x_1' = f_1(t, x_1, ..., x_n) \\ ... \\ x_n' = f_n(t, x_1, ..., x_n) \end{gathered}$$

Given $x_1(t_0)$, ... $x_n(t_0)$.

AT provides the oderksys(functions,t-start,t-end,initial-conditions,steps) function. For example, to solve the system for $t \in [0, 0.5]$ with 5 steps:

$$y' = -4y - 3z.sin(t) + 6$$

$$z' = -2.4y + 1.6z + 3.6$$

$$y(0) = 0 \ ; \ z(0) = 0$$

```
AT> yp = @(t,y,z){-4*y+3*z*sin(t)+6.0;};
AT> zp = @(t,y,z){-2.4*y+1.6*z+3.6;};
AT> ? oderksys([yp;zp], 0.0, 0.5, [0.0;0.0], 5);
0.00000     0.00000     0.00000
0.100000    0.497423    0.323238
0.200000    0.846215    0.593585
0.300000    1.10698     0.832265
0.400000    1.31824     1.05143
0.500000    1.50433     1.25724
```

Note that the answer provides a matrix with values for t, y(t) and z(t).

Boundary Value 2nd order ODEs Check the `bvprk2l()` and `bvprk2n()` functions.

5.3.3 Phase Diagrams

For two-variable autonomous ODE systems, a phase diagram may be plotted. For the system:

$$x' = ax + by$$

$$y' = cx + dy$$

The matrix:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

allows the ODE to be written as $X' = AX$; in that case the syntax `phase2d(A)` generates a phase diagram using the region $X * Y = [-1, 1] * [-1, 1]$.

Other regions should be selected using `range()`:

```
A=[2.0, 1.0; 1.0, -1.0];
x=range(-5,5,0.5);
y=range(-5,5,0.5);
phase2d(A, x, y);
```

Note that the number of elements in the range do define the number of field arrows in the graph.

For non linear systems with the form:

$$x' = u(x, y)$$

$$y' = v(x, y)$$

The matrix:

$$A = \left[\begin{array}{c} u(x,y) \\ v(x,y) \end{array} \right]$$

will be provided to `phase2d()` with the same syntax. For example, to draw the phase diagram for the system:

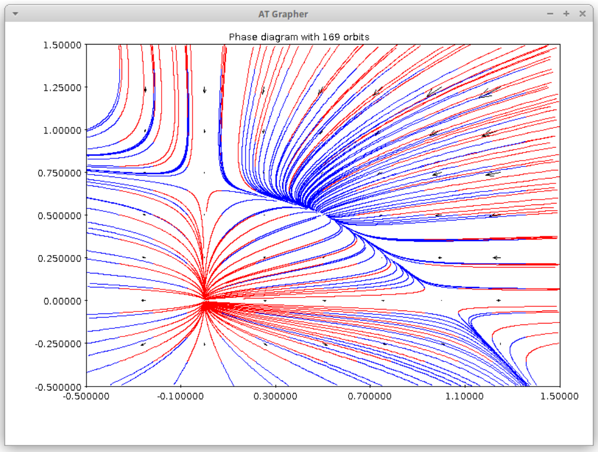
$$x' = (1 - x - y)x$$

$$y' = (\tfrac{1}{2} - \tfrac{1}{4}x - \tfrac{3}{4}y)y$$

and with a hint of 200 orbits (a reduced number will be actually produced):

```
fx=@(x,y){(1.0-x-y)*x};
fy=@(x,y){(0.5-0.25*x-0.75*y)*y};
A=[fx;fy];
x=range(-0.5, 1.5, 0.25); y=range(-0.5, 1.5, 0.25);
phase2d(A, x, y, 200);
```

The diagram shows the singularities:



6 Partial Derivatives

6.1 Symbolic Derivation

Let's suppose we have the function:

$$F(x,y,U,V,W) = sin(y - 5ax) + 4U + 5\frac{\partial V}{\partial x} - 4\frac{\partial W}{\partial y}$$

Let's find its partial derivatives:

$$\frac{\partial F}{\partial x} = -5a.cos(y - 5ax) + 4\frac{\partial U}{\partial x} + 5\frac{\partial^2 V}{\partial x^2} - 4\frac{\partial^2 W}{\partial y \partial x}$$

$$\frac{\partial F}{\partial y} = cos(y - 5ax) + 4\frac{\partial U}{\partial y} + 5\frac{\partial^2 V}{\partial x \partial y} - 4\frac{\partial^2 W}{\partial y^2}$$

We must notify that U, V and W are some functions (and not constants); in order to specify $\frac{\partial V}{\partial x}$ we use the convention:

```
V__x
```

i.e. we add two underscores before the derivation variable:

```
AT> F=@(x,y,U,V__x,W__y){sin(y-5*a*x)+4*U+5*V__x-4*W__y};
AT> ? symder(F, "x", ["U","V","W"]);
@(x,y,U,V__x,W__y){ -5*a*cos(y+-5*a*x)+4*U__x+5*V__x__x+-4*W__x__y; }
AT> ? symder(F, "y", ["U","V","W"]);
@(x,y,U,V__x,W__y){ cos(y+-5*a*x)+4*U__y+5*V__x__y+-4*W__y__y; }
```

Note that higher order partial derivatives are specified by repetition of this convention.

6.2 Symbolic Substitution

Any function variable can be replaced by another expression with the `subs()` function; for example:

```
AT> G=@(x,y){sin(x)+3*x+4*y};
AT> ? subs(G, "x", @(t){t+1});
@(x,y){ sin(t+1)+3*(t+1)+4*y; }
```

6.3 Transformation of Coordinates

Let's suppose we need a transformation of the (x,y) coordinates to an (m,n) system, for the expression:

$$A(x, y) = sin(x + y) + 3x^2 + 4y$$

The new and old coordinates are related by:

$$x = 19m - tan(n)$$

$$y = mn$$

For AT we will define the "transformation" matrix:

$$T = \left[\begin{array}{cc} ”x” & 19m - tan(n) \\ ”y” & mn \end{array} \right]$$

And do the substitution:

```
AT> A=@(x,y){sin(x+y)+3*x^2+4*y};
AT> tx=@(m,n){19*m-tan(n)};
AT> ty=@(m,n){m*n};
AT> T=["x", tx; "y", ty];
AT> ? subs(A, T);
@(m,n){ sin(19*m+-1*tan(n))+57*m+-3*tan(n)+4*m*n; }
```

Now let's suppose we have a function $U(x, y)$ which will be represented as $V(m, n)$ in the new system (i.e. $U = V$.) If the expression is now:

$$A(x, y, U) = sin(x + y) + 3x^2 + U^2 + 4y$$

Then we get $A(m, n)$ by providing a new matrix containing the function equivalences: each equivalence in a row; every row containing the old function name and the new one:

```
AT> A=@(x,y){sin(x+y)+3*x^2+U^2+4*y};
AT> tx=@(m,n){19*m-tan(n)};
AT> ty=@(m,n){m*n};
AT> T=["x", tx; "y", ty];
AT> ufs=["U","V"];
AT> ? subs(A, T, ufs);
@(m,n){ sin(19*m+-1*tan(n)+m*n)+1083*m^2+-114*m*tan(n)+3*tan(n)^2+V^2+4*m*n; }
```

