Manual for `tk_ode_by_mpfr.rr`

Nobuki Takayama

2020.09.09

Todo, poly_to_mpfr for monomials.

Examples before Sep 2020 can be performed in the folder `num-ht` in `misc-2019` or `num-ht2` in `misc-2020`. Most parts are copied to asir-contrib on Sep 8, 2020 with the name `tk_ode_by_mpfr.rr`.

1 `register_obj(Obj)`

This function associates the asir object `Obj` to a mpfr variable in the C source code and generates a correspondence table entry in `Mpfr_obj`. The function `clear_all()` clears all the entries.

`Obj` is an asir object. If `Obj` is already associated to a mpfr variable `ob_n`, where n is a number, it returns `[ob_n,1]`. 1 means "found". If not, it generates a new mpfr variable `ob_n`, generates an initialization code `mpfr_init2`, and returns `[ob_n,0]`.

Example:

```
->import("tk_asir2mpfr.rr}); //// it may be ommited in the following
                             //// examples.
->register_obj(t);
->register_obj(3);
->Mpfr_obj;
```

If the option `no_init_code=1` is given, it does not generate an initialization code.

Example:

```
-> register_obj(t);
[ob_0,0]
-> output_code();
mpfr_init2(ob_0,PREC); /* register t */
-> register_obj(x | no_init_code=1);
[ob_1,0]
-> output_code();
mpfr_init2(ob_0,PREC); /* register t */
```

See also: `begin_function`

Changelog:

1. 2020.02.14, if `use_init_code()` is called, the initialization code of `mpfr_init2` is stored in `Mpfr_init_code` and is not stored in `AllCode`, which is output by the command `output_code()`. `output_init_code()` returns these initialization codes.

2 new_tmp(Arg)

When `Arg` is 0, it generates a new mpfr variable `tmp_n`, which is assumed not to be readonly and returns `[tmp_n,0]`. If `Arg` is not 0, it only returns `Arg`. Note that the value of `tmp_n` is not set. In order to set it to a number, call `new_mpfr(Num)`.

Notes: This function is mainly used to sum up values in `poly_to_mpfr`, `rpoly_to_mpfr`, ... `register_obj(Obj)` does not generate a new mpfr variable when the asir object `Obj` is assigned to a mpfr variable `ob_n`. The function starting with `new_` always generates a new mpfr variable.

Example: By modifying the following codes by hand, we can evaluate $1+2+\cdots+100$. By `output_code();`, we can see the code generated at any point.

```
->load("tk_asir2mpfr.rr");
-> Sum = new_mpfr(0); // use new_ for variables
[tmp_0, mpfr_set_si(tmp_0,(long) 0,MPFR_RNDD); ]
-> I = new_mpfr(1);  // use new_ for variables
[tmp_1, mpfr_strtofr(tmp_1,"1",10,MPFR_RNDD); ]
-> N = int_to_mpfr(100);  // This is a read-only object. register_obj is used
                          // for read-only object.
[ob_0,mpfr_strtofr(ob_0,"100",NULL,10,MPFR_RNDD); ]
-> Step = int_to_mpfr(1);
[ob_1,mpfr_strtofr(ob_1,"1",NULL,10,MPFR_RNDD); ]
-> Code=sprintf("mpfr_add(%a,%a,%a);",Sum[0],Sum[0],I[0]); add_code(Code);
mpfr_add(tmp_0,tmp_0,tmp_1);
-> Code=sprintf("mpfr_add(%a,%a,%a);",I[0],I[0],Step[0]); add_code(Code);
mpfr_add(tmp_1,tmp_1,ob_1);
-> output_code();
mpfr_init2(tmp_4,PREC);
 mpfr_set_si(tmp_4,(long) 0,MPFR_RNDD);
mpfr_init2(tmp_5,PREC);
 mpfr_strtofr(tmp_5,"1",10,MPFR_RNDD);
mpfr_init2(ob_0,PREC); /* register 100 */
mpfr_strtofr(ob_0,"100",NULL,10,MPFR_RNDD);
mpfr_init2(ob_1,PREC); /* register 1 */
mpfr_strtofr(ob_1,"1",NULL,10,MPFR_RNDD);

mpfr_add(tmp_0,tmp_0,tmp_1);   // Sum =Sum+I
mpfr_add(tmp_1,tmp_1,ob_1);    // I = I+1
```

3 search_obj(Obj)

If the asir object `Obj` is found in `Mpfr_obj`, it returns the mpfr variable name standing for the `Obj` else it returns 0.

4 poly_to_mpfr(Obj,V)

`Obj` is a polynomial or a number. `V` is a list of variables which are assumed to be changed. The variables except `V` are assumed not to be changed (assumed to be constant objects). They are evaluated only when the static variable `first_n` is equal to 1. It returns `[ob_n, string]`. Mpfr variable `ob_n` is assumed to be readonly object.

Internal: `Obj` is translated to a distributed polynomial and mpfr codes are generated. The summation variable is in `Ans` (in the asir source) which stands for the mpfr variable generated by `new_mpfr(0)`.

