

Floquet Toolkit

Generated by Doxygen 1.8.17

1 FloquetToolkit	1
1.1 How to Install	1
1.2 How to Run	1
2 File Index	3
2.1 File List	3
3 File Documentation	5
3.1 include/diffEqSolvers.h File Reference	5
3.1.1 Function Documentation	6
3.1.1.1 bulsto_final_matrix_floquet_type_real()	6
3.1.1.2 rk4_adaptive_final_matrix_floquet_type_real()	7
3.1.1.3 rk4_fixed_final_matrix_floquet_type_complex()	9
3.1.1.4 rk4_fixed_final_matrix_floquet_type_real()	12
3.1.1.5 rk4_fixed_final_vector_real()	13
3.2 diffEqSolvers.h	13
3.3 include/floquet.h File Reference	14
3.3.1 Function Documentation	15
3.3.1.1 floquet_get_stability_array_real_double_param_general()	15
3.3.1.2 floquet_get_stability_array_real_single_param_general()	16
3.3.1.3 floquet_get_stability_reals_general()	17
3.4 floquet.h	20
3.5 src/diffEqSolvers.c File Reference	20
3.5.1 Function Documentation	21
3.5.1.1 bulsto_final_matrix_floquet_type_real()	21
3.5.1.2 rk4_adaptive_final_matrix_floquet_type_real()	22
3.5.1.3 rk4_fixed_final_matrix_floquet_type_complex()	24
3.5.1.4 rk4_fixed_final_matrix_floquet_type_real()	26
3.5.1.5 rk4_fixed_final_vector_real()	27
3.6 diffEqSolvers.c	28
3.7 src/floquet.c File Reference	33
3.7.1 Function Documentation	34
3.7.1.1 floquet_get_stability_array_real_double_param_general()	34
3.7.1.2 floquet_get_stability_array_real_single_param_general()	35
3.7.1.3 floquet_get_stability_reals_general()	36
3.8 floquet.c	39
3.9 src/hill_meissner.c File Reference	41
3.10 hill_meissner.c	41
3.11 src/hill_meissner_damped1.c File Reference	42
3.12 hill_meissner_damped1.c	43
3.13 src/hill_meissner_damped2.c File Reference	44
3.14 hill_meissner_damped2.c	44
3.15 src/mathieu_damped_k_0_1.c File Reference	45

3.16 mathieu_damped_k_0_1.c	46
3.17 src/mathieu_damped_k_0_5.c File Reference	47
3.18 mathieu_damped_k_0_5.c	48
3.19 src/mathieu_damped_k_1.c File Reference	49
3.20 mathieu_damped_k_1.c	49
3.21 src/mathieu_damped_k_10.c File Reference	50
3.22 mathieu_damped_k_10.c	51
3.23 src/mathieu_undamped.c File Reference	52
3.24 mathieu_undamped.c	53
3.25 src/population_dynamics.c File Reference	54
3.26 population_dynamics.c	54
Index	57

Chapter 1

FloquetToolkit

This is an attempt to create a C library to perform Floquet Analysis on a general system.

1.1 How to Install

To compile this, you would require GSL. In Debian-based distributions, it can be obtained by using

```
sudo apt install libgsl-dev
```

Just go to src and then run

```
make all
```

to compile all the programs. To compile just for specific cases, use

```
make mathieu
```

to get the programs corresponding to mathieu equation.

```
make meissner
```

to get the programs corresponding to hill-meissner equation.

```
make population_dynamics
```

to get the programs corresponding to the results on population dynamics.

1.2 How to Run

To run this, just go to the respective directories and run the compiled binaries. They will generate some data files.

To plot them and get the plots, run

```
python3 plotter3d.py
```

```
python3 plotter2d.py
```

Depending on the type of file that is available. plotter3d.py generated plots when 2 parameters are involved and plotter2d generates plots for when only 1 parameter is involved.

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

hill_meissner/ plotter_3d.py	??
hill_meissner_damped1/ plotter_3d.py	??
hill_meissner_damped2/ plotter_3d.py	??
include/ diffEqSolvers.h	5
include/ floquet.h	14
mathieu_damped_k_0_1/ plotter_3d.py	??
mathieu_damped_k_0_5/ plotter_3d.py	??
mathieu_damped_k_1/ plotter_3d.py	??
mathieu_damped_k_10/ plotter_3d.py	??
mathieu_undamped/ plotter_3d.py	??
population_dynamics_1/ plotter_2d.py	??
src/ diffEqSolvers.c	20
src/ floquet.c	33
src/ hill_meissner.c	41
src/ hill_meissner_damped1.c	42
src/ hill_meissner_damped2.c	44
src/ mathieu_damped_k_0_1.c	45
src/ mathieu_damped_k_0_5.c	47
src/ mathieu_damped_k_1.c	49
src/ mathieu_damped_k_10.c	50
src/ mathieu_undamped.c	52
src/ plotter_2d.py	??
src/ plotter_3d.py	??
src/ population_dynamics.c	54

Chapter 3

File Documentation

3.1 include/diffEqSolvers.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_cblas.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_complex.h>
#include <gsl/gsl_complex_math.h>
```

Macros

- `#define GSL_RANGE_CHECK_OFF`
- `#define HAVE_INLINE`
- `#define RK4_MAX_SCALE 5`
- `#define RK4_MIN_SCALE 0.2`
- `#define RK4_MAX_SLICES 1e8`
- `#define BULSTO_STEP_MAX 16`
Maximum number of midpoint method evaluations in one instance of Bulirsch-Stoer Method.
- `#define BULSTO_MAX_LAYERS 32`
Maximum depth of halving of Bulirsch-Stoer Method (this means that $H_{min} = 2^{-32} H$)

Functions

- void `rk4_fixed_final_vector_real` (int ndim, double *x_i, double t_i, double H, double h, void(*evol)↵
func)(double *, double, double *, void *), double *x_f, void *params)
Fixed Step RK4 for vectors for general $\dot{x} = f_{\lambda}(x, t)$.
- void `rk4_fixed_final_matrix_floquet_type_real` (gsl_matrix *x_i, double t_i, double H, double h, void(*A)(double,
gsl_matrix *, void *), gsl_matrix *x_f, void *params)
Fixed Step RK4 for real matrices for evolution of the form $\dot{X} = A_{\lambda}(t)X$.
- void `rk4_fixed_final_matrix_floquet_type_complex` (gsl_matrix_complex *x_i, double t_i, double H, double h,
void(*A)(double, gsl_matrix_complex *, void *), gsl_matrix_complex *x_f, void *params)

Fixed Step RK4 for complex matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

- void [rk4_adaptive_final_matrix_floquet_type_real](#) (gsl_matrix *x_i, double t_i, double H, double delta, void(*A)(double, gsl_matrix *, void *), gsl_matrix *x_f, void *params)

Adaptive Step RK4 for real matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

- void [bulsto_final_matrix_floquet_type_real](#) (gsl_matrix *x_i, double t_i, double H, double delta, void(*A)(double, gsl_matrix *, void *), gsl_matrix *x_f, void *params)

Bulirsch-Stoer Method for real matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

3.1.1 Function Documentation

3.1.1.1 bulsto_final_matrix_floquet_type_real()

```
void bulsto_final_matrix_floquet_type_real (
    gsl_matrix * x_i,
    double t_i,
    double H,
    double delta,
    void(*) (double, gsl_matrix *, void *) A,
    gsl_matrix * x_f,
    void * params )
```

Bulirsch-Stoer Method for real matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

<i>x_i</i>	Initial Matrix $x_i \in \mathbb{R}^{n \times m}$
<i>t_i</i>	Time when x_i is specified
<i>H</i>	Interval after which final x is required
<i>delta</i>	Maximum error allowed per unit time (The error is taken to be the maximum of the error of each element of the matrix)
<i>evol_func</i>	Function that computes $A_\lambda(t)$. The function should be of the form void A(double t, gsl_matrix* out, void* params)
<i>x_f</i>	Array to store the final x into. This should be preallocated
<i>params</i>	Parameters to be passed to A(t)

Definition at line 430 of file [diffEqSolvers.c](#).