6.4 Transformation of Coordinates with Partial Derivatives

Let's suppose that we need to transform the expression:

$$A(x, y, U) = 3U_{xx} + 10U_{xy} + 3U_{yy}$$

To the coordinates (m,n) related by the equation system:

$$x = m + n$$

$$y = m - n$$

or in AT style:

$$T = \left[\begin{array}{cc} ”x” & m + n \\ ”y” & m - n \end{array} \right]$$

But in this case we *must* provide both the direct and the inverse transformation relations (i.e. for “m” and “n” in terms of “x” and “y”.) To see why, let’s suppose we need to transform the expression $\frac{\partial U}{\partial x}$ to the system system (m,n); this entails the following expansion:

$$\frac{\partial U}{\partial x} = \frac{\partial V}{\partial m}\frac{\partial m}{\partial x} + \frac{\partial V}{\partial n}\frac{\partial n}{\partial x}$$

So AT does need some way to compute expressions -in the (m,n) coordinates- for $\frac{\partial m}{\partial x}$ and $\frac{\partial n}{\partial x}$; ergo, we must provide $m = m(x, y)$ and $n = n(x, y)$ in a matrix for the new coordinates (the user must solve “m” and “n” in terms of “x” and “y” from the previous equation system):

$$TR = \begin{bmatrix} ”m” & \frac{1}{2}(x+y) \\ ”n” & \frac{1}{2}(x-y) \end{bmatrix}$$

We are ready to execute `subs()`:

```
AT> A=@(x,y){3*U__x__x + 10*U__x__y + 3*U__y__y};
AT> tx=@(m,n){m+n};
AT> ty=@(m,n){m-n};
AT> T=["x", tx; "y", ty];
AT> ufs=["U","V"];
AT> tm=@(x,y){(x+y)/2};
AT> tn=@(x,y){(x-y)/2};
AT> TR=["m", tm; "n", tn];
AT> ? subs(A, T, ufs, TR);
@(m,n){ 4*V__m__m+-1*V__n__n; }
```

The result was:

$$A(x, y, U) = 4V_{mm} - V_{nn}$$

6.4.1 Canonical Form

As an application of this section, consider the transformation of

$$A(x, y, U) = 3U_{xx} + 10U_{xy} + 3U_{yy}$$

to its canonical form. Since the coefficients are constants, we may get the roots of the polynomial $t^2 + 10t + 3 = 0$ (see below for more information about root extraction):

```
AT> ? roots(poly(3,10,3));
-1/3
-3
```

And solve the ODEs:

$$\frac{dy}{dx} + (-\frac{1}{3}) = 0$$

$$\frac{dy}{dx} + (-3) = 0$$

which amounts to:

```
AT> ? symodel(1/3, [0; 1], [0]);
@(t){ 1/3*t; }
AT> ? symodel(3, [0; 1], [0]);
@(t){ 3*t; }
```

So the characteristic equations are $y = \frac{x}{3} + c_1$ and $y = 3x + c_2$; now we do the transformation:

$$m = y - \frac{x}{3};\ n = y - 3x$$

Which implies:

$$x = \frac{m-n}{8};\ y = \frac{9m-n}{8}$$

```
AT> T=["x",@(m,n){(m-n)/8};"y",@(m,n){(9*m-n)/8}];
AT> TR=["m",@(x,y){y-x/3};"n",@(x,y){y-3*x}];
AT> ? subs(A, T, ["U","V"], TR);
@(m,n){ -64/3*V__m__n; }
```

The last result corresponds to the canonical expression:

$$A(m, n, V) = -\frac{64}{3}V_{mn}$$

7 Polynomials

This is a data structure allowing to deal with polynomials and related operations. A polynomial can be created with the `poly()` function[7]:

```
AT> ? poly(6,0,0,1,-1);
+6x^4 +0x^3 +0x^2 +1x -1
```

Polynomials are considered as numeric elements, so they support most of their operations:

[7] Currently, all polynomials are automatically assumed on "x" variable. Future releases may allow arbitrary variable names.

```
AT> p1 = poly(1,-2,1);
AT> p2 = poly(1, 2,1);
AT> ? p1*p2;
+1x^4 +0x^3 -2x^2 +0x +1
```

7.1 Evaluation

Use `polyval()` to calculate P(a) for any polynomial "P" and any argument "a".

```
AT> ? polyval(poly(1,-5,6),2);
0
AT> ? polyval(poly(1,-5,6),123);
14520
```

7.2 Polynomial Interpolation

Using Lagrange interpolation, for two column vectors with the same size “n”, or a n*2 matrix containing the ’x’ and ’y’ columns, a polynomial of degree (n-1) can be interpolated. For example:

```
AT> ? polyfit([1,1 ; 2,4 ; 3,9]);
+1x^2 +0x +0
AT> ? polyfit([1,1 ; 2,8 ; 3,27]);
+6x^2 -11x +6
```

8 Finding Roots

8.1 Roots of Polynomials

For polynomials, the `roots()` function extracts all the roots (possibly complex numbers.) If the polynomial’s coefficients are exact numbers, AT tries to get exact roots (in a column vector):

```
AT> ? roots(p1*p2);
1
1
-1
-1
AT> ? p1 + p2;
+2 +0x +2x^2
AT> ? roots(p1 + p2);
1i
-1i
```

8.2 Roots of Arbitrary Functions

These functions allow the extraction of roots for arbitrary non linear single-variable functions. Some classic algorithms are provided; for example, the Newton-Raphson method is implemented in the `newtonraphson()` function. As is well known, this method does require the evaluation of the function derivative, which should be provided as second argument:

```
AT> f=@(x){x^3+4*x^2-10.0};
AT> ? newtonraphson(f, symder(f), 1, :, 5);
1.36523
```

If it is not possible to provide the function derivative, a wildcard instructs `newtonraphson()` to calculate an approximation for it (which is slower and in general less precise.)

```
AT> f=@(x){x^3+4*x^2-10.0};
AT> ? newtonraphson(f, :, 1, :, 5);
1.36523
```

The last arguments of this function are the required error tolerance and the maximum number of iteration steps: at least one of these arguments must be provided.

Note that `newtonraphson()` and similar functions like `secant()` and `bisection()` may also accept a polynomial as target function, but this has several disadvantages when compared to the previous `roots()` function:

1. Exaction of only a single root at a time, upon “guessed” starting points
2. No exact (rational/ratroot) roots
3. Only real roots (no complex roots)

9 Natural Numbers

AT does provide some basic functions related to natural numbers; it is up to the user to ensure that positive integers are provided to functions expecting natural numbers.

9.1 Prime Factors

For example, to compute the GCD (greatest common divisor) of a pair of numbers:

```
AT> ? gcd([2400; 840]);
120
```

The following example shows how to decompose a natural number in prime factors (note that $2400 = 2^5.3.5^2$)

```
AT> ? factor2(2400);
2       5
3       1
5       2
```

A column matrix with the first "n" primes can be generated with the `list_primes()` function (see also the `primes()` function):

```
AT> ? list_primes(10);
2
3
5
7
11
13
17
19
23
29
```

The Euler phi "totient" function $\phi(n)$ is implemented with the `totient()` function:

```
AT> ? totient(36);
12
```

9.2 Combinatorials

The `factorial()` function allows the calculation of (very big) factorials. There is also the `logfactorial()` function in order to get the logarithm of the factorial, which is faster but not exact (it uses Stirling approximation for big values.)

```
AT> ? factorial(20);
2432902008176640000
AT> ? logfactorial(20);
42.3356
AT> ? exp(logfactorial(20));
2.43290e+18
```

The combinations of "n" objects taken by "k" at a time can be found with the `comb(n,k)` function:

```
AT> ? comb(50,10);
10272278170
```

See also the `logcomb(n,k)`.