Example: By setting the value of `mpfr_t ob_3` (t), we can obtain the value of $1/2 + t^2$ in the variable `ob_7`. 1/2, 2 are constant objects in the following code.

```
->import("tk_asir2mpfr.rr");
->poly_to_mpfr(1/2+t^2,[t]);   // t is a variable
[ob_7,mpfr_set(ob_7,tmp_321,MPFR_RNDD); ]
  // [1/2+t^2,ob_7] 1/2+t^2 is associated to ob_7

->gen_declaration_static("\n");
static int first_0=1

->gen_declaration([[t,1.1]],"\n");
mpfr_t ob_3

->gen_declaration_except([[t,1.1]],";");
mpfr_t ob_7;mpfr_t ob_6;mpfr_t ob_5;mpfr_t ob_4;
mpfr_t ob_2;mpfr_t ob_1;mpfr_t ob_0;
->gen_declaration_tmp(";");
mpfr_t tmp_323;mpfr_t tmp_322;mpfr_t tmp_321;

->output_code();
/*begin of poly_to_mpfr(t^2+1/2,[t]); */
    mpfr_init2(tmp_321,PREC);
     mpfr_set_si(tmp_321,(long) 0,MPFR_RNDD);
    if (first_0) {
        mpfr_init2(ob_0,PREC); /* register 1 */
        mpfr_strtofr(ob_0,"1",NULL,10,MPFR_RNDD);
        mpfr_init2(ob_1,PREC); /* register 1/2 */
        mpfr_init2(ob_2,PREC); /* register 2 */
        mpfr_strtofr(ob_2,"2",NULL,10,MPFR_RNDD);
```

```
        mpfr_div(ob_1,ob_0,ob_2,MPFR_RNDD);
        first_0=0;
        }

    /* evaluate t^2 */
    mpfr_init2(ob_3,PREC); /* register t */
    /* Warning. ob_3 is not initialized. */
    mpfr_init2(tmp_322,PREC);
     mpfr_set_si(tmp_322,(long) 0,MPFR_RNDD);
    mpfr_init2(tmp_323,PREC);
        mpfr_init2(ob_4,PREC); /* register t^2 */
        mpfr_pow_ui(ob_4,ob_3,2,MPFR_RNDD);
        mpfr_mul(tmp_323,ob_4,ob_0,MPFR_RNDD);/* Tmp=t^2*ob_0  */
        mpfr_add(tmp_322,tmp_322,tmp_323,MPFR_RNDD); /* Fr_F += Tmp */
        /*Note: The above summation tmp_ var is not used */
    /* Done of t^2 */
    mpfr_init2(ob_5,PREC); /* register (1)*<<2>> */
    mpfr_mul(ob_5,ob_0,ob_4,MPFR_RNDD);
    mpfr_add(tmp_321,tmp_321,ob_5,MPFR_RNDD);
    mpfr_init2(ob_6,PREC); /* register (1/2)*<<0>> */
    mpfr_mul(ob_6,ob_1,ob_0,MPFR_RNDD);
    mpfr_add(tmp_321,tmp_321,ob_6,MPFR_RNDD);


mpfr_init2(ob_7,PREC); /* register t^2+1/2 */
mpfr_set(ob_7,tmp_321,MPFR_RNDD);
/*end of poly_to_mpfr(); */

->poly_to_mpfr((h+t)^2,[t])$
->output_code()$
--- snip ---
```

Changelog: 2020.02.22. This function must be carefully used in a loop. See the code `code_mat_fac_mpfr` to generate a code of matrix factorial. It is safer to use `rpoly_to_mpfr` and `use_init_code()`, `output_init_code()`. The object associated to a number in Z or Q is constructed in the block of an initialization.

Notes: Other functions `rat_to_mpfr`, `rpoly_to_mpfr`, `ratnum_to_mpfr`, `int_to_mpfr` are assumed to be evaluated only once.

See also: `init_code`

5 rat_to_mpfr(Obj)

It is used to evaluate rational number or rational function by mpfr. It generates a read-only mpfr variable standing for `Obj`.

6 test1()

It generates the C code `t0.c` to get values of $3t^2$ for $t = 1+k/10^5$, $k = 0, 1, 2, \ldots$. Note that the global variables must be initialized. If they are not initialized, it returns an error.

Example: See the code of `test1a(F)` in the package.

```
-> load("tk_asir2mpfr.rr");
-> test1()$
3                 // 3*t^2,  t=1
3.0000600003      // 3*t^2,  t=1+1/10^5
3.0001200012      // 3*t^2,  t=1+2/10^5
3.0001800027      // 3*t^2,  t=1+3/10^5
Write to t0.c

=> gcc t0.c -lmpfr -lgmp
=> ./a.out
```

7 init_code(L)

`L` is a list of rules. It is used to set values to constant objects.