```
00431 {
00432     // This program only initializes and provides and frees temp variables
00433     int ndim = x_i->size1;
00434     gsl_matrix** R1 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
00435     gsl_matrix** R2 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
00436
00437     for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
00438     {
00439         R1[i] = gsl_matrix_alloc(ndim, ndim);
00440         R2[i] = gsl_matrix_alloc(ndim, ndim);
00441     }
00442
00443     gsl_matrix* y = gsl_matrix_alloc(ndim, ndim);
00444     gsl_matrix* eval = gsl_matrix_alloc(ndim, ndim);
00445     gsl_matrix* epsilon = gsl_matrix_calloc(ndim, ndim);
00446 }
```

```

00447     __bulsto_final_matrix_floquet_type_real_runner(0, x_i, t_i, H, delta, A, x_f, params, y, eval, R1,
00448     R2, epsilon);
00448     //printf("%e %e %e\n",error, H, error/H);
00449     for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
00450     {
00451         gsl_matrix_free(R1[i]);
00452         gsl_matrix_free(R2[i]);
00453     }
00454     free(R1);
00455     free(R2);
00456     gsl_matrix_free(y);
00457     gsl_matrix_free(eval);
00458     gsl_matrix_free(epsilon);
00459 }

```

References [BULSTO_STEP_MAX](#).

Referenced by [floquet_get_stability_reals_general\(\)](#).

3.1.1.2 rk4_adaptive_final_matrix_floquet_type_real()

```

void rk4_adaptive_final_matrix_floquet_type_real (
    gsl_matrix * x_i,
    double t_i,
    double H,
    double delta,
    void(*) (double, gsl_matrix *, void *) A,
    gsl_matrix * x_f,
    void * params )

```

Adaptive Step RK4 for real matrices for evolution of the form $\dot{X} = A_{\lambda}(t)X$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

<i>x_i</i>	Initial Matrix $x_i \in \mathbb{R}^{n \times m}$
<i>t_i</i>	Time when <i>x_i</i> is specified
<i>H</i>	Interval after which final x is required
<i>delta</i>	Maximum error allowed per unit time (The error is taken to be the maximum of the error of each element of the matrix)
<i>evol_func</i>	Function that computes $A_{\lambda}(t)$. The function should be of the form void A(double t, gsl_matrix* out, void* params)
<i>x_f</i>	Array to store the final x into. This should be preallocated
<i>params</i>	Parameters to be passed to A(t)

Definition at line 165 of file [diffEqSolvers.c](#).

```

00166 {
00167     double h_min = H/RK4_MAX_SLICES;
00168     int nr = x_i->size1;
00169     int nc = x_i->size2;
00170     gsl_matrix* k[4];
00171     gsl_matrix* k_in[4];
00172     gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
00173     for (int i = 0; i < 4; ++i)
00174     {
00175         k[i] = gsl_matrix_calloc(nr,nc);
00176         k_in[i] = gsl_matrix_calloc(nr,nc);
00177     }
00178 }

```

```

00179     double t = t_i;
00180     double t_f = t_i + H;
00181     double h = H/10.; // Initial h. This is a bit conservative, but will be refined further.
00182     gsl_matrix_memcpy(x_f,x_i);
00183
00184     gsl_matrix* x1 = gsl_matrix_alloc(nr,nc);
00185     gsl_matrix* x2 = gsl_matrix_alloc(nr,nc);
00186
00187     while (t<t_f)
00188     {
00189         // Evaluate x1
00190         gsl_matrix_memcpy(x1,x_f);
00191
00192         A(t,A_val,params);
00193         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00194
00195         gsl_matrix_memcpy(k_in[0], k[0]);
00196         gsl_matrix_scale(k_in[0],0.5);
00197         gsl_matrix_add(k_in[0],x_f);
00198
00199         A(t+0.5*h,A_val,params);
00200         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00201
00202         gsl_matrix_memcpy(k_in[1], k[1]);
00203         gsl_matrix_scale(k_in[1],0.5);
00204         gsl_matrix_add(k_in[1],x_f);
00205
00206         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00207
00208         gsl_matrix_memcpy(k_in[2], k[2]);
00209         gsl_matrix_add(k_in[2],x_f);
00210
00211         A(t+h,A_val,params);
00212         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);
00213
00214         gsl_matrix_memcpy(k_in[3],k[3]);
00215         gsl_matrix_add(k_in[3],k[2]);
00216         gsl_matrix_add(k_in[3],k[2]);
00217         gsl_matrix_add(k_in[3],k[1]);
00218         gsl_matrix_add(k_in[3],k[1]);
00219         gsl_matrix_add(k_in[3],k[0]);
00220         gsl_matrix_scale(k_in[3], 1./6.);
00221         gsl_matrix_add(x1,k_in[3]);
00222
00223         t += h;
00224
00225         A(t,A_val,params);
00226         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00227
00228         gsl_matrix_memcpy(k_in[0], k[0]);
00229         gsl_matrix_scale(k_in[0],0.5);
00230         gsl_matrix_add(k_in[0],x1);
00231
00232         A(t+0.5*h,A_val,params);
00233         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00234
00235         gsl_matrix_memcpy(k_in[1], k[1]);
00236         gsl_matrix_scale(k_in[1],0.5);
00237         gsl_matrix_add(k_in[1],x1);
00238
00239         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00240
00241         gsl_matrix_memcpy(k_in[2], k[2]);
00242         gsl_matrix_add(k_in[2],x1);
00243
00244         A(t+h,A_val,params);
00245         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);
00246
00247         gsl_matrix_memcpy(k_in[3],k[3]);
00248         gsl_matrix_add(k_in[3],k[2]);
00249         gsl_matrix_add(k_in[3],k[2]);
00250         gsl_matrix_add(k_in[3],k[1]);
00251         gsl_matrix_add(k_in[3],k[1]);
00252         gsl_matrix_add(k_in[3],k[0]);
00253         gsl_matrix_scale(k_in[3], 1./6.);
00254         gsl_matrix_add(x1,k_in[3]);
00255
00256         t -= h;
00257
00258         // Evaluate x2
00259         gsl_matrix_memcpy(x2,x_f);
00260
00261         A(t,A_val,params);
00262         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, x_f, 0., k[0]);
00263
00264         gsl_matrix_memcpy(k_in[0], k[0]);
00265         gsl_matrix_scale(k_in[0],0.5);

```

```

00266     gsl_matrix_add(k_in[0], x_f);
00267
00268     A(t+h, A_val, params);
00269     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[0], 0., k[1]);
00270
00271     gsl_matrix_memcpy(k_in[1], k[1]);
00272     gsl_matrix_scale(k_in[1], 0.5);
00273     gsl_matrix_add(k_in[1], x_f);
00274
00275     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[1], 0., k[2]);
00276
00277     gsl_matrix_memcpy(k_in[2], k[2]);
00278     gsl_matrix_add(k_in[2], x_f);
00279
00280     A(t+2*h, A_val, params);
00281     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[2], 0., k[3]);
00282
00283     gsl_matrix_memcpy(k_in[3], k[3]);
00284     gsl_matrix_add(k_in[3], k[2]);
00285     gsl_matrix_add(k_in[3], k[2]);
00286     gsl_matrix_add(k_in[3], k[1]);
00287     gsl_matrix_add(k_in[3], k[1]);
00288     gsl_matrix_add(k_in[3], k[0]);
00289     gsl_matrix_scale(k_in[3], 1./6.);
00290     gsl_matrix_add(x2, k_in[3]);
00291
00292     // Evaluate Error
00293     gsl_matrix_sub(x2, x1);
00294     double rho_to_the_fourth = (30.*h*delta)/GSL_MAX(gsl_matrix_max(x2), -1.*gsl_matrix_min(x2));
00295     rho_to_the_fourth = pow(rho_to_the_fourth, 0.25);
00296     if (rho_to_the_fourth > 1)
00297     {
00298         gsl_matrix_memcpy(x_f, x1);
00299         gsl_matrix_scale(x2, -1./15.);
00300         gsl_matrix_add(x_f, x2);
00301         t += 2*h;
00302         h = h*(GSL_MIN(rho_to_the_fourth, RK4_MAX_SCALE));
00303         h = GSL_MIN(0.5*(t_f-t), h);
00304     }
00305     else if (h > h_min)
00306     {
00307         h = h*(GSL_MAX(rho_to_the_fourth, RK4_MIN_SCALE));
00308     }
00309     else
00310     {
00311         gsl_matrix_memcpy(x_f, x1);
00312         gsl_matrix_scale(x2, -1./15.);
00313         gsl_matrix_add(x_f, x2);
00314         t += 2*h;
00315         h = h_min;
00316     }
00317 }
00318
00319 gsl_matrix_free(A_val);
00320
00321 for (int i = 0; i < 4; ++i)
00322 {
00323     gsl_matrix_free(k[i]);
00324     gsl_matrix_free(k_in[i]);
00325 }
00326
00327 gsl_matrix_free(x1);
00328 gsl_matrix_free(x2);
00329 }

```

3.1.1.3 rk4_fixed_final_matrix_floquet_type_complex()