10 Finite Groups

Finite groups[8] are implemented with special matrices containing their elements and operations; the elements are any numeric element supported by AT. For example, the symmetric S_n groups can be created with the `grpsym()` function; its elements may be obtained with the `grpelm()` function:

```
AT> g=grpsym(3);
AT> ? grpelm(g);
1
2
3
4
5
6
```

Since the S_n group can be viewed as a set of permutations, these may be extracted from the elements using `grpsym()` again:

```
AT> for(i = 1; i <= grpord(g); i = i + 1) { ? sprintf("%d->
%s", i, grpsym(g,i)) };
1->    1       2       3
2->    1       3       2
3->    2       1       3
4->    2       3       1
5->    3       1       2
6->    3       2       1
```

Note that the groups are stored in a special format matrix which can be displayed (as any matrix); but, its contents and layout are not guaranteed for future releases, so the user should not rely on its contents and use the accessor functions we are describing.

In order to get all the available information in a group object, use the `grpshow()` function:

```
AT> g=grpsym(3);
AT> ? grpshow(g);
identity: 1

group elements:
   1: 1
   2: 2
   3: 3
   4: 4
   5: 5
   6: 6
```

[8] The implemented algorithms were taken from some ideas and exercises in Butler, G. "Fundamental Algorithms for permutation groups". Springer. 1991. We didn't use the most optimized versions but favored the clearest not-trivial ones.

```
containing group elements:
   1: 1
   2: 2
   3: 3
   4: 4
   5: 5
   6: 6

operation table matrix:
1       2       3       4       5       6
2       1       4       3       6       5
3       5       1       6       2       4
4       6       2       5       1       3
5       3       6       1       4       2
6       4       5       2       3       1

generators:
   1: 4
   2: 2
```

For the creation of arbitrary groups, an operation table matrix must be provided; see the `grpnew()` function for more information.

10.1 Operation Table

The `grptab()` function extracts a matrix with the full (containing) group. Note that the actual group may be a subset (subgroup) of the operation matrix (this allows the building of cosets, for example.)

```
AT> ? grptab(g);
1       2       3       4       5       6
2       1       4       3       6       5
3       5       1       6       2       4
4       6       2       5       1       3
5       3       6       1       4       2
6       4       5       2       3       1
```

10.2 Generators

Given a group and some elements (generators), the `grpgen()` allows the generation of a group/subgroup. For example, for the group of the symmetries of a square, the $< 2 >$ generator corresponds to the rotational symmetries subgroup:

```
AT> sq=grpsqsym();
AT> sub=grpgen(sq,2);
AT> ? grpelm(sub);
1
2
```

```
3
4
```

Besides specific elements, a matrix with the generators may be provided to grpgen().

On the other hand, from a group it is possible to get a set of generators (in a column matrix) with grpgetgen():

```
AT> ? grpgetgen(grpsqsym());
2
5
```

So $< 2, 5 >$ is a generator for the "symmetries of the square" group.

10.3 Subgroups

The grpsub() extracts all the subgroups of a provided group. They are returned in a column matrix. In the following example, sg contains the eight proper subgroups of the symmetries of the square; one of these subgroups is assigned to "sg4" and its elements are shown. Finally, sg contains all the subgroups (including the total group.)

```
AT> sg=grpsub(grpsqsym());
AT> ? size(sg);
8       1
AT> sg4=sg(4);
AT>  ? grpelm(sg4);
1
3
5
7
AT> sg=grpsub(grpsqsym(), true);
AT> ? size(sg);
9       1
```

Note that the answer from grpsub() is not useful for calling the group functions: its elements (which are the subgroups) must be extracted beforehand.

11 Bits and Bytes

11.1 Byte Matrices

The bit manipulation functions work on integers and "byte matrices". Byte matrices are column matrices containing unsigned integers which represent bytes (their range should be $[0, 255]$ and radix 16, though not required); such matrices represent byte arrays, which are useful for bit manipulations and cryptography.

Byte matrices may be created directly with the usual notation:

```
AT> x=[0x5f; 0x3; 0x7a];
```

11.1.1 Hexadecimal Sequences

Often it is more straightforward to start from a text string containing hexadecimal two-digit numbers representing individual bytes; the `hex2bm()` function is in handy:

```
AT> Y=hex2bm("5f037a");
AT> ? Y;
0x5f
0x3
0x7a
```

Note that the provided text is a sequence of two-byte hexadecimal bytes; for example the byte ”$0x3$” had to be written as the text ”03”.

There is also the corresponding `bm2hex()` function for building an hexadecimal string from a byte matrix.

11.1.2 Characters and Bytes

Strings are sequences of Unicode characters which are encoding in bytes with some scheme. Any string can be converted to a byte matrix using the `text2bm()` function, which by default uses the popular UTF-8 encoding. There is also the reverse `bm2text()` function:

```
AT> ? text2bm("Hello");
0x48
0x65
0x6c
0x6c
0x6f
AT> ? bm2text([0x48;0x65;0x6c;0x6c;0x6f]);
Hello
```

The text2bm() and bm2text() allow an arbitrary encoding by passing its name as a second argument; for more information see the documentation of the Charset Java class.

11.2 Binary operations

Standard operations are provided via bitand(), bitor() and bitxor(). Those operations work on integers and (same size) byte matrices:

```
AT> ? bitand(124,551);
36
AT> ? bitand([0x13;0xab],[0x4c;0x91]);
0x0
0x81
AT> // more human friendly hexadecimal text
AT> ? bm2hex(bitand(hex2bm("13ab"),hex2bm("4c91")));
0081
```

12 Cryptography

The cryptographic support is based on byte matrices.

12.1 Symmetric Key Algorithms

Some common algorithms and operation modes (which we call “methods”) are supported like DES and Triple-DES[9].

```
AT> // setup data for encryption
AT> data = hex2bm("ffffffffffffffff");
AT> // setup key
AT> key = hex2bm("0123456789abcdef");
AT> // encrypt data
AT> edata = encrypt(data, "DES/ECB/NoPadding", key);
AT> ? edata;
0x59
0x73
0x23
0x56
0xf3
0x6f
0xde
0x6
AT> // as hexadecimal string
AT> ? bm2hex(edata);
59732356F36FDE06
```

The `encrypt()`/`decrypt()` functions can receive hex-strings and automatically do the conversion to byte matrices; this is handy for small pieces of data like keys. Note that the output is always a byte matrix, which in the following example is converted with `bm2hex()`:

```
AT> ? bm2hex(encrypt("ffffffffffffffff",
--> "DES/ECB/NoPadding", "0123456789abcdef"));
59732356F36FDE06
```

To decrypt the previous output:

```
AT> ? bm2hex(decrypt("59732356F36FDE06",
--> "DES/ECB/NoPadding", "0123456789abcdef"));
FFFFFFFFFFFFFFFF
```

To find the “check value” of a key, encrypt a block of zeroes:

```
AT> ? bm2hex(encrypt(zeros(8,1),"DES/ECB/NoPadding",
--> "0123456789abcdef"));
D5D44FF720683D0D
```

[9] The encryption/decryption “method” is usually the “transformation” documented in the Java Cipher class documentation. When the transformation is a simple word, it is assumed the algorithm name; otherwise, the algorithm name is assumed to be the first component. We avoid the word “transformation” to allow for non Cipher based schemes in the future.