Example: h and a are constant object and t will be a variable. We want to get the value of $a + th$.

```
--> init_code([[h,1/10^5],[a,2]]);
mpfr_set(ob_0,ob_1,MPFR_RNDD); /* set variable h=1/100000 */
mpfr_set(ob_4,ob_5,MPFR_RNDD); /* set variable a=2 */

--> poly_to_mpfr(a+t*h,[t]);
[ob_9,mpfr_set(ob_9,tmp_30,MPFR_RNDD); ]
--> search_obj(t);
ob_6
--> output_code();
[2273] output_code();
mpfr_init2(ob_0,PREC); /* register h */
// ---snip
mpfr_init2(ob_4,PREC); /* register a */
mpfr_init2(ob_5,PREC); /* register 2 */
mpfr_strtofr(ob_5,"2",NULL,10,MPFR_RNDD);
mpfr_set(ob_0,ob_1,MPFR_RNDD); /* set variable h=1/100000 */
mpfr_set(ob_4,ob_5,MPFR_RNDD); /* set variable a=2 */

/*begin of poly_to_mpfr(h*t+a,[t]); */
    mpfr_init2(tmp_30,PREC);
     mpfr_set_si(tmp_30,(long) 0,MPFR_RNDD);
```

```
    if (first_0) {
        /* h is already evaluated as ob_0 */
        /* a is already evaluated as ob_4 */
        first_0=0;
        }

    /* evaluate t */
    mpfr_init2(ob_6,PREC); /* register t */
    /* Warning. ob_6 is not initialized. */
  // --- snip
mpfr_init2(ob_9,PREC); /* register h*t+a */
mpfr_set(ob_9,tmp_30,MPFR_RNDD);
/*end of poly_to_mpfr(); */
```

If the optional variable (list) `var` is given, the variables are assumed to be a variable and will not be initialized.

See also: `dep_vars`

Example:

```
--> use_init_code();

--> init_code([[d_0,t+1],[d_1,h*t]] | var=[t]);
mpfr_set(ob_0,ob_5,MPFR_RNDD); /* set variable d_0=t+1 */
mpfr_set(ob_6,ob_9,MPFR_RNDD); /* set variable d_1=h*t */
--> output_init_code();
--- snip
mpfr_init2(ob_2,PREC); /* register t */ //// no number is set to t
--- snip
mpfr_init2(ob_8,PREC); /* register (h)*<<1>> */
mpfr_init2(ob_9,PREC); /* register h*t */

--> output_code();
/* register d_0 to ob_0 in init_code */
/*begin of poly_to_mpfr(t+1,[t]); */
     mpfr_set_si(tmp_10,(long) 0,MPFR_RNDD);
    if (first_0) {
        /* register 1 to ob_1 in init_code */
        mpfr_strtofr(ob_1,"1",NULL,10,MPFR_RNDD);
        first_0=0;
        }

    /* evaluate t */
    /* register t to ob_2 in init_code */
    /* Warning. ob_2 is not initialized. */
     mpfr_set_si(tmp_11,(long) 0,MPFR_RNDD);
        mpfr_mul(tmp_12,ob_2,ob_1,MPFR_RNDD);/* Tmp=t^1*ob_1  */
```

```
        mpfr_add(tmp_11,tmp_11,tmp_12,MPFR_RNDD); /* Fr_F += Tmp */
        /*Note: The above summation tmp_ var is not used */
    /* Done of t */
--- snip
/* register t+1 to ob_5 in init_code */
mpfr_set(ob_5,tmp_10,MPFR_RNDD);
/*end of poly_to_mpfr(); */
/* register d_1 to ob_6 in init_code */
/*begin of poly_to_mpfr(h*t,[t]); */
--- snip
/* register h*t to ob_9 in init_code */
mpfr_set(ob_9,tmp_13,MPFR_RNDD);
/*end of poly_to_mpfr(); */
mpfr_set(ob_0,ob_5,MPFR_RNDD); /* set variable d_0=t+1 */
mpfr_set(ob_6,ob_9,MPFR_RNDD); /* set variable d_1=h*t */
```

8 clear_all()

It clears all global variables of the package. (a) Registerd objects are forgotten. (b) The codes generated are discarded.

9 begin_function(Fname,ArgList,OtherArgList), end_function(Return

`Fname` is the function name. `ArgList` is a list of mpfr variables. `OtherArgList` is a list of strings which are passed to the C code.

Example:

```
-> Begin=begin_function("foo",[t],["int i"]);
int foo(mpfr_t ob_0,int i) {
-> End=end_function(i);
  return(i);
}
```

Example: The following code generates a function of evaluating $a + b$. Note that $a + b$ can be changed to any polynomial in a and b in the following inputs.

```
clear_all();
-> Begin=begin_function("myadd",[r,a,b],[]);
 int myadd(mpfr_t ob_0 /* r */,mpfr_t ob_1 /* a */,mpfr_t ob_2 /* b */)
-> R=poly_to_mpfr(a+b,[a,b]); //// replace a+b by any polynomial.
-> D0=gen_declaration_except([[r,0],[a,0],[b,0]],";\n");
      //// arguments should be initialized out of the function.
      //// r, a, b are arguments for a function
-> D1=gen_declaration_tmp(";\n");
      //// declare mpfr variables with the name tmp*
-> D2=gen_declaration_static(";\n");
      /// declare static variables with the name first*
```

```
-> set_obj_mpfr(r,R[0]);  ////set the return value to the argument r=ob_0
-> End=end_function(0);
-> Begin+D0+D1+D2+output_code()+End;
int myadd(mpfr_t ob_0 /* r */,
          mpfr_t ob_1 /* a */,mpfr_t ob_2 /* b */) {
  mpfr_t ob_6;
--- snip
  mpfr_init2(ob_6,PREC); /* register a+b */
  mpfr_set(ob_6,tmp_6,MPFR_RNDD);
  /*end of poly_to_mpfr(); */
  mpfr_set(ob_0,ob_6,MPFR_RNDD); /* r <- ob_6 */
  return(0);
}
```

10 set_obj_mpfr(Obj,M), set_mpfr_obj(M,Obj), set_obj_obj(Obj,ObjM), set_mpfr_mpfr(R,M)

`set_obj_mpfr(Obj,M)` assigns `M` to `Obj`. `set_mpfr_obj(M,Obj)` assigns `Obj` to `M` `set_obj_obj(Obj,ObjM)` assigns `ObjM` to `Obj`. `set_mpfr_mpfr(R,M)` assigns `M` to `R`. When the asir object `Obj*` is given, it looks for the corresponding mpfr variable which is used for the assignment.