```

void rk4_fixed_final_matrix_floquet_type_complex (
    gsl_matrix_complex * x_i,
    double t_i,
    double H,
    double h,
    void(*) (double, gsl_matrix_complex *, void *) A,
    gsl_matrix_complex * x_f,
    void * params )

```

Fixed Step RK4 for complex matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

x_i	Initial Matrix $x_i \in \mathbb{C}^{n \times m}$
t_i	Time when x_i is specified
H	Interval after which final x is required
h	Step size
<i>evol_func</i>	Function that computes $A_\lambda(t)$. The function should be of the form void A(double t, gsl_matrix_complex* out, void* params)
x_f	Array to store the final x into. This should be preallocated
<i>params</i>	Parameters to be passed to A(t)

Definition at line 102 of file [diffEqSolvers.c](#).

```

00103 {
00104     int nr = x_i->size1;
00105     int nc = x_i->size2;
00106     gsl_matrix_complex* k[4];
00107     gsl_matrix_complex* k_in[4];
00108     gsl_matrix_complex* A_val = gsl_matrix_complex_alloc(nr,nc);
00109     for (int i = 0; i < 4; ++i)
00110     {
00111         k[i] = gsl_matrix_complex_alloc(nr,nc);
00112         k_in[i] = gsl_matrix_complex_alloc(nr,nc);
00113     }
00114
00115     gsl_complex h = gsl_complex_rect(h_,0.);
00116     gsl_complex zero = gsl_complex_rect(0.,0.);
00117     gsl_complex half = gsl_complex_rect(0.5,0.);
00118     gsl_complex one_sixth = gsl_complex_rect(1./6.,0.);
00119
00120     double t = t_i;
00121     double t_f = t_i + H;
00122     gsl_matrix_complex_memcpy(x_f,x_i);
00123     while (t<t_f)
00124     {
00125         A(t,A_val,params);
00126         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, zero, k[0]);
00127
00128         gsl_matrix_complex_memcpy(k_in[0], k[0]);
00129         gsl_matrix_complex_scale(k_in[0],half);
00130         gsl_matrix_complex_add(k_in[0],x_f);
00131
00132         A(t+0.5*h_,A_val,params);
00133         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], zero, k[1]);
00134
00135         gsl_matrix_complex_memcpy(k_in[1], k[1]);
00136         gsl_matrix_complex_scale(k_in[1],half);
00137         gsl_matrix_complex_add(k_in[1],x_f);
00138
00139         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], zero, k[2]);
00140
00141         gsl_matrix_complex_memcpy(k_in[2], k[2]);
00142         gsl_matrix_complex_add(k_in[2],x_f);
00143
00144         A(t+h_,A_val,params);
00145         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], zero, k[3]);
00146
00147         gsl_matrix_complex_memcpy(k_in[3],k[3]);
00148         gsl_matrix_complex_add(k_in[3],k[2]);
00149         gsl_matrix_complex_add(k_in[3],k[2]);
00150         gsl_matrix_complex_add(k_in[3],k[1]);
00151         gsl_matrix_complex_add(k_in[3],k[1]);
00152         gsl_matrix_complex_add(k_in[3],k[0]);
00153         gsl_matrix_complex_scale(k_in[3], one_sixth);
00154         gsl_matrix_complex_add(x_f,k_in[3]);
00155         t += h_;
00156     }
00157
00158     for (int i = 0; i < 4; ++i)
00159     {
00160         gsl_matrix_complex_free(k[i]);
00161         gsl_matrix_complex_free(k_in[i]);
00162     }
00163 }

```

3.1.1.4 rk4_fixed_final_matrix_floquet_type_real()

```
void rk4_fixed_final_matrix_floquet_type_real (
    gsl_matrix * x_i,
    double t_i,
    double H,
    double h,
    void(*) (double, gsl_matrix *, void *) A,
    gsl_matrix * x_f,
    void * params )
```

Fixed Step RK4 for real matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

x_i	Initial Matrix $x_i \in \mathbb{R}^{n \times m}$
t_i	Time when x_i is specified
H	Interval after which final x is required
h	Step size
<i>evol_func</i>	Function that computes $A_\lambda(t)$. The function should be of the form void A(double t, gsl_matrix* out, void* params)
x_f	Array to store the final x into. This should be preallocated
<i>params</i>	Parameters to be passed to A(t)

Definition at line 44 of file [diffEqSolvers.c](#).

```
00045 {
00046     int nr = x_i->size1;
00047     int nc = x_i->size2;
00048     gsl_matrix* k[4];
00049     gsl_matrix* k_in[4];
00050     gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
00051     for (int i = 0; i < 4; ++i)
00052     {
00053         k[i] = gsl_matrix_calloc(nr,nc);
00054         k_in[i] = gsl_matrix_calloc(nr,nc);
00055     }
00056
00057     double t = t_i;
00058     double t_f = t_i + H;
00059     gsl_matrix_memcpy(x_f, x_i);
00060     while (t < t_f)
00061     {
00062         A(t, A_val, params);
00063         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00064
00065         gsl_matrix_memcpy(k_in[0], k[0]);
00066         gsl_matrix_scale(k_in[0], 0.5);
00067         gsl_matrix_add(k_in[0], x_f);
00068
00069         A(t+0.5*h, A_val, params);
00070         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00071
00072         gsl_matrix_memcpy(k_in[1], k[1]);
00073         gsl_matrix_scale(k_in[1], 0.5);
00074         gsl_matrix_add(k_in[1], x_f);
00075
00076         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00077
00078         gsl_matrix_memcpy(k_in[2], k[2]);
00079         gsl_matrix_add(k_in[2], x_f);
00080
00081         A(t+h, A_val, params);
00082         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);
00083
00084         gsl_matrix_memcpy(k_in[3], k[3]);
00085         gsl_matrix_add(k_in[3], k[2]);
00086         gsl_matrix_add(k_in[3], k[2]);
```



```

00087     gsl_matrix_add(k_in[3],k[1]);
00088     gsl_matrix_add(k_in[3],k[1]);
00089     gsl_matrix_add(k_in[3],k[0]);
00090     gsl_matrix_scale(k_in[3], 1./6.);
00091     gsl_matrix_add(x_f,k_in[3]);
00092     t += h;
00093 }
00094
00095 for (int i = 0; i < 4; ++i)
00096 {
00097     gsl_matrix_free(k[i]);
00098     gsl_matrix_free(k_in[i]);
00099 }
00100 }

```

3.1.1.5 rk4_fixed_final_vector_real()

```

void rk4_fixed_final_vector_real (
    int ndim,
    double * x_i,
    double t_i,
    double H,
    double h,
    void(*) (double *, double, double *, void *) evol_func,
    double * x_f,
    void * params )

```

Fixed Step RK4 for vectors for general $\dot{x} = f_{\lambda}(x, t)$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

<i>ndim</i>	Dimensionality of x
<i>x_i</i>	Initial Vector $x_i \in \mathbb{R}^n$
<i>t_i</i>	Time when <i>x_i</i> is specified
<i>H</i>	Interval after which final x is required
<i>h</i>	Step size
<i>evol_func</i>	Function that computes $\frac{dx}{dt}(x, t)$. The function should be of the form void evol_func(double* x, double t, double* x_dot, void* params)
<i>x_f</i>	Array to store the final x into. This should be preallocated
<i>params</i>	Parameters to be passed to evol_func

Definition at line 31 of file [diffEqSolvers.c](#).