A Triple-DES test with 16-byte key:

```
AT> ? bm2hex(encrypt(ones(8,1),"DESede/ECB/NoPadding",
--> "aaa1456789a8883f012345675123cdef"));
4F4407A1DA3E29A3
```

This is equivalent to the 24-byte version:

```
AT> ? bm2hex(encrypt(ones(8,1),"DESede/ECB/NoPadding",
--> "aaa1456789a8883f012345675123cdefaaa1456789a8883f"));
4F4407A1DA3E29A3
```

The following example shows the usage of the CBC mode with its initialization vector; padding is added to the output.

```
AT> ? bm2hex(encrypt(ones(8,1),"DESede/CBC/PKCS5Padding",
--> "aaa1456789a8883f012345675123cdef",
--> "0000000000000000"));
4F4407A1DA3E29A353B66A16F1836CF1
```

12.1.1 Key Generation

The `genkey()` function does generate random symmetric keys for the supported algorithms:

```
AT> ? bm2hex(genkey("DES"));
B6DC32C4E98315C8
AT> ? bm2hex(genkey("DESede"));
4FC1688A9E10BC86C832754FD68062DCC275807F62A77379
AT> ? bm2hex(genkey("AES"));
9F54B8C1E9A57193B546B076A5EC1230
```

12.2 Public Key Cryptography

Currently AT only supports the RSA algorithm. Following plain Java requirements the keys (public and private) must be in PKCS#8 DER binary format[10].

As public and private key are too big for direct typing, we assume in the following example that they are stored in the files `/tmp/pubkey.dat` and `/tmp/priv.dat`. Note that encryption does require the public key, but decryption needs the private one:

```
AT> data="This is our secret!";
AT> method="RSA/ECB/OAEPWithSHA1AndMGF1Padding";
AT> edata=encrypt(text2bm(data), method,
--> rawLoad("/tmp/pubkey.dat"));
AT> ? size(edata);
```

[10] For PEM/PKCS#1 file formats, look out the “rsa” and “pkcs8” subcommands of openssl for the required conversion conversion. Note that Java developers may use excellent libraries like Bouncy Castle for a comprehensive range of algorithms and file formats.

```
128     1
AT> ? bm2text(decrypt(edata, method,
--> rawLoad("/tmp/priv.dat")));
This is our secret!
```

12.3 Digital Signatures

Hereon the key pair must be of RSA type.

```
AT> signature=bm2hex(ds_sign(rawLoad("/tmp/testfile.png"),
--> "SHA1withRSA", rawLoad("/tmp/privkey.dat")));
AT> ? signature;
175E09433C118AEECC4744D109923057EC5E70527FF58B902BDE7AB0...
```

Later, the integrity of the data and the authenticity of its signature can be verified with:

```
AT> ? ds_verify(rawLoad("/tmp/testfile.png"),
--> signature, "SHA1withRSA", rawLoad("/tmp/pubkey.dat"));
true
```

12.4 Message Digests

The `digest()` function allows the calculation of hash values for a byte matrix using the specified algorithm:

```
AT> ? bm2hex(digest(text2bm("this is a test"), "SHA-256"));
2E99758548972A8E8822AD47FA1017FF72F06F3FF6A016851F45C398732BC50C
AT> ? bm2hex(digest(text2bm("this is a test"), "SHA-1"));
FA26BE19DE6BFF93F70BC2308434E4A440BBAD02
AT> ? bm2hex(digest(text2bm("this is a test"), "MD5"));
54B0C58C7CE9F2A8B551351102EE0938
```

12.5 Message Authentication Codes

The `mac()` function is in charge. This takes a byte matrix with data, the MAC algorithm, and the bytes for the key[11].

```
AT> ? bm2hex(mac(text2bm("this is a test"), "HmacSHA512",
--> "125518531122441259917423224566651649c...8351741283"));
968BCB56C818E32522095AF6E598FC67E1796AEBBE5F53751117BC...
```

[11] The key length varies with the selected algorithm and with the desired level of resistance to brute force attacks.

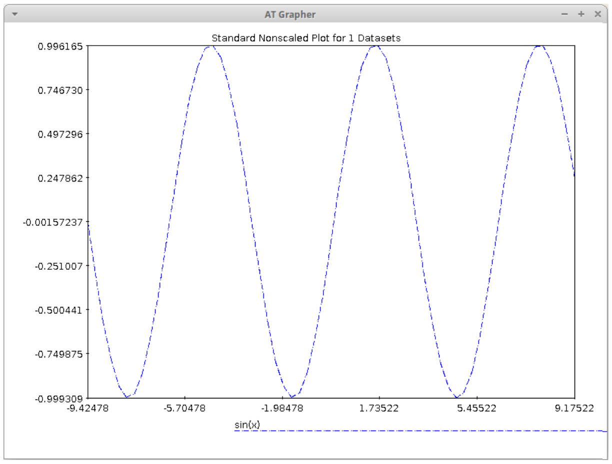
13 Graphics

13.1 X-Y plots

The x-axis and y-axis are provided in two columns of a single matrix[12]:

```
AT> a=range(-3*pi,3*pi,0.3);
AT> plot(a||sin(a), "--;sin(x);");
```

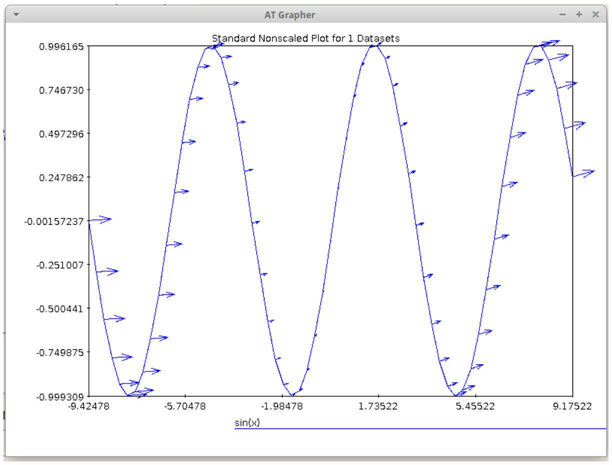
We get the graph:



A graph of the vectors does require the (x,y) origins and the (vx,vy) vectors in a 4-column matrix:

```
AT> plot(a||sin(a)||abs(a)||exp(0.1*a), "vector;sin(x);");
```

The resulting graph being:



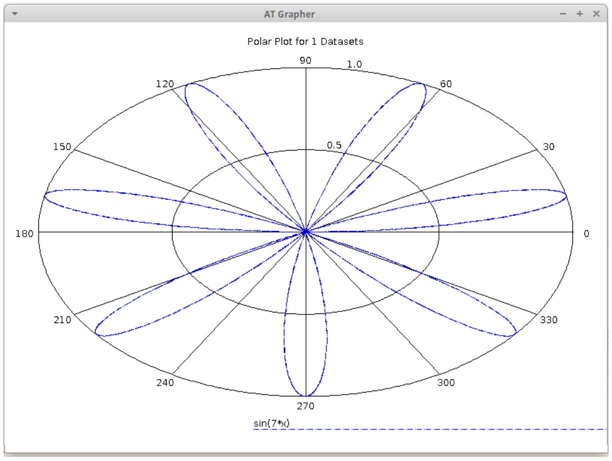
[12] This is unlike Octave where two matrices can be provided for the same purpose.