Example:

```
-> clear_all();
-> R=poly_to_mpfr(x^2+y^2,[x]);
[ob_7,mpfr_set(ob_7,tmp_5,MPFR_RNDD); ]
-> T=rpoly_to_mpfr(t);
[ob_8,1]
-> set_obj_mpfr(t,R[0]); //// set_mpfr_mpfr(T[0],R[0] is OK.
mpfr_set(ob_8,ob_7,MPFR_RNDD);
-> output_code();
/*begin of poly_to_mpfr(x^2+y^2,[x]); */
--- snip
mpfr_init2(ob_7,PREC); /* register x^2+y^2 */
mpfr_set(ob_7,tmp_5,MPFR_RNDD);
/*end of poly_to_mpfr(); */
/* evaluate t */
mpfr_init2(ob_8,PREC); /* register t */
/* Warning. ob_8 is not initialized. */
--- snip
/* Done of t */
mpfr_set(ob_8,ob_7,MPFR_RNDD);  //// assignement
```

11 gen_declaration(Rule,Sep), gen_declaration_except(Rule,Sep), gen_declaration_tmp(Rule,Sep), gen_declaration_static(Sep)

They return codes to declare variables already registered by translating functions from asir objects to mpfr objects, e.g., `poly_to_mpfr()`.

See also: `begin_function()`

12 gen_declaration_ob(Sep)

It can be used only when `use_init_code()` is executed. It outputs declarations of all `ob` variables.

See also: `gen_declaration_tmp(Sep)`

13 Code for debug 1

Let us solve $y' = \frac{2}{t}y$, $y(t_0) = y_0$ by the Euler difference scheme. The recurrence is

$$y_{n+1} = y_n + h\frac{2}{t_n}y_n = \left(1 + h\frac{2}{t_n}\right) y_n$$

where h is a small step number. Then the matrix factorial

$$M_f(n) = \left(1 + h\frac{2}{t_0 + (n-1)h}\right) \cdots \left(1 + h\frac{2}{t_0 + h}\right) \left(1 + h\frac{2}{t_0}\right)$$

gives an approximate solutin $y_n = M_f(n)y_0 \sim y(t_0 + nh)$. The following code has been used to debug codes of the matrix factorial.

```
-> load("tk_mat2mpfr.rr");
-> clear_all();
-> QQ=[newmat(1,1,[[1+2*d0*h]]),[[d0,1/t]]];
   //// Elements of the matrix should be polynomials.
   //// Denominators are given by the rule format, e.g., [[d0,1/t]]
   //// See rk4_mat(A)
-> Code=code_mat_fac_mpfr(QQ)$
-> Code;
```

The `Code` contains the function `mat_fac(t0,h,n,ans,argcv,argv[])` which gives $M_f(n)$ in `ans` (as a vector). `argv[0]` is the last time $t_0 + nh$ and `argv[1]` is h.

Changelog: 2020.02.22. `rpoly_to_mpfr` is used rather than `poly_to_mpfr` is the for-loop of the matrix factorial.

14 prod_atomic_matrix_mpfr(M,P,I,J,Prod_IJ,TT)

It returns (I, J) element of the matrix product MP. The elements of M and P must be atomic. In other words, M_{pq} and P_{pq} must be variables such as `ob_n` or `tmp_m`. The expression of MP_{IJ} in these variables is `Prod_IJ`. `TT` is a work area.

Example:

```
-> load("tk_mat2mpfr.rr");
-> QQ=[ [[t,1],[2,t]], []];
-> Code=code_mat_fac_mpfr(QQ)$
-> Code;
```

Matrix size `N` should be given.

Changelog: The first version is written in 2020.02.23.

15 test_template2(QQ,N,TT,HH,Times)

It generates a C code `t0.c` to test a code of the matrix factorial of

$$M(t+h(r-1))M(t+h(r-2))\cdots M(t+h)M(t) \tag{1}$$

where t =`TT`, h =`HH`, r =`Times`. `N` is the size of the square matrix M. The matrix M is given by `QQ[0]` with the replacement rule `QQ[1]`. Elements of `QQ[0]` must be a polynomial.

`TT` and `HH` must be rationals.

It returns the matrix factorial value evaluated by asir.

Example:

```
-> load("tk_mat2mpfr.rr");
-> test2m();
Code generation is done.
Evalute matrix factorial by asir... Time of
mac_fac_by_asir = [ 0.001303 0 10752 0.00130296 ]
    // asir matrix factorial takes 0.001s. Value by asir is embeded in t0.c
    // and is used to comare it with the result by mpfr C code in t0.c
Done.
[ 6 0 ]
[ 0 60 ]
Written to t0.c
-> quit();

gcc -I$OpenXM_HOME/include -L$OpenXM_HOME/lib t0.c -lmpfr -lgmp
time ./a.out
--- snip
```

where

```
def test2m() {
  N = 2;
  QQ=[ [[t,0],[0,t+2]], []];
  return test_template2(QQ,N,1,1,3);
}
```

```
cc t0.c -lmpfr -lgmp
```

A comparison of the result by the C program and by an asir function is automatically performed.