```

00032 {
00033     double t = t_i;
00034     double t_f = t_i + H;
00035
00036     cblas_dcopy (ndim,x_i,sizeof(x_i[0]),x_f,sizeof(x_f[0]));
00037     while(t<t_f)
00038     {
00039         __rk4_single_vector (ndim,x_f,t,h,evol_func,x_f, params);
00040         t += h;
00041     }
00042 }

```

3.2 diffEqSolvers.h

```
00001 #ifndef DIFFEQ_SOLVERS
```

```

00003 #define DIFFEQ_SOLVERS
00004
00005 #include <stdio.h>
00006 #include <stdlib.h>
00007 #include <math.h>
00008 #include <gsl/gsl_math.h>
00009 #include <gsl/gsl_cblas.h>
00010 #include <gsl/gsl_matrix.h>
00011 #include <gsl/gsl_blas.h>
00012 #include <gsl/gsl_complex.h>
00013 #include <gsl/gsl_complex_math.h>
00014
00015 #define GSL_RANGE_CHECK_OFF
00016 #define HAVE_INLINE
00017
00018 #define RK4_MAX_SCALE 5
00019 #define RK4_MIN_SCALE 0.2
00020 #define RK4_MAX_SLICES 1e8
00021 #define BULSTO_STEP_MAX 16
00022 #define BULSTO_MAX_LAYERS 32
00023
00024
00037 void rk4_fixed_final_vector_real(int ndim, double* x_i, double t_i, double H, double h, void
    (*evol_func)(double*, double, double*, void*), double* x_f, void* params);
00038
00050 void rk4_fixed_final_matrix_floquet_type_real(gsl_matrix* x_i, double t_i, double H, double h, void
    (*A)(double, gsl_matrix*, void*), gsl_matrix* x_f, void* params);
00051
00063 void rk4_fixed_final_matrix_floquet_type_complex(gsl_matrix_complex* x_i, double t_i, double H, double
    h, void (*A)(double, gsl_matrix_complex*, void*), gsl_matrix_complex* x_f, void* params);
00064
00076 void rk4_adaptive_final_matrix_floquet_type_real(gsl_matrix* x_i, double t_i, double H, double delta,
    void (*A)(double, gsl_matrix*, void*), gsl_matrix* x_f, void* params);
00077
00089 void bulsto_final_matrix_floquet_type_real(gsl_matrix* x_i, double t_i, double H, double delta, void
    (*A)(double, gsl_matrix*, void*), gsl_matrix* x_f, void* params);
00090
00091 #endif

```

3.3 include/floquet.h File Reference

```

#include "diffEqSolvers.h"
#include <gsl/gsl_eigen.h>
#include <omp.h>

```

Macros

- `#define ERR_TOL 1e-6`
Maximum total error to which the differential equation would be solved.
- `#define ERR_EIGEN_TOL 1e-5`
Maximum value of $||\rho_{max}| - 1|$ to differentiate between stable and unstable system.

Functions

- int [floquet_get_stability_reals_general](#) (int n, void(*A)(double, gsl_matrix *, void *), void *params, double T, gsl_complex *largest_multiplier, double *largest_multiplier_abs)
Function which checks if the function is Floquet Stable or unstable for a general function.
- void [floquet_get_stability_array_real_single_param_general](#) (int n, void(*A)(double, gsl_matrix *, void *), double T, double start, double end, int nstep, int *stability, gsl_complex *largest_multiplier, double *largest_multiplier_abs)
CPU Parallelized Function which iterates over a range of a parameter and checks if the function is Floquet Stable or unstable for a general function.
- void [floquet_get_stability_array_real_double_param_general](#) (int n, void(*A)(double, gsl_matrix *, void *), double T, double *start, double *end, int *nstep, int **stability, gsl_complex **largest_multiplier, double **largest_multiplier_abs)
CPU Parallelized Function which iterates over the ranges of 2 parameters and checks if the function is Floquet Stable or unstable for a general function.

3.3.1 Function Documentation

3.3.1.1 floquet_get_stability_array_real_double_param_general()

```
void floquet_get_stability_array_real_double_param_general (
    int n,
    void(*) (double, gsl_matrix *, void *) A,
    double T,
    double * start,
    double * end,
    int * nstep,
    int ** stability,
    gsl_complex ** largest_multiplier,
    double ** largest_multiplier_abs )
```

CPU Parallelized Function which iterates over the ranges of 2 parameters and checks if the function is Floquet Stable or unstable for a general function.

Run `floquet_get_stability_reals_general` on the ranges of 2 parameters and stores the stability, floquet multiplier corresponding to the largest absolute value, and the absolute value of the largest floquet multiplier. This is a memory naive implementation.

Parameters

<i>n</i>	Number of elements of vector that $A(t)$ operates on
<i>A</i>	$A(t)$ matrix corresponding to the equation $\dot{x} = Ax$. The void* should be resolved to a double inside the function because a double* with 2 doubles would be passed.
<i>T</i>	Period of the evolution function $A(t)$ s.t. $A(t + T) = A(t) \forall t \in \mathbb{R}$
<i>start</i>	Starting values of the parameters
<i>end</i>	Ending values of the parameters
<i>nstep</i>	Number of steps to take inclusive of the first and last values keeping the other parameter constant. Should be at least 2. Behaviour not defined otherwise.
<i>largest_multiplier</i>	Matrix to store the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.
<i>largest_multiplier_abs</i>	Matrix to store the absolute value of the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.

Definition at line 110 of file `floquet.c`.

```
00111 {
00112     gsl_complex** mult_temp;
00113     double** mult_abs_temp;
00114
00115     if (largest_multiplier)
00116     {
00117         mult_temp = largest_multiplier;
00118     }
00119     else
00120     {
00121         mult_temp = (gsl_complex**) malloc(nstep[0]*sizeof(gsl_complex*));
00122         for (int i = 0; i < nstep[0]; ++i)
00123         {
00124             mult_temp[i] = (gsl_complex*) malloc(nstep[1]*sizeof(gsl_complex));
00125         }
00126     }
00127
00128     if (largest_multiplier_abs)
00129     {
```

```

00130         mult_abs_temp = largest_multiplier_abs;
00131     }
00132     else
00133     {
00134         mult_abs_temp = (double**) malloc(nstep[0]*sizeof(double*));
00135         for (int i = 0; i < nstep[0]; ++i)
00136         {
00137             mult_abs_temp[i] = (double*) malloc(nstep[1]*sizeof(double));
00138         }
00139     }
00140
00141     double step[2] = {(end[0]-start[0])/(nstep[0]-1), (end[1]-start[1])/(nstep[1]-1)};
00142     #pragma omp parallel for collapse(2) schedule(guided)
00143     for (int i = 0; i < nstep[0]; ++i)
00144     {
00145         for (int j = 0; j < nstep[1]; ++j)
00146         {
00147             double param[2] = {start[0] + step[0]*i, start[1] + step[1]*j };
00148             stability[i][j] =
00149             floquet_get_stability_reals_general(n,A,param,T,&mult_temp[i][j],&mult_abs_temp[i][j]);
00150         }
00151     }
00152     if(!largest_multiplier)
00153     {
00154         for (int i = 0; i < nstep[0]; ++i)
00155         {
00156             free(mult_temp[i]);
00157         }
00158         free(mult_temp);
00159     }
00160     if(!largest_multiplier_abs)
00161     {
00162         for (int i = 0; i < nstep[0]; ++i)
00163         {
00164             free(mult_abs_temp[i]);
00165         }
00166         free(mult_abs_temp);
00167     }
00168 }
00169 }

```

References [floquet_get_stability_reals_general\(\)](#).

3.3.1.2 floquet_get_stability_array_real_single_param_general()