13.2 Polar plots

Based on (ρ, θ), polar plots are straightforward:

```
AT> a=range(0,2*pi,0.01);
AT> polar([a || sin(7*a)],"--;sin(7*x);");
```

We get:



14 Datasets and regression

The functions of this and the following sections are associated with the "stats" category.

14.1 Random Number Generation

The `rand()` function builds a matrix filled with uniformly distributed random numbers in the interval <0,1>:

```
AT> // 1x3 matrix
AT> ? rand(1,3);
0.0501105       0.941507        0.175867
AT> // column of five numbers in [1,10]
AT> ? integer(rand(5,1)*10.0 + 1);
9
10
8
2
3
```

14.2 Data Samples

Several functions allow the processing of data samples. The data samples are represented by rows of a matrix, and each column corresponds to a variable[13]:

```
AT> A=[1.1,3;1.2,3.4;1.3,3.9];
AT> ? mean(A);
1.20000 3.43333
```

Some functions allow the specification of the "correction bias mode", where "corrected" is the default (division by N-1) but "uncorrected" mode (division by N) may be specified by adding a "U" argument. The following example finds the co-variance matrix using both modes:

```
AT> A=[1.1,3;1.2,3.4;1.3,3.9];
AT> B=[2.1,1,2;2.4,1,2.13;2.6,1.1,2.44];
AT> ? cov(A,B);
0.0250000       0.00500000      0.0220000
0.111667         0.0233333      0.100500

AT> ? cov(A,B,"U");
0.0166667       0.00333333      0.0146667
0.0744444       0.0155556        0.0670000
```

14.3 Linear Regression

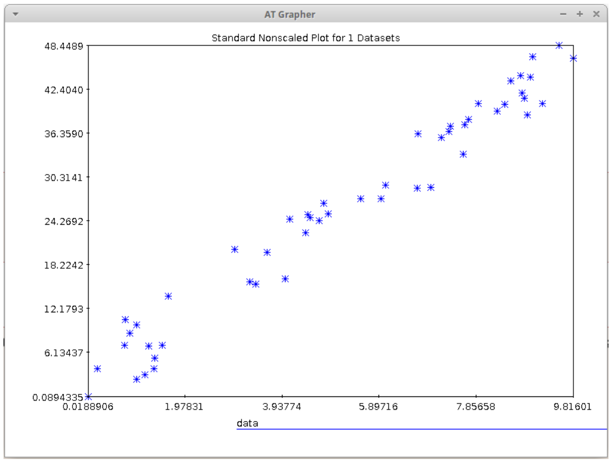
The `slr()` function supports "simple linear regression". For example, given the points:

```
AT> X=rand(50,1)*10.0;
AT> Y=map(X,@(t){4.5*t + 2.15})+10.0*(rand(50,1)-0.5);
AT> plot(X||Y, "*;data;");
```

Here we have 50 numbers (x-axis) distributed in $< 0, 10 >$, and 50 numbers (y-axis) obtained by $y = 4.5t + 2.15$ with some added "noise"[14]. The resulting plot follows:

[13] The functions of this section are intended to work on real numbers; if the data has some "exact" type (like integers or rationals) then it is strongly recommended to force a previous conversion to the real (or complex) type; otherwise, the internal calculations will be excessive, and usually no useful answer will be achieved.

[14] The standard gaussian distribution assumption for the randomness is not considered in this example.



We fit a line with `slr()`, providing the Y and X matrices (in that order![15]) as shown below:

```
AT> fit = slr(Y,X);
AT> ? fit(1,1);
4.57644
2.16441
```

As shown, the `slr()` output returns a column matrix of adjusted coefficients in the position (1,1), so we have the fitted line:

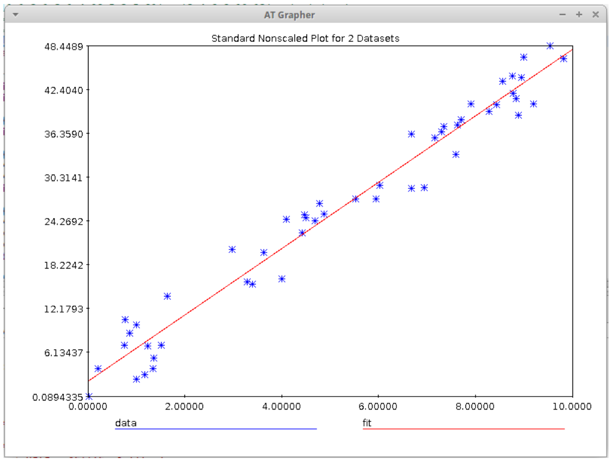
$$\hat{y} = 4.57644\hat{x} + 2.16441$$

this equation is certainly similar to the function used in the generation of the simulated data: `@(t){4.5*t + 2.15}`. Plotting the line:

```
AT> XF=[0.0 ; 10.0];
AT> YF=4.57644*XF+2.16441;
AT> plot(X||Y, "*;data;", XF||YF, "r-;fit;");
```

We get the following combined graph:

[15] The expected order would be X, Y; but here we opted to follow the syntax of Octave.



The element (1,4) of the answer does provide the “coefficient of determination” known as R^2, that informs about the quality of the fit (near to one is better.) See the help of `slr()` for more information.

For multivariate fitness, see the `ols()` function.

14.4 Logistic Regression

We explain this with an example[16]. Fit the binary dependent variable Y (pass exam) to X (hours of study):

Hours	Pass?		Hours	Pass?
0.5	0		2.75	1
0.75	0		3.0	0
1.0	0		3.25	1
1.25	0		3.50	0
1.50	0		4.0	1
1.75	0		4.25	1
1.75	1		4.50	1
2.0	0		4.75	1
2.25	1		5.0	1
2.50	0		5.50	1

We define the column matrices and apply `lrfit()`:

```
AT> x=[0.5;0.75;1.0;1.25;1.50;1.75;1.75;2.0;2.25;
--> 2.50;2.75;3.0;3.25;3.50;4.0;4.25;4.50;4.75;5.0;5.50];
AT> y=[0;0;0;0;0;0;1;0;1;
--> 0;1;0;1;0;1;1;1;1;1;1];
AT> fit=lrfit(x,y);
AT> ? fit(1,1);
-4.07771    1.76099
1.50465     0.628721
```

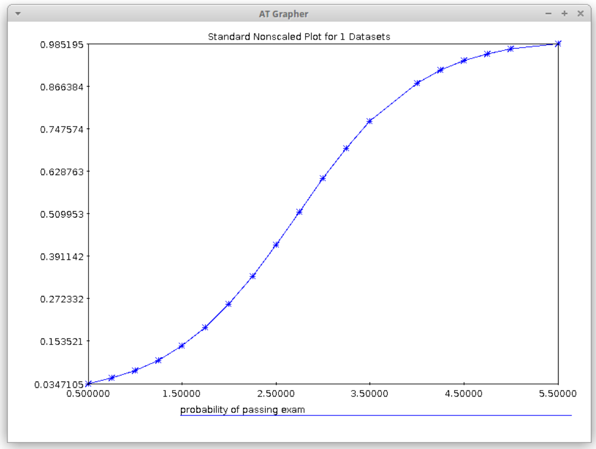
[16] Following https://en.wikipedia.org/wiki/Logistic_regression

So we got the coefficients of the logit:

$$g(x) = -4.07771 + 1.50465x$$

Next the sigmoid function and its plot:

```
AT> f=@(x){1.0/(1.0+exp(-(-4.07771+1.50465*x)))};
AT> yfit=map(x,f);
AT> plot(x||yfit,"-*;probability of passing exam");
```



15 Probability distributions

In the following sections we deal with some probability distributions; the function names follow those used in Octave.