Changelog: The first version is written in 2020.02.23 tested by `test2m()`.

16 code_mat_fac_mpfr(MatRule)

It generates a C function

```
int mat_fac(mpfr_t ob_0 /* t */,
            mpfr_t ob_1 /* h */,
            int n,
            mpfr_t ans[],int *argcv, mpfr_t *argv)
```

to evaluate the matrix factorial (1). The result is stored in the variable `ans`. The matrix factorial M $(r \times r)$ is flatten as a vector `ans` in the format

$$(M_{11}, M_{12}, \ldots, M_{1r}, M_{21}, M_{22}, \ldots, M_{2r}, \ldots, M_{rr})$$

This function is the main part of `test_template2`. `MatRule` is the format of the function `rk_mat2` of `ak2.rr` (see defusing.tex). `t` and `h` are reserved variables. Other variables must be constants and the matrix is embedded in the program.

```
--> load("tk_mat2mpfr.rr")$
--> QQ=rk4_mat([[0,1],[t,0]]);  // Airy diff eq.
[ (1/12*d1^2*d0+1/12*d2*d1^2)*h^3*t^2+...  ---snip ],
  [[d0,1],[d1,1],[d2,1]]]
--> clear_all();
--> code_mat_fac_mpfr(QQ);
// It returns a C code in a string.
```

Changelog: 2020.03.21 New arguments `int *argcv` and `mpfr_t opt[N*N+2]` are added return the values of the last t and h. Note that when $h = 0.00001$, mpfr translates this value to, e.g., $h = 1.00000000000000008180305391403130954586231382563710212707519531$05 (by setprec(100);). It causes a difference to the last t. Check these things by `test3m()`. Files: `tk_mat2mpfr.rr`.
Todo-2020-09-09-a. Store intermediate matrix factorials:

```
int intermediatec;
mpfr_t intermediatev[];
code_mat_fac_mpfr2
```

Todo-2020-09-09-b. matrix factorial for inhomogeneous case.

See also: Ref: 2020-09-07-airy.c.goodnotes(private note)

17 code_compare_mat(P)

It generates a C function `compare_mat(ans)` to compare the result by mpfr and that by asir `mat_fac_by_asir(QQ,TT,HH,Times)`. If the relative error is larger than $2^{-p/2}$, the C code output an error where p is `PREC` (precision by bit).

```
-> load("check_mpfr.rr");
-> test4m();

gcc t0.c -lmpfr -lgmp
time ./a.out
```

The `test4m()` stands for `rk_airy(1000)` of `ak2.rr`.

Changelog: 2020.03.17, tested by `test3m()` and `test4m()` of `check_mpfr.rr`.

Changelog: 2020.03.21 Add `HH` in the format like `0.00001` instead of `1/10^5` under proper `setprec()` to make the comparison works. Files: `tk_mat2mpfr.rr`.

18 parts.c

It contains C functions to implement defusing method, especially for H_n^k. `test2-ak2.rr`, `ev_ak2.rr` are refered for `eigen_nonsym` and `fit_init`.

```
Translate ans to the double matrix ans_d:
int mat_get_d(double ans_d[N][N],mpfr_t ans[N][N]);

Obtain the eigenvalues and the eigenvectors:
int my_gsl_eigen_nonsymmv(double *data, int n,
   double eigen_re[],double eigen_im[],
   double eigen_vec_re[][N],double eigen_vec_im[][N]);
```

When `n` is 2, the matrix for which eigenvalues and eigenvectors are computed is

$$\begin{pmatrix} d_0 & d_1 \\ d_2 & d_3 \end{pmatrix}$$

where d_i is `data[i]`. In other words, the matrix is linearized by row/column.

```
-->load("ak2.rr");
-->rk_airy(1000);
NewY=(1/24*a2*a1^2*a0*h^4+(1/12*a1^2*a0+1/12*a2*a1^2)*h^3+(1/6*a1*a0+1/6*a1^2+1/6*a2*a1)*h^2
[ 1.17283 1.08669 ]
[ 0.535208 1.34853 ]
ratio = -1.37172116419844834727194023524703545648063187331 3
ratio of eigenval  0.49301023999809948101905632583903668732 = -1.59848295160668155456028142
ratio of eigenval  2.02835543538376591232625727318163959 47 = 1.27020591401662965848572418272
Airy at t=0.999 with iv=0.355 will be [ 0.17512821324620175155984252766504603798981173160 27
Airy at t=0.999 with iv=0.355028 will be [ 0.17512821324620175155984252766504603798981173160
Airy at t=0 with iv=0.355 will be [ 0.35477791448754843031888837451320786340773083206 68 -0.2
Airy at t=0 with iv=0.355028 will be [ 0.35477791448754843031888837451320786340773083206 68
[[0.49301023999809948101905632583903668732,[-1.086687635685808304419938699210587371721347113
```

Changelog: 2020.03.17

See also: 2020-09-01-2020-01-parts.c.goodnotes, reviewing this program on goodnotes.