```

void floquet_get_stability_array_real_single_param_general (
    int n,
    void(*) (double, gsl_matrix *, void *) A,
    double T,
    double start,
    double end,
    int nstep,
    int * stability,
    gsl_complex * largest_multiplier,
    double * largest_multiplier_abs )

```

CPU Parallelized Function which iterates over a range of a parameter and checks if the function is Floquet Stable or unstable for a general function.

Run `floquet_get_stability_reals_general` on a range of a parameter and stores the stability, floquet multiplier corresponding to the largest absolute value, and the absolute value of the largest floquet multiplier. This is a memory naive implementation.

Parameters

<i>n</i>	Number of elements of vector that $A(t)$ operates on
----------	--

Parameters

<i>A</i>	$A(t)$ matrix corresponding to the equation $\dot{x} = Ax$. The void* should be resolved to a double inside the function because a double* with a single double would be passed.
<i>T</i>	Period of the evolution function $A(t)$ s.t. $A(t + T) = A(t) \forall t \in \mathbb{R}$
<i>start</i>	Starting value of the parameter
<i>end</i>	Ending value of the parameter
<i>nstep</i>	Number of steps to take inclusive of the first and last values. Should be at least 2. Behaviour not defined otherwise.
<i>largest_multiplier</i>	Array to store the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.
<i>largest_multiplier_abs</i>	Array to store the absolute value of the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.

Definition at line 69 of file [floquet.c](#).

```

00070 {
00071     gsl_complex* mult_temp;
00072     double* mult_abs_temp;
00073
00074     if (largest_multiplier)
00075     {
00076         mult_temp = largest_multiplier;
00077     }
00078     else
00079     {
00080         mult_temp = (gsl_complex*) malloc(nstep*sizeof(gsl_complex));
00081     }
00082
00083     if (largest_multiplier_abs)
00084     {
00085         mult_abs_temp = largest_multiplier_abs;
00086     }
00087     else
00088     {
00089         mult_abs_temp = (double*) malloc(nstep*sizeof(double));
00090     }
00091
00092     double step = (end-start)/(nstep-1);
00093     #pragma omp parallel for
00094     for (int i = 0; i < nstep; ++i)
00095     {
00096         double param = start + step*i;
00097         stability[i] = floquet_get_stability_reals_general(n,A,&param,T,mult_temp+i,mult_abs_temp+i);
00098     }
00099
00100     if(!largest_multiplier)
00101     {
00102         free(mult_temp);
00103     }
00104     if(!largest_multiplier_abs)
00105     {
00106         free(mult_abs_temp);
00107     }
00108 }

```

References [floquet_get_stability_reals_general\(\)](#).

3.3.1.3 floquet_get_stability_reals_general()

```

int floquet_get_stability_reals_general (
    int n,
    void(*) (double, gsl_matrix *, void *) A,
    void * params,
    double T,

```

```
gsl_complex * largest_multiplier,  
double * largest_multiplier_abs )
```

Function which checks if the function is Floquet Stable or unstable for a general function.

Naive implementation with the elements of the $X(T)$ matrix calculated to precision ERR_TOL

Parameters

<i>n</i>	Number of elements of vector that $A(t)$ operates on
<i>A</i>	$A(t)$ matrix corresponding to the equation $\dot{x} = Ax$
<i>params</i>	Parameters to be passed to $A(t)$
<i>T</i>	Period of the evolution function $A(t)$ s.t. $A(t + T) = A(t) \forall t \in \mathbb{R}$
<i>largest_multiplier</i>	Pointer to store the largest (by abs) computed Floquet multiplier into. No multiplier will be stored if NULL is passed.
<i>largest_multiplier_abs</i>	Pointer to store the absolute value of the largest (by abs) computed Floquet multiplier into. No multiplier will be stored if NULL is passed.

Returns

1 if Stable, -1 if unstable, 0 if periodic or indeterminate to accuracy ERR_EIGEN_TOL. In rare cases, 2 would be returned if none of the computed floquet multipliers lead to instability, but not all of multipliers could be computed.

Definition at line 4 of file [floquet.c](#).

```

00005 {
00006     gsl_matrix* X = gsl_matrix_alloc(n,n);
00007     gsl_matrix_set_identity(X);
00008
00009     gsl_matrix* B = gsl_matrix_alloc(n,n);
00010     bulsto_final_matrix_floquet_type_real(X, 0., T, ERR_TOL, A, B, params);
00011
00012     gsl_vector_complex* eigenvals = gsl_vector_complex_alloc(n);
00013
00014     gsl_eigen_nonsymm_workspace* w = gsl_eigen_nonsymm_alloc(n);
00015
00016     int n_eigenvals_evaluated = n;
00017
00018     int err_code = gsl_eigen_nonsymm(B,eigenvals,w);
00019     if(err_code)
00020     {
00021         n_eigenvals_evaluated = w->n_evals;
00022     }
00023
00024     double mult_max_abs = -HUGE_VAL;
00025     gsl_complex mult_max;
00026     double ev_test;
00027     for (int i = 0; i < n_eigenvals_evaluated; ++i)
00028     {
00029         ev_test = gsl_complex_logabs(gsl_vector_complex_get(eigenvals,i));
00030         if (mult_max_abs < ev_test)
00031         {
00032             mult_max_abs = ev_test;
00033             mult_max = gsl_vector_complex_get(eigenvals,i);
00034         }
00035     }
00036
00037     if (largest_multiplier != NULL)
00038     {
00039         *largest_multiplier = mult_max;
00040     }
00041     if (largest_multiplier_abs != NULL)
00042     {
00043         *largest_multiplier_abs = gsl_complex_abs(mult_max);
00044     }
00045
00046     gsl_eigen_nonsymm_free(w);
00047     gsl_vector_complex_free(eigenvals);
00048     gsl_matrix_free(B);
00049     gsl_matrix_free(X);
00050
00051     if (mult_max_abs > ERR_EIGEN_TOL)
00052     {
00053         return 1;
00054     }
00055     else if (mult_max_abs < (-ERR_EIGEN_TOL))
00056     {
00057         if (n_eigenvals_evaluated < n)
00058         {
00059             return 2;

```

```

00060     }
00061     return -1;
00062 }
00063 else
00064 {
00065     return 0;
00066 }
00067 }

```

References [bulsto_final_matrix_floquet_type_real\(\)](#), [ERR_EIGEN_TOL](#), and [ERR_TOL](#).

Referenced by [floquet_get_stability_array_real_double_param_general\(\)](#), and [floquet_get_stability_array_real_single_param_general\(\)](#).

3.4 floquet.h

```

00001 #ifndef FLOQUET
00002 #define FLOQUET
00003
00004
00005 #define ERR_TOL 1e-6
00006 #define ERR_EIGEN_TOL 1e-5
00007
00008 #include "diffEqSolvers.h"
00009 #include <gsl/gsl_eigen.h>
00010 #include <omp.h>
00011
00023 int floquet_get_stability_reals_general(int n, void (*A)(double, gsl_matrix*, void*), void* params,
    double T, gsl_complex* largest_multiplier, double* largest_multiplier_abs);
00024
00038 void floquet_get_stability_array_real_single_param_general(int n, void (*A)(double, gsl_matrix*,
    void*), double T, double start, double end, int nstep, int* stability, gsl_complex*
    largest_multiplier, double* largest_multiplier_abs);
00039
00053 void floquet_get_stability_array_real_double_param_general(int n, void (*A)(double, gsl_matrix*,
    void*), double T, double* start, double* end, int* nstep, int** stability, gsl_complex**
    largest_multiplier, double** largest_multiplier_abs);
00054
00055 #endif

```

3.5 src/diffEqSolvers.c File Reference