The beta, gamma and error functions used in the implementation of some distributions were adapted from the code in Apache Commons Math. Most distributions were implemented following their analytical expressions as found in Wikipedia and a few ones were adapted from their implementations in Octave.

For each of the provided distributions up to three functions are implemented: the probability density function (*pdf), the cumulative distribution function (*cdf) and the inverse cumulative distribution function (*inv).

15.1 Chi-Square Distribution

Following Wikipedia: "The chi-squared distribution is used primarily in hypothesis testing. Unlike more widely known distributions [...] the chi-squared distribution is not as often applied in the direct modeling of natural phenomena."

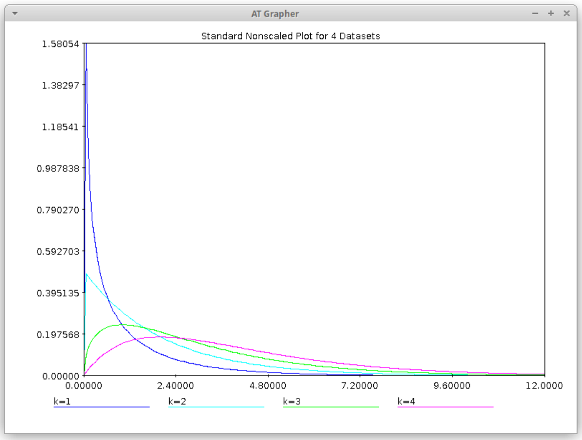
Here we show some trivial calculations:

```
AT> ? chi2cdf(5.0,5.0);
0.584120
AT> ? chi2cdf([4.0,5.0,6.0], 5.0);
0.450584        0.584120        0.693781
AT> ? chi2inv(0.584120,5.0);
5.00000
```

The following commands build a graph for the probability density function:

```
AT> a=linspace(0.0, 12.0, 200);
AT> plot(a||chi2pdf(a,1),"-;k=1", a||chi2pdf(a,2),"-;k=2",
--> a||chi2pdf(a,3),"-;k=3", a||chi2pdf(a,4),"-;k=4");
```

Which results in:



15.2 Pearson Chi-Square Test

This case is taken from Wikipedia:

“For example, to test the hypothesis that a random sample of 100 people has been drawn from a population in which men and women are equal in frequency, the observed number of men and women would be compared to the theoretical frequencies of 50 men and 50 women. If there were 44 men in the sample and 56 women, then

$$\chi^2 = \frac{(44-50)^2}{50} + \frac{(56-50)^2}{50} = 1.44$$

If the null hypothesis is true (i.e., men and women are chosen with equal probability), the test statistic will be drawn from a chi-squared distribution with one degree of freedom (because if the male frequency is known, then the female frequency is determined).”

We setup two vectors: the observed frequencies of the outcomes (here 44 and 56), and the expected value (50 in each case if the choice is fair):

```
AT> obs=[44;56];
AT> ex=[50;50];
```

The `chi2gof()` function (from "Chi-square goodness-of-fit") helps to validate such null hypothesis:

```
AT> ? chi2gof(obs,ex);
chi-square  1.44000
cdf         0.769861
1-cdf       0.230139
```

It means that -given the null hypothesis- there is a probability of $p = 23.013\%$ for this or a more extreme (farther from 50/50) experimental result. It is customary to require $p < 5\%$ for rejecting the null hypothesis[17], so here we can't do that (i.e. we should accept the possibility of a fair/random choice from the people pool.)

15.3 Normal Distribution

From Wikipedia:

"Normal distributions are important in statistics and are often used in the natural and social sciences to represent real-valued random variables whose distributions are not known. [...] The normal distribution is useful because of the central limit theorem. In its most general form, under some conditions (which include finite variance), it states that averages of samples of observations of random variables independently drawn from independent distributions converge in distribution to the normal [...]"

We'll illustrate this distribution with an application.

Let's suppose the members of the chorus of some university have registered their heights in a table. The chorus has 235 members and the university 25000 students. Without doing a full census, we want:

a) to know (approximately) the number of students whose height is larger than two meters

b) having an old stock of about 5000 "small size" t-shirts, a "top height" is needed in order to give away such t-shirts to the smaller students.

We will assume that the heights follow a normal distribution. Using a real dataset taken from R-Datasets, we extract the values from a CSV file[18]; the heights (in inches) are in the second column, which is converted to centimeters:

```
AT> sample=csvread("/home/tester/singer.csv","skip=1");
AT> ? size(sample);
235     3
AT> heights=2.54*sample(:,2);
```

[17] This is an oversimplification for a rather controversial subject; see for example Statistical Significance.

[18] See below the CSV-related functions.

```
AT> h_mean=mean(heights)(1,1);
AT> h_std=std(heights)(1,1);
AT> ? h_mean;
170.937
```

Note that we need the real versions for `h_mean` and `h_std` so they were extracted from the matrices resulting by taking their first and single element.

For the "a" question, we "scale" the 2m=200 cm:

```
AT> z=(200-h_mean)/h_std;
AT> ? z;
2.99163
```

Now we get the probability for $P(x < 2.99163)$, i.e. the probability for a person to have a height smaller than 2.00m:

```
AT> ? normcdf(z);
0.998613
```

Since we are interested in people bigger than 2.00m, we need the complementary probability, which in turn is multiplied by the total population:

```
AT> p=1-normcdf(z);
AT> ? 25000*p;
34.6866
```

So we propose that about 35 persons are bigger than 2.00m.

Another faster way is to leverage the optional arguments for `normcdf()`:

```
AT> ? 25000*(1-normcdf(200,h_mean,h_std));
34.6866
```

The "b" question is the inverse problem: 5000 is the lower $\frac{5000}{25000} = 0.2$ ratio of the population, so we want to know the height for a probability 0.2:

```
AT> ? norminv(0.2);
-0.841621
AT> ? (-0.841621*h_std)+h_mean;
162.760
```

So we would assign the t-shirts to the people with height $\leq 1.62m$.

As before, the previous steps may be abbreviated:

```
AT> ? norminv(0.2,h_mean,h_std);
162.760
```

15.4 Binomial Distribution

From Wikipedia:

"The binomial distribution with parameters n and p is the discrete probability distribution of the number of successes in a sequence of n independent experiments, each asking a yes–no question, and each with its own boolean-valued outcome: a random variable containing a single bit of information: success/yes/true/one (with probability p) or failure/no/false/zero (with probability $q = 1 - p$)."