19 ox_mat_fac_mpfr

It is an ox server, which evaluates matrix factorials by mpfr. See `matfac.rr` on the usage.

```
Pid=ox_launch(0,pwd()+"/ox_mat_fac_mpfr");
ox_cmo_rpc(Pid,"mat_fac_mpfr",[TT=1,HH=0.001,Times=1000]);
Ans=ox_pop_cmo(Pid); Ans=eval_str(Ans);
```

`TT` (start time) and `HH` (step) are machine double. When the bigfloat is the default, the argument should be given as

```
ox_cmo_rpc(Pid,"mat_fac_mpfr",map(deval,[TT=1,HH=0.001,Times=1000]));
```

The code for the factorial is embedded in the function `mat_fac.c`. In the current directry structure, this embedded code is stored in the folder `C_mat_fac/` and included in `mat_fac.c`.

The static variables `static int first...` should not be used in the function and `int first...` should be used, because the variables `mpfr ob...` are broken in the second call of function. See `gen_declaration_static` in `tk_mat2mpfr.rr`. Use `no_static` option.

Changelog: 2020.03.18

20 hkn_y in test3-ev.rr

Note on 2020.03.19 to review. `solve_real_ode_ev()` in `ev_ak2.rr` (`ev` stands for eigenvector) and `solve_rk_mat2()` in `ak2.rr` is the main part of ODE

solver. rk_mat2() generates a matrix for the matrix factorial. from a system of ODE. solve_rk_mat2() computes the matrix factorial. The return value is [[T,V],Vm,Mat] where T is the time and V is the dependent value at the time T. Vm is a list of intermediate values of the dependent variable vector. See the option every.

The automatic defusing method corrects the initial value at every Defused_step steps. See hkn_y_multi_defused() in test3-ev.rr.

test1029a() in test3-ev.rr is the successful case. Todo, make this function faster by the ox_mat_fac_mpfr server.

21 Computing the matrix factorial by the ox server

```
/* cf. rk_airy() in gauge.rr */
def test_omfm1() {  // in ak2.rr
  start_omfm();
  Iv=number_eval([0.355028053887817*exp(0),-0.258819403792807*exp(0)]);
  Iv=newvect(2,Iv);
  return solve_rk_mat2(P=0,From=0,To=5,Iv,HH=0.001 | func="ox_mat_fac_mpfr_airy");
}
```

When you want to add new fast matrix factorial by mpfr, add a C code to mat_fac.c. The prototype declaration and a calling sequece should be added to ox_mat_fac_mpfr.c. Do grep ox_mat_fac_mpfr_airy.

test_omfm1() compares the output with that by Mathematica at $t = 5$. Wwhy smaller h and larger MPFR_PREC do not give more accurate answer? See 2020.03.22.

Changelog: 2020.03.19

Changelog: 2020.03.21, new return values [t,h]. See code_mat_fac_mpfr(). Files: mat_fac.c, C_mat_fac/matfac-airy.c.

22 Generating mpfr code for H_n^k in test1029a().

```
load("test3-ak2.rr");
hkn_y();  // k=10, n=1
AK_MatRule;  // use this to generate code_mat_fac_mpfr,
// It is bsaved in C_mat_fac/hkn-k10-n1-rk_mat2.ab

AK_ode_sys;
```

Solving $H_1^{10}(x, y)$ with respect to y by the defusing method. C_mat_fac/matfac-hkn-10-1.c is added to the server.

```
-> load("test3-ak2.rr");
-> test0320a();
```

Check if the precision is improved by increasing MPFR_PREC.

```
-> load("/Users/nobuki/tmp/misc-2019/09/num-ht/tk_mat2mpfr.rr");
All global variables of the package are cleared.
-> setprec(100);
15
-> test3m(|rational=1);
Code generation is done. Evalute matrix factorial by asir... Time of mac_fac_by_asir = [ 0.0
Done.
[ 2500175003/2500025000 ]
Written to t0.c

=> MPFR_PREC 32
 ./a.out
1.0000599990
small eps=1.5258789062e-5
Diff (not relative) of (0,0): 1.3969838619e-9

=> MPFR_PREC 64
./a.out
1.00006000059999399973
small eps=2.328306436538696289 06e-10
Diff (not relative) of (0,0): 2.168404344971008868 01e-19

=> MPFR_PREC 128
 ./a.out
1.000060000599994000064907681403034080060
small eps=5.42101086242752217003726400434970855712 8e-20
Diff (not relative) of (0,0): -4.90828139703414006463536028438407280292 2e-21

=> MPFR_PREC 256
 ./a.out
1.000060000599994000064907681403034080072962172366525968536836621665142211762793
small eps=2.93873587705571876992184134305561419454666389193021880377187926569604314863681 7e-
Diff (not relative) of (0,0): -4.90828139703414007236217836646596913683062172514161176880076
err, rel=1
```

Notes: 2020.03.21. Why the precision does not get better? The following is a code in t0.c

```
  mpfr_set_d(t,(double) 1,MPFR_RNDD); mpfr_set_d(h,(double) 1e-05,MPFR_RNDD);
  mat_fac(t,h,1,ans);
```

The variable h stands for ob_1. It's value is h=1.0000000000000008180e-5 (MPFR_PREC 64) and you find junk numbers 8180.