```
#include "diffEqSolvers.h"
```

Functions

- void [__rk4_single_vector](#) (int n, double *x, double t, double h, void(*evol_func)(double *, double, double *, void *), double *x_f, void *params)
- void [rk4_fixed_final_vector_real](#) (int ndim, double *x_i, double t_i, double H, double h, void(*evol_func)(double *, double, double *, void *), double *x_f, void *params)
Fixed Step RK4 for vectors for general $\dot{x} = f_{\lambda}(x, t)$.
- void [rk4_fixed_final_matrix_floquet_type_real](#) (gsl_matrix *x_i, double t_i, double H, double h, void(*A)(double, gsl_matrix *, void *), gsl_matrix *x_f, void *params)
Fixed Step RK4 for real matrices for evolution of the form $\dot{X} = A_{\lambda}(t)X$.
- void [rk4_fixed_final_matrix_floquet_type_complex](#) (gsl_matrix_complex *x_i, double t_i, double H, double h, void(*A)(double, gsl_matrix_complex *, void *), gsl_matrix_complex *x_f, void *params)
Fixed Step RK4 for complex matrices for evolution of the form $\dot{X} = A_{\lambda}(t)X$.
- void [rk4_adaptive_final_matrix_floquet_type_real](#) (gsl_matrix *x_i, double t_i, double H, double delta, void(*A)(double, gsl_matrix *, void *), gsl_matrix *x_f, void *params)
Adaptive Step RK4 for real matrices for evolution of the form $\dot{X} = A_{\lambda}(t)X$.
- void [__midpoint_method](#) (gsl_matrix *x, double t_i, double H, void(*A)(double, gsl_matrix *, void *), gsl_matrix *x_f, void *params, gsl_matrix *y, gsl_matrix *eval)

- double **__bulsto_final_matrix_floquet_type_real_main** (gsl_matrix *x_i, double t_i, double H, double delta, void(*A)(double, gsl_matrix *, void *), gsl_matrix *x_f, void *params, gsl_matrix *y, gsl_matrix *eval, gsl_matrix **R1, gsl_matrix **R2, gsl_matrix *epsilon)
- double **__bulsto_final_matrix_floquet_type_real_runner** (int nlayer, gsl_matrix *x_i, double t_i, double H, double delta, void(*A)(double, gsl_matrix *, void *), gsl_matrix *x_f, void *params, gsl_matrix *y, gsl_matrix *eval, gsl_matrix **R1, gsl_matrix **R2, gsl_matrix *epsilon)
- void **bulsto_final_matrix_floquet_type_real** (gsl_matrix *x_i, double t_i, double H, double delta, void(*A)(double, gsl_matrix *, void *), gsl_matrix *x_f, void *params)

Bulirsch-Stoer Method for real matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

3.5.1 Function Documentation

3.5.1.1 bulsto_final_matrix_floquet_type_real()

```
void bulsto_final_matrix_floquet_type_real (
    gsl_matrix * x_i,
    double t_i,
    double H,
    double delta,
    void(*) (double, gsl_matrix *, void *) A,
    gsl_matrix * x_f,
    void * params )
```

Bulirsch-Stoer Method for real matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

x_i	Initial Matrix $x_i \in \mathbb{R}^{n \times m}$
t_i	Time when x_i is specified
H	Interval after which final x is required
δ	Maximum error allowed per unit time (The error is taken to be the maximum of the error of each element of the matrix)
$evol_func$	Function that computes $A_\lambda(t)$. The function should be of the form void A(double t, gsl_matrix* out, void* params)
x_f	Array to store the final x into. This should be preallocated
$params$	Parameters to be passed to A(t)

Definition at line 430 of file diffEqSolvers.c.

```
00431 {
00432     // This program only initializes and provides and frees temp variables
00433     int ndim = x_i->size1;
00434     gsl_matrix** R1 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
00435     gsl_matrix** R2 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
00436
00437     for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
00438     {
00439         R1[i] = gsl_matrix_alloc(ndim, ndim);
00440         R2[i] = gsl_matrix_alloc(ndim, ndim);
00441     }
00442
00443     gsl_matrix* y = gsl_matrix_alloc(ndim, ndim);
00444     gsl_matrix* eval = gsl_matrix_alloc(ndim, ndim);
```

```

00445     gsl_matrix* epsilon = gsl_matrix_calloc(ndim,ndim);
00446
00447     __bulsto_final_matrix_floquet_type_real_runner(0, x_i, t_i, H, delta, A, x_f, params, y, eval, R1,
R2, epsilon);
00448     //printf("%e %e %e\n",error, H, error/H);
00449     for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
00450     {
00451         gsl_matrix_free(R1[i]);
00452         gsl_matrix_free(R2[i]);
00453     }
00454     free(R1);
00455     free(R2);
00456     gsl_matrix_free(y);
00457     gsl_matrix_free(eval);
00458     gsl_matrix_free(epsilon);
00459 }

```

References [BULSTO_STEP_MAX](#).

Referenced by [floquet_get_stability_reals_general\(\)](#).

3.5.1.2 rk4_adaptive_final_matrix_floquet_type_real()

```

void rk4_adaptive_final_matrix_floquet_type_real (
    gsl_matrix * x_i,
    double t_i,
    double H,
    double delta,
    void(*) (double, gsl_matrix *, void *) A,
    gsl_matrix * x_f,
    void * params )

```

Adaptive Step RK4 for real matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

x_i	Initial Matrix $x_i \in \mathbb{R}^{n \times m}$
t_i	Time when x_i is specified
H	Interval after which final x is required
δ	Maximum error allowed per unit time (The error is taken to be the maximum of the error of each element of the matrix)
$evol_func$	Function that computes $A_\lambda(t)$. The function should be of the form void A(double t, gsl_matrix* out, void* params)
x_f	Array to store the final x into. This should be preallocated
$params$	Parameters to be passed to A(t)

Definition at line 165 of file [diffEqSolvers.c](#).

```

00166 {
00167     double h_min = H/RK4_MAX_SLICES;
00168     int nr = x_i->size1;
00169     int nc = x_i->size2;
00170     gsl_matrix* k[4];
00171     gsl_matrix* k_in[4];
00172     gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
00173     for (int i = 0; i < 4; ++i)
00174     {
00175         k[i] = gsl_matrix_calloc(nr,nc);
00176         k_in[i] = gsl_matrix_calloc(nr,nc);

```