Let's illustrate it with an example taken from MathWorks:

A Quality Assurance inspector tests 200 circuit boards a day. If 2% of the boards have defects, then:

a) what is the probability that the inspector will find no defective boards on any given day?

```
AT> ? binopdf(0,200,0.02);
0.0175879
```

The answer was a probability of about 1.76%. Note that a defect appearance is considered a statistical "success" tied to the 2% probability.

b) What is the probability that the inspector will find exactly ten defective boards on any given day?

```
AT> ? binopdf(10,200,0.02);
0.00494869
```

c) What is the probability for "up to 3" defective boards?

```
AT> ? binocdf(3,200,0.02);
0.431495
```

The same answer can be obtained (rather slowly) with the pdf:

```
AT> ? binopdf(0,200,0.02) + binopdf(1,200,0.02) +
--> binopdf(2,200,0.02) + binopdf(3,200,0.02);
0.431495
```

d) What is the most likely number of defective boards the inspector will find?

```
AT> defects=range(0,200);
AT> y = binopdf(defects,200,0.02);
AT> m = max(y);
AT> idx = indexof(y,m);
AT> ? defects(idx);
4
```

15.5 Poisson Distribution

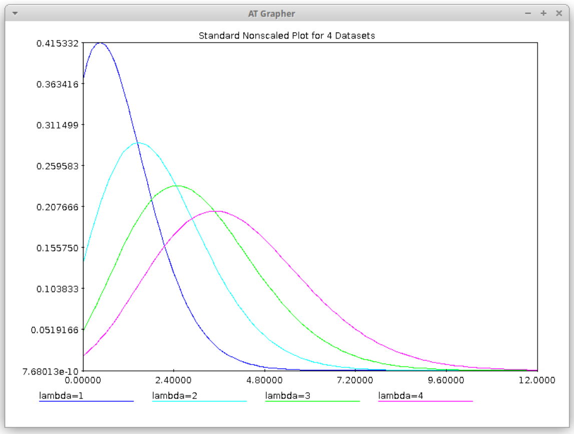
Here we take the example from Wikipedia: given the historical knowledge that the average number of goals per match in the football world cup is 2.5, we want to know the probability for a match to end with exactly zero, one, two, or three goals. Using the Poisson Distribution we get:

```
AT> ? poisspdf(0,2.5)*100;
8.20850
AT> ? poisspdf(1,2.5)*100;
20.5212
AT> ? poisspdf(2,2.5)*100;
25.6516
AT> ? poisspdf(3,2.5)*100;
21.3763
AT> ? poisscdf(3,2.5)*100;
75.7576
```

So, there is a probability of about 8.2% for a goal-less match, 20.5% for single goal matches, and so on; there is a cumulative probability of 75.7% for a match to end with zero, one, two or up to three goals.

A graph for the distribution with the first values for lambda follows:

```
AT> a=linspace(0.0, 12.0, 100);
AT> plot(a||poisspdf(a,1),"-;lambda=1", a||poisspdf(a,2),"-;lambda=2",
--> a||poisspdf(a,3),"-;lambda=3", a||poisspdf(a,4),"-;lambda=4");
```



16 Flow Control

AT provides basic control instructions.

16.1 Conditional Execution

The classical `if(condition)...then...else...` sentence is provided. Note that the “then” part of the sentence is mandatory.

```
AT> if (5.7 < sqrt(33)) then { ? "is true" } else { ? "is false" };
is true
AT> if (5.8 < sqrt(33)) then { ? "is true" } else { ? "is false" };
```

```
is false
```

Since the blocks are simple, this can be simplified to:

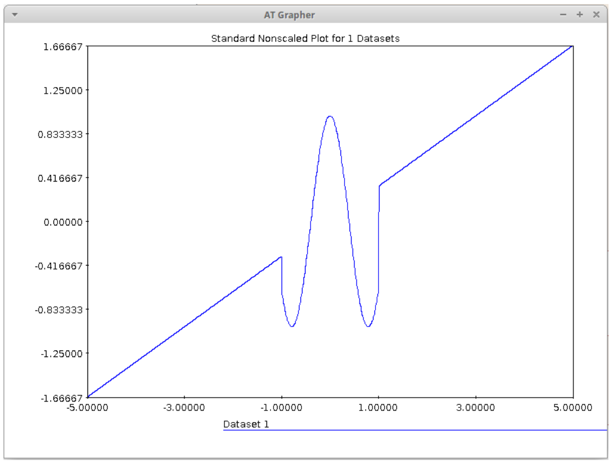
```
AT> if (5.7 < sqrt(33)) then ? "is true" else ? "is false";
is true
AT> if (5.8 < sqrt(33)) then ? "is true" else ? "is false";
is false
```

The following example builds a function defined by subintervals:

```
AT> f=@(x){if (x < -1 || x > 1) then x/3 else cos(4*x)};
```

Which can be plotted this way:

```
AT> x=range(-5,5,0.01);
AT> plot(x||map(x,f));
```



Note that such functions can not be subject to symbolic transformations, but are totally OK for working with AT's "approximation methods".

16.2 Loops

The `for(;;)` sentence allows the repetition of a block of sentences. It follows the syntax of the C-derived programming languages. For example:

```
AT> for(a=10;a < 15 ; a=a+1) ? a*2;
20
22
24
26
28
AT> ? a;
15
```

It can be included inside a function in order to build a complex one. For example, the following function is a re-implementation of Runge-Kutta:

```
// AT-language implementation of Runge-Kutta
test_rk = @(f, start, end, y0, n) {
        t=start;
        w=y0;
        ans = zeros(n+1,2);
        ans(1,1)=start;
        ans(1,2)=y0;
        h=(end-start)/n;
        for(step=1;step<=n;step=step+1) {
                k1=h*f(t, w);
                k2=h*f(t + h/2, w + k1/2);
                k3=h*f(t + h/2, w + k2/2);
                k4=h*f(t + h, w + k3);
                w=w+(k1+2*k2+2*k3+k4)/6.0;
                t=start+step*h;
                ans(step+1,1)=t;
                ans(step+1,2)=w;
        };
        ans;
};
```

This can be called in the following way (compare our previous example of oderk() for solving with Runge-Kutta):

```
AT> ? test_rk(@(t,y){y-t*t+1;}, 0.0, 2.0, 0.5, 10);
0.00000 0.500000
0.20000 0.829293
0.40000 1.21408
0.60000 1.64892
0.80000 2.12720
1.00000 2.64082
1.20000 3.17989
1.40000 3.73234
1.60000 4.28341
1.80000 4.81509
2.00000 5.30536
```

There is also a `while()` loop. Look out the 'help syntax' for more information.

17 Dealing with Files

17.1 Session Data

Session data is stored in text files using a format that tries to avoid losing information at load time. These files are user editable and may contain functions and control sentences.

17.1.1 Saving Session Data

Use the `save(“file-name”, “variable-name”...)` function in order to save one or more variables to a disk file. If no variable is specified, all the variables in the context are stored.

Remember to provide `save()` with the variable names, and NOT with their values: for example, `save(“file”, ”A”)` is correct, but not `save(“file”, A)`.

17.1.2 Loading Session Data and Program Files

The `use()` function loads the definitions stored in a file.