These also generate junks as above.

```
mpfr_strtofr(h,"1e-5",NULL,10,MPFR_RNDD);
mpfr_strtofr(h,"0.00001",NULL,10,MPFR_RNDD);
mpfr_strtofr(h,"100000",NULL,10,MPFR_RNDD);  mpfr_div(h,t,h,MPFR_RNDD);
```

It is not possible to set the exact `h`, then t should be returned.

Changelog: 2020.03.21. New return values are the last t and h from the server. See `code_mat_fac_mpfr()`. Files: `mat_fac.c`, `C_mat_fac/matfac-airy.c`, `matfac.rr`(test client)., `ak2.rr`

23 matfac.rr, mat_fac.c

Add codes so that the server accepts a list of bignum's. `mpfr_init2` should be done when constructing an array of `mpfr_t`. Doing it in `cmo2bigfloat` does not work.

```
mpfr_t *d;
d = (mpfr_t *) GC_malloc(sizeof(mpfr_t)*(*length+1));
cellp = list_first((cmo_list *)c);
entry = cellp->cmo;
for (i=0; i<n; i++) {
  mpfr_init2(d[i],MPFR_PREC);
  cmo2bigfloat(d[i],entry);
  ...
}
```

`mat_fac.c` is changed to use the function pointer for a matrix factorial function generated by `code_mat_fac_mpfr()` as

```
int call_mat_fac_mpfr_generic(int (* func)(mpfr_t,mpfr_t,int,mpfr_t *,
                              int *, mpf_t *),int nn);
```

where `nn` is the size of the matrix.

Changelog: 2020.03.22. `check_mpfr.rr` (`test4m()` is for Airy). `tk_mat2mpfr.rr` (`test3m()` for check 1×1). `matfac.rr` (test for Airy). `mat_fac.c`, `C_mat_fac/*.c`, `ox_mat_fac_mpfr.*` (changed C codes for the function pointer and getting bignum list). `ak2.rr`, `test3-ak2.rr` (for `test0320a()` and `test0320a0()`, to check the time is properly evaluated.)

```
-> load("matfac.rr");
2+3 = 5  // test of add_int
[[2.730883017890067867106324635448221764280469836108218582059696198979369201406 51e+00,3.611
[t_last,h]=[2.000000000000000041633363423443370265886187553405761718 75e+00,1.000000000000000
[[2.73088,3.61107],[3.25952,4.67627]]
[AiryAi,AiryAi'] at t_last = [ 0.0349241304232732 -0.053090384433657 ]
N[AiryAi[2.00000000000000004163336342344 3],30] -> 0.034924130423274376924990811 3752
---------------
```

```
0
[[1.17229997005792741585458689048204549116843012040768083416706938243616303720507e+00,1.0853
[[1.1723,1.08534],[0.534035,1.34744]]
[t_last,h]=[1.00000000000000002081668171172168513294309377670288085937 5e+00,1.0000000000000
[AiryAi,AiryAi'] at t_last = [ 0.135292416312881 -0.159147441296793 ]
 N[AiryAi[1.00000000000000002081668],30]-> 0.135292416312881412211226 1
```

24 Demo 1

```
-> import("tk_mat2mpfr.rr");   // it generates C codes for mpfr
-> import("ak2.rr");  // for rk_mat2()
-> Q = rk_mat2(newmat(2,2,[[0,1],[t,0]]))$
-> size(Q[0]);
  [2,2]
-> Code=code_mat_fac_mpfr(Q)$
```

The airy differential equation $y'' - ty = 0$ is translated to the system

$$\frac{dF}{dt} = PF, \quad \begin{pmatrix} 0 & 1 \\ t & 0 \end{pmatrix}$$

with the base $F = (y, y')^T$. `Q` is a matrix and rules standing for the 4th order Runge-Kutta scheme. `Code` is a C function to evaluate the matrix factorial of `Q` of `n` times. The C function has the following prototpye declaration.

```
int mat_fac(mpfr_t ob_0 /* t */,mpfr_t ob_1 /* h */,int n,
            mpfr_t *ans,int *argcv,mpfr_t *argv)
```

Q is written in comment lines. `t` and `h` are t and h respectively. `n` is the times of the matrix factorial. The matrix size `M` and `N` (`M` $\times$ `N` matrix) should be given by the `define` macro when you use this C code. The size information is in `Q` as in the example above. `M` = `N` is assumed in the current implementation.

The result is stored in `ans`. `ans[i*M+j]` is the (i, j) element of the matrix factorial. `argv[0]` is the time t of the next step.

Demo of $H_n^k(1, t)$, $k = 10, n = 1$.

```
-> load("check_mpfr.rr");
-> test6m();

// The matrix factorial of $1000$ times.
```

Changelog: 2020.03.24, demo.

25 Example: constructing a solver of Airy differential equation with MPFR and the matrix factorial

The Airy differential equation is

$$y'' - ty = 0. \tag{2}$$

The Airy function can be approximately obtained by the initial value at $t = 0$

$$(y, y') = (0.355028053887817239260063186004, -0.258819403792806798405183560189).$$

We will explain how to construct a solver of the Airy differential equation by the matrix factorial with MPFR and how to install it to the OpenXM server `ox_mat_fac_mpfr`.