```

00177     }
00178
00179     double t = t_i;
00180     double t_f = t_i + H;
00181     double h = H/10.; // Initial h. This is a bit conservative, but will be refined further.
00182     gsl_matrix_memcpy(x_f,x_i);
00183
00184     gsl_matrix* x1 = gsl_matrix_alloc(nr,nc);
00185     gsl_matrix* x2 = gsl_matrix_alloc(nr,nc);
00186
00187     while (t<t_f)
00188     {
00189         // Evaluate x1
00190         gsl_matrix_memcpy(x1,x_f);
00191
00192         A(t,A_val,params);
00193         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00194
00195         gsl_matrix_memcpy(k_in[0], k[0]);
00196         gsl_matrix_scale(k_in[0],0.5);
00197         gsl_matrix_add(k_in[0],x_f);
00198
00199         A(t+0.5*h,A_val,params);
00200         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00201
00202         gsl_matrix_memcpy(k_in[1], k[1]);
00203         gsl_matrix_scale(k_in[1],0.5);
00204         gsl_matrix_add(k_in[1],x_f);
00205
00206         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00207
00208         gsl_matrix_memcpy(k_in[2], k[2]);
00209         gsl_matrix_add(k_in[2],x_f);
00210
00211         A(t+h,A_val,params);
00212         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);
00213
00214         gsl_matrix_memcpy(k_in[3],k[3]);
00215         gsl_matrix_add(k_in[3],k[2]);
00216         gsl_matrix_add(k_in[3],k[2]);
00217         gsl_matrix_add(k_in[3],k[1]);
00218         gsl_matrix_add(k_in[3],k[1]);
00219         gsl_matrix_add(k_in[3],k[0]);
00220         gsl_matrix_scale(k_in[3], 1./6.);
00221         gsl_matrix_add(x1,k_in[3]);
00222
00223         t += h;
00224
00225         A(t,A_val,params);
00226         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00227
00228         gsl_matrix_memcpy(k_in[0], k[0]);
00229         gsl_matrix_scale(k_in[0],0.5);
00230         gsl_matrix_add(k_in[0],x1);
00231
00232         A(t+0.5*h,A_val,params);
00233         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00234
00235         gsl_matrix_memcpy(k_in[1], k[1]);
00236         gsl_matrix_scale(k_in[1],0.5);
00237         gsl_matrix_add(k_in[1],x1);
00238
00239         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00240
00241         gsl_matrix_memcpy(k_in[2], k[2]);
00242         gsl_matrix_add(k_in[2],x1);
00243
00244         A(t+h,A_val,params);
00245         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);
00246
00247         gsl_matrix_memcpy(k_in[3],k[3]);
00248         gsl_matrix_add(k_in[3],k[2]);
00249         gsl_matrix_add(k_in[3],k[2]);
00250         gsl_matrix_add(k_in[3],k[1]);
00251         gsl_matrix_add(k_in[3],k[1]);
00252         gsl_matrix_add(k_in[3],k[0]);
00253         gsl_matrix_scale(k_in[3], 1./6.);
00254         gsl_matrix_add(x1,k_in[3]);
00255
00256         t -= h;
00257
00258         // Evaluate x2
00259         gsl_matrix_memcpy(x2,x_f);
00260
00261         A(t,A_val,params);
00262         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, x_f, 0., k[0]);
00263

```

```

00264     gsl_matrix_memcpy(k_in[0], k[0]);
00265     gsl_matrix_scale(k_in[0], 0.5);
00266     gsl_matrix_add(k_in[0], x_f);
00267
00268     A(t+h, A_val, params);
00269     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[0], 0., k[1]);
00270
00271     gsl_matrix_memcpy(k_in[1], k[1]);
00272     gsl_matrix_scale(k_in[1], 0.5);
00273     gsl_matrix_add(k_in[1], x_f);
00274
00275     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[1], 0., k[2]);
00276
00277     gsl_matrix_memcpy(k_in[2], k[2]);
00278     gsl_matrix_add(k_in[2], x_f);
00279
00280     A(t+2*h, A_val, params);
00281     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[2], 0., k[3]);
00282
00283     gsl_matrix_memcpy(k_in[3], k[3]);
00284     gsl_matrix_add(k_in[3], k[2]);
00285     gsl_matrix_add(k_in[3], k[2]);
00286     gsl_matrix_add(k_in[3], k[1]);
00287     gsl_matrix_add(k_in[3], k[1]);
00288     gsl_matrix_add(k_in[3], k[0]);
00289     gsl_matrix_scale(k_in[3], 1./6.);
00290     gsl_matrix_add(x2, k_in[3]);
00291
00292     // Evaluate Error
00293     gsl_matrix_sub(x2, x1);
00294     double rho_to_the_fourth = (30.*h*delta)/GSL_MAX(gsl_matrix_max(x2), -1.*gsl_matrix_min(x2));
00295     rho_to_the_fourth = pow(rho_to_the_fourth, 0.25);
00296     if (rho_to_the_fourth > 1)
00297     {
00298         gsl_matrix_memcpy(x_f, x1);
00299         gsl_matrix_scale(x2, -1./15.);
00300         gsl_matrix_add(x_f, x2);
00301         t += 2*h;
00302         h = h*(GSL_MIN(rho_to_the_fourth, RK4_MAX_SCALE));
00303         h = GSL_MIN(0.5*(t_f-t), h);
00304     }
00305     else if (h > h_min)
00306     {
00307         h = h*(GSL_MAX(rho_to_the_fourth, RK4_MIN_SCALE));
00308     }
00309     else
00310     {
00311         gsl_matrix_memcpy(x_f, x1);
00312         gsl_matrix_scale(x2, -1./15.);
00313         gsl_matrix_add(x_f, x2);
00314         t += 2*h;
00315         h = h_min;
00316     }
00317 }
00318
00319 gsl_matrix_free(A_val);
00320
00321 for (int i = 0; i < 4; ++i)
00322 {
00323     gsl_matrix_free(k[i]);
00324     gsl_matrix_free(k_in[i]);
00325 }
00326
00327 gsl_matrix_free(x1);
00328 gsl_matrix_free(x2);
00329 }

```

3.5.1.3 rk4_fixed_final_matrix_floquet_type_complex()

```

void rk4_fixed_final_matrix_floquet_type_complex (
    gsl_matrix_complex * x_i,
    double t_i,
    double H,
    double h,
    void(*) (double, gsl_matrix_complex *, void *) A,
    gsl_matrix_complex * x_f,
    void * params )

```

Fixed Step RK4 for complex matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

x_i	Initial Matrix $x_i \in \mathbb{C}^{n \times m}$
t_i	Time when x_i is specified
H	Interval after which final x is required
h	Step size
<i>evol_func</i>	Function that computes $A_\lambda(t)$. The function should be of the form void A(double t, gsl_matrix_complex* out, void* params)
x_f	Array to store the final x into. This should be preallocated
<i>params</i>	Parameters to be passed to A(t)

Definition at line 102 of file diffEqSolvers.c.

```

00103 {
00104     int nr = x_i->size1;
00105     int nc = x_i->size2;
00106     gsl_matrix_complex* k[4];
00107     gsl_matrix_complex* k_in[4];
00108     gsl_matrix_complex* A_val = gsl_matrix_complex_alloc(nr,nc);
00109     for (int i = 0; i < 4; ++i)
00110     {
00111         k[i] = gsl_matrix_complex_alloc(nr,nc);
00112         k_in[i] = gsl_matrix_complex_alloc(nr,nc);
00113     }
00114
00115     gsl_complex h = gsl_complex_rect(h_,0.);
00116     gsl_complex zero = gsl_complex_rect(0.,0.);
00117     gsl_complex half = gsl_complex_rect(0.5,0.);
00118     gsl_complex one_sixth = gsl_complex_rect(1./6.,0.);
00119
00120     double t = t_i;
00121     double t_f = t_i + H;
00122     gsl_matrix_complex_memcpy(x_f,x_i);
00123     while (t<t_f)
00124     {
00125         A(t,A_val,params);
00126         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, zero, k[0]);
00127
00128         gsl_matrix_complex_memcpy(k_in[0], k[0]);
00129         gsl_matrix_complex_scale(k_in[0],half);
00130         gsl_matrix_complex_add(k_in[0],x_f);
00131
00132         A(t+0.5*h_,A_val,params);
00133         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], zero, k[1]);
00134
00135         gsl_matrix_complex_memcpy(k_in[1], k[1]);
00136         gsl_matrix_complex_scale(k_in[1],half);
00137         gsl_matrix_complex_add(k_in[1],x_f);
00138
00139         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], zero, k[2]);
00140
00141         gsl_matrix_complex_memcpy(k_in[2], k[2]);
00142         gsl_matrix_complex_add(k_in[2],x_f);
00143
00144         A(t+h_,A_val,params);
00145         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], zero, k[3]);
00146
00147         gsl_matrix_complex_memcpy(k_in[3],k[3]);
00148         gsl_matrix_complex_add(k_in[3],k[2]);
00149         gsl_matrix_complex_add(k_in[3],k[2]);
00150         gsl_matrix_complex_add(k_in[3],k[1]);
00151         gsl_matrix_complex_add(k_in[3],k[1]);
00152         gsl_matrix_complex_add(k_in[3],k[0]);
00153         gsl_matrix_complex_scale(k_in[3], one_sixth);
00154         gsl_matrix_complex_add(x_f,k_in[3]);
00155         t += h_;
00156     }
00157
00158     for (int i = 0; i < 4; ++i)
00159     {
00160         gsl_matrix_complex_free(k[i]);
00161         gsl_matrix_complex_free(k_in[i]);
00162     }

```