With an external editor a plain text file can be created containing functions and any variable definitions. For example, the function defined above in the “Loops” section (named “`test_rk()`”) is a good candidate for external storage since is pretty tedious to write directly in the AT console. If such function is stored in the file `/tmp/test.at`, it can be loaded into the AT context with:

```
AT> use("/tmp/test.at");
```

17.2 Text Files

To save textual data (strings) stored in a matrix, use the `textwrite()` function; to load from text files, use the `textread()` function:

```
AT> data=["Hello World!";"from AT!"];
AT> textwrite("/tmp/hi.txt", data);
AT> data2=textread("/tmp/hi.txt");
AT> ? data2;
Hello World!
from AT!
```

17.2.1 Controlling the Read Process

The `textread()` function supports some handy options for dealing with text files. The following examples illustrate how to skip lines and set a limit for them; also we show a way to deal with structured data by using format instructions in a way resembling to the traditional `scanf()` from the C language[19].

```
AT> data3=textread("/etc/hosts","skip=2,count=3");
AT> ? data3;
127.0.1.1        fr1120
192.168.1.142    fr1120
192.168.1.131    cool
```

[19] The scanner is currently very primitive and unstable; for example, there is a mandatory space (blank delimiter) after the last %d. This should be fixed in future versions.

```
AT> data4=textread("/etc/hosts","format=%d.%d.%d.%d ,skip=6,count=3");
AT> ? data4;
192     168     1       104
192     168     1       120
192     168     1       136
```

17.2.2 CSV Files

The `csvread()` function allows to read a CSV (comma-separated-value) text file. Also, the delimiter may be set to any regular expression if a comma is not suitable.

For example, the following data file containing an extract from the results of a clinic study[20] contains text lines whose elements are delimited by “tabs”:

```
ID      LOW     AGE     LWT     RACE    SMOKE   FTV     BWT
4       1       28      120     3       1       0       709
10      1       29      130     1       0       2       1021
11      1       34      187     2       1       0       1135
13      1       25      105     3       0       0       1330
15      1       25      85      3       0       0       1474
16      1       27      150     3       0       0       1588
```

It can be read with `csvread()`:

```
AT> x=csvread("/folder/LOWBWT.txt","delim=\t,skip=1");
AT> ? x(1,1);
4
```

17.3 Raw Data Files

Correspond to plain byte sequences and provide the maximum flexibility for the kind of represented information (for example, image files.) As always, in AT such byte sequences are implemented by “byte matrices”.

The following example makes a copy to an image file:

```
AT> x=rawLoad("/tmp/x.png");
AT> ? size(x);
328401  1
AT> rawSave("/tmp/x-copy.png", x);
```

[20] The “Low Birth Weight Study” is analyzed in the book from Hosmer, D.W. Lemeshow, S. “Applied Logistic Regression” 2nd ed. John Wiley & Sons. New York. 2000. The data can be downloaded from the editor web site.

17.4 Directories/Folders

AT does provide the functions pwd() to obtain the current working directory (in a string), chdir(path) to change the current working directory, and dir([path]) to get a matrix containing a listing of a directory.

```
AT> ? pwd();
/tmp/xyz
AT> ? dir();
algorithm-bbq.at    F     115
img1.png            F     328401
```

note that the results of pwd() and dir() must be explicitly printed to be shown.

18 Using the AT library for developing custom applications

18.1 Basic Functionality

The use of the AT library is enabled by adding the AT jar to the classpath. Let's show an example:

```
public void testDummyMatrix() {
        Matrix m1 = new Matrix(4, 4);
        m1.set(1, 2, new Rational(1,2));
        Matrix m2 = Matrix.INSTANCE.magic(4);
        Matrix m3 = m1.add(m2);
        System.out.println(m3);
}
```

The first line creates a 4*4 matrix filled with integer zeroes (i.e. of Java class ATInteger.) The second line sets (resets) the element in the (first row, second column) with a rational value ($\frac{1}{2}$.) Note that matrices are mutable elements via the set() method.

The third line does create a "magic square" in new 4*4 matrix (again, the elements have ATInteger class.) Next, the matrices are added to create a new one, which is finally printed:

```
16    5/2    3     13
5     11     10    8
9     7      6     12
4     14     15    1
```

18.2 Numerical Methods

Several numerical methods which work with real numbers are provided to use the raw java "double" and "Double" datatypes. For example, in the page 331 of

Burden and Faires's Numerical Analysis[21], the following problem is suggested as illustration of the solution of ODE systems with Runge-Kutta:

$$\begin{cases} I_1' = -4I_1 + 3I_2 + 6 \\ I_2' = -2.4I_1 + 1.6I_2 + 3.6 \end{cases} \qquad I_1(0) = I_2(0) = 0$$

Its exact solutions are provided for verification:

$$\begin{cases} I_1(t) = -3.37e^{-2t} + 1.875e^{-0.4t} + 1.5 \\ I_2(t) = -2.25e^{-2t} + 2.25e^{-0.4t} \end{cases}$$

The following JUnit test shows that Runge-Kutta does provide a reasonable approximation:

```
@Test
public void testOdeSystem() {
        // equation system
        XBiFunction<Double, double[], Double> f1 = (t,y)->{
                return -4*y[0]+3*y[1]+6.0;
        };
        XBiFunction<Double, double[], Double> f2 = (t,y)->{
                return -2.4*y[0]+1.6*y[1]+3.6;
        };
        List<XBiFunction<Double, double[], Double>> f =
                new ArrayList<>();
        f.add(f1); f.add(f2);
        // initial conditions
        List<Double> iv = new ArrayList<>();
        iv.add(0.0); iv.add(0.0);
        // exact solutions
        XFunction<Double, Double> ef1 = (t)->{
                return -3.375*Math.exp(-2*t) +
                        1.875*Math.exp(-0.4*t) + 1.5;
        };
        XFunction<Double, Double> ef2 = (t)->{
                return -2.25*Math.exp(-2*t) +
                        2.25*Math.exp(-0.4*t);
        };
        // Runge-Kutta
        Ode i = new Ode();
        double[][] ans = i.rk4(f, 0, 0.5, iv, 5);
        // check results
        for(double[] a : ans) {
                double t = a[0];
                assertEquals(ef1.apply(t), a[1], 1e-4);
                assertEquals(ef2.apply(t), a[2], 1e-4);
        }
}
```

[21] Burden, R., & Faires, J. (2011). Numerical Analysis 9th edn (Boston: Brooks/Cole).

18.3 Expression Evaluation

Another scenery is the evaluation of AT-language expressions. Here the `automata.jar` file must be added to the classpath.

The following example finds the inverse of a matrix using the `invadj()` function:

```
public void testDummyExpr() {
        String txt = "A=[1,0;-1,1]; invadj(A);";
        AT testAt = ConsoleMain.pcSetup();
        Matrix inv = (Matrix)testAt.processText(txt);
        System.out.println(inv);
}
```

As is shown, we had to create an `AT`-class object in order to evaluate arbitrary AT-language expressions; such AT object is created from a static method of `ConsoleMain` when working in a standard command line environment where the output is sent to the console[22]. The result is shown in the console:

```
1        0
1        1
```

An abbreviated version of the same example is shown below:

```
public void testDummyExpr2() {
        String txt = "A=[1,0;-1,1]; ? invadj(A);";
        AT testAt = ConsoleMain.pcSetup();
        AT testAt.processText(txt);
}
```

Note the use of the print "?" command inserted in the expression.

[22] The print (`?`) command may be included in the expression; in that case it will use the configured "output consumer" for display; the ConsoleMain class provides a simple implementation which displays into standard output. In other environments (like the Android version) a different "output" implementation is in order.