25.1 Generating a code for MPFR

```
-> load("ak2.rr");
-> P = single2sys0(dt^2-t,[t,dt]);
[ 0  1 ]
[ t  0 ]
```

The matrix `P` is the coefficient matrix of the system of differential equation for the dependent vector valued function $F = (y, y')^t$. We have

$$\frac{dF}{dt} = PF, \quad P = \begin{pmatrix} 0 & 1 \\ t & 0 \end{pmatrix}$$

The following inputs generate a C code utilizing the MPFR library.

```
load("tk_mat2mpfr.rr");
-> MatRule = rk_mat2(P);
-> S = code_mat_fac_mpfr(MatRule);
// copy the code stored in S and paste it to a new file, e.g., myairy.c or
-> util_write_string_to_a_file("myairy.c",S);
```

`MatRule` contains a matrix generated by the 4th order Runge-Kutta method. The scheme is

$$F_{k+1} = Q(t_0 + kh)F_k, \quad F_0 = F(t_0) \tag{3}$$

where h is the step size. The variable `MatRule` is a list of $\tilde{Q}$ and a substitution rule of the variable in $\tilde{Q}$. By applying the substition rule to $\tilde{Q}$, we obtain Q in (3). The rule is used to avoid big denominator polynomials.

25.2 Installing to the OX server

1. Move the file `myairy.c` to the folder `C_mat_fac/`.
2. Add `myairy.o` to `C_mat_fac/Makefile` and `Makefile`.

3. Edit the file `myairy.c`. The function `mat_fac()` in `myairy.c` should be renamed. For example, we change the name to `myAiryMatFac()`. Add the prototype declaration of `myAiryMatFac()` to `ox_mat_fac_mpfr.h`

4. Add the following lines to the function `sm_executeFunction` of the ox server `ox_mat_fac_mpfr.c`.

```
}else if (strcmp(func->s,"myAiry")==0) {
    call_mat_fac_mpfr_generic(myAiryMatFac,2);
}
```

   where 2 is the rank of the Airy differential equation.

See also: `C_mat_fac/matfac-hkn-10-1.c`

25.3 Testing the code

```
import("names.rr")$
Pid=ox_launch(0,pwd()+"/ox_mat_fac_mpfr");
Iv=number_eval([0.3550280538878172392600631860004*exp(0),
    -0.258819403792806798405183560189*exp(0)])$ Iv=newvect(2,Iv)$
ox_cmo_rpc(Pid,"myAiry",[TT=0,HH=0.001,Times=2000]); Ans=ox_pop_cmo(Pid);
// HH may be HH=number_eval(exp(0)/10^3) to be a bigfloat.
Ans_th=ox_pop_cmo(Pid)$ // Matrix factorial is stored in Ans
// last t and h in the bignum format is stored in Ans_th
printf("[t_last,h]=%a\n",Ans_th)$
Ans_2000=eval_str(Ans);
printf("[AiryAi,AiryAi'] at t_last = %a\n",matrix_list_to_matrix(Ans_2000)*Iv)$
printf("N[AiryAi[2.00000000000000004163336342344 3],30] -> 0.034924130423274376924990811 3752\n")$
end$
```

See also: `matfac.rr` The script above in stored in `matfac.rr`

26 `code_solve_ode_by_rk4_with_defuse(Pmat,T0,F0,T1)`

Global variable `A2M_template`: the folder where template and library C programs `proj.c`, `proj.h`, `solve_ode_by_rd4.c`, `solve_ode_by_rd4.h` are put.

It returns a C program to solve the initial value program of the ODE

$$\frac{dF}{dt} = PF, \quad F(t_0) = F_0 \tag{4}$$

upto $t = t_1$ by a matrix factorial with the defusing heuristics.

`Pmat`	P
`T0`	t_0
`F0`	F_0
`T1`	t_1

Options are

Option		default value
`verbose`		0
`prec`	doube size of MPFR	64
`progname`		`tmp-test`
`h`	step size	0.001
`t_noproj`	time to apply defusing	0
`n_prune`	number of eigen vectors to prune	1
`strat`	projection strategy	1
`n_defuse`	number of the matrix factorial	$5 \cdot \lfloor 1/h \rfloor$
`ref_value_file`	File name of exact values	`tmp_ref_value.txt`

When `n_prune = 0` and `t_noproj = T0`, the behavior of the output program is that of the 4th order Runge-Kutta method.

The exact values in the file `ref_value_file` is used when the strategy 2 is taken. The format of this file is

$$t,\ r,\ f_1,\ f_2,\ f_3,\ \ldots\ ,f_r$$

where $[f_1, \ldots, f_r]$ is the first r part of the solution vector $F(t)$.

Example:

```
load("tk_ode\_by\_mpfr.rr")$
Code=code_solve_ode_by_rk4_with_defuse([[0,1],[t,0]],0,[0.355028053887817,-0.258819403792807],10.1)$
util_write_string_to_a_file("tmp-test.c",Code)$

ln -s ${OpenXM_HOME}/lib/asir-contrib/tk_any2mpfr/proj.c tmp-proj.c
cc -I${OpenXM_HOME}/lib/asir-contrib/tk_any2mpfr -DNN=2 -c tmp-proj.c
cc -o tmp-test tmp-test.c proj.o -lmpfr -lgmp  -lgsl -lgslcblas  -lm
```

Changelog: 2020.09.07. Version 1.