```
00163 }
```

3.5.1.4 rk4_fixed_final_matrix_floquet_type_real()

```
void rk4_fixed_final_matrix_floquet_type_real (
    gsl_matrix * x_i,
    double t_i,
    double H,
    double h,
    void(*) (double, gsl_matrix *, void *) A,
    gsl_matrix * x_f,
    void * params )
```

Fixed Step RK4 for real matrices for evolution of the form $\dot{X} = A_\lambda(t)X$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

x_i	Initial Matrix $x_i \in \mathbb{R}^{n \times m}$
t_i	Time when x_i is specified
H	Interval after which final x is required
h	Step size
<i>evol_func</i>	Function that computes $A_\lambda(t)$. The function should be of the form void A(double t, gsl_matrix* out, void* params)
x_f	Array to store the final x into. This should be preallocated
<i>params</i>	Parameters to be passed to A(t)

Definition at line 44 of file [diffEqSolvers.c](#).

```
00045 {
00046     int nr = x_i->size1;
00047     int nc = x_i->size2;
00048     gsl_matrix* k[4];
00049     gsl_matrix* k_in[4];
00050     gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
00051     for (int i = 0; i < 4; ++i)
00052     {
00053         k[i] = gsl_matrix_calloc(nr,nc);
00054         k_in[i] = gsl_matrix_calloc(nr,nc);
00055     }
00056
00057     double t = t_i;
00058     double t_f = t_i + H;
00059     gsl_matrix_memcpy(x_f, x_i);
00060     while (t < t_f)
00061     {
00062         A(t, A_val, params);
00063         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00064
00065         gsl_matrix_memcpy(k_in[0], k[0]);
00066         gsl_matrix_scale(k_in[0], 0.5);
00067         gsl_matrix_add(k_in[0], x_f);
00068
00069         A(t+0.5*h, A_val, params);
00070         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00071
00072         gsl_matrix_memcpy(k_in[1], k[1]);
00073         gsl_matrix_scale(k_in[1], 0.5);
00074         gsl_matrix_add(k_in[1], x_f);
00075
00076         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00077
00078         gsl_matrix_memcpy(k_in[2], k[2]);
```

```

00079     gsl_matrix_add(k_in[2],x_f);
00080
00081     A(t+h,A_val,params);
00082     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);
00083
00084     gsl_matrix_memcpy(k_in[3],k[3]);
00085     gsl_matrix_add(k_in[3],k[2]);
00086     gsl_matrix_add(k_in[3],k[2]);
00087     gsl_matrix_add(k_in[3],k[1]);
00088     gsl_matrix_add(k_in[3],k[1]);
00089     gsl_matrix_add(k_in[3],k[0]);
00090     gsl_matrix_scale(k_in[3], 1./6.);
00091     gsl_matrix_add(x_f,k_in[3]);
00092     t += h;
00093 }
00094
00095 for (int i = 0; i < 4; ++i)
00096 {
00097     gsl_matrix_free(k[i]);
00098     gsl_matrix_free(k_in[i]);
00099 }
00100 }

```

3.5.1.5 rk4_fixed_final_vector_real()

```

void rk4_fixed_final_vector_real (
    int ndim,
    double * x_i,
    double t_i,
    double H,
    double h,
    void(*) (double *, double, double *, void *) evol_func,
    double * x_f,
    void * params )

```

Fixed Step RK4 for vectors for general $\dot{x} = f_{\lambda}(x, t)$.

Implemented according to Computational Physics, Mark Newman (2013)

Parameters

<i>ndim</i>	Dimensionality of x
<i>x_i</i>	Initial Vector $x_i \in \mathbb{R}^n$
<i>t_i</i>	Time when <i>x_i</i> is specified
<i>H</i>	Interval after which final x is required
<i>h</i>	Step size
<i>evol_func</i>	Function that computes $\frac{dx}{dt}(x, t)$. The function should be of the form void evol_func(double* x, double t, double* x_dot, void* params)
<i>x_f</i>	Array to store the final x into. This should be preallocated
<i>params</i>	Parameters to be passed to evol_func

Definition at line 31 of file [diffEqSolvers.c](#).

```

00032 {
00033     double t = t_i;
00034     double t_f = t_i + H;
00035
00036     cblas_dcopy (ndim,x_i,sizeof(x_i[0]),x_f,sizeof(x_f[0]));
00037     while(t<t_f)
00038     {
00039         __rk4_single_vector(ndim,x_f,t,h,evol_func,x_f, params);
00040         t += h;
00041     }
00042 }

```

3.6 diffEqSolvers.c

```

00001
00002 #include "diffEqSolvers.h"
00003
00004 void __rk4_single_vector(int n, double* x, double t, double h, void (*evol_func)(double*, double,
00005 double*, void*), double* x_f, void* params)
00006 {
00007     double k1[n], k2[n], k3[n], k4[n], k2_in[n], k3_in[n], k4_in[n];
00008     evol_func(x,t,k1,params);
00009     cblas_dscal(n,h,k1,sizeof(k1[0]));
00010
00011     cblas_dcopy(n,x,sizeof(x[0]),k2_in,sizeof(k2_in[0]));
00012     cblas_daxpy(n,0.5,k1,sizeof(k1[0]),k2_in,sizeof(k2_in[0]));
00013     evol_func(k2_in,t+0.5*h,k2,params);
00014     cblas_dscal(n,h,k2,sizeof(k2[0]));
00015
00016     cblas_dcopy(n,x,sizeof(x[0]),k3_in,sizeof(k3_in[0]));
00017     cblas_daxpy(n,0.5,k2,sizeof(k2[0]),k3_in,sizeof(k3_in[0]));
00018     evol_func(k3_in,t+0.5*h,k3,params);
00019     cblas_dscal(n,h,k3,sizeof(k3[0]));
00020
00021     cblas_dcopy(n,x,sizeof(x[0]),k4_in,sizeof(k4_in[0]));
00022     cblas_daxpy(n,1.,k3,sizeof(k3[0]),k4_in,sizeof(k4_in[0]));
00023     evol_func(k4_in,t+h,k4,params);
00024     cblas_dscal(n,h,k4,sizeof(k4[0]));
00025
00026     for (int i = 0; i < n; ++i)
00027     {
00028         x_f[i] = x[i] + (1./6.)*(k1[i] + 2*k2[i] + 2*k3[i] + k4[i]);
00029     }
00030 }
00031 void rk4_fixed_final_vector_real(int ndim, double* x_i, double t_i, double H, double h, void
00032 (*evol_func)(double*, double, double*, void*), double* x_f, void* params)
00033 {
00034     double t = t_i;
00035     double t_f = t_i + H;
00036
00037     cblas_dcopy(ndim,x_i,sizeof(x_i[0]),x_f,sizeof(x_f[0]));
00038     while(t<t_f)
00039     {
00040         __rk4_single_vector(ndim,x_f,t,h,evol_func,x_f, params);
00041         t += h;
00042     }
00043 }
00044 void rk4_fixed_final_matrix_floquet_type_real(gsl_matrix* x_i, double t_i, double H, double h, void
00045 (*A)(double, gsl_matrix*, void*), gsl_matrix* x_f, void* params)
00046 {
00047     int nr = x_i->size1;
00048     int nc = x_i->size2;
00049     gsl_matrix* k[4];
00050     gsl_matrix* k_in[4];
00051     gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
00052     for (int i = 0; i < 4; ++i)
00053     {
00054         k[i] = gsl_matrix_calloc(nr,nc);
00055         k_in[i] = gsl_matrix_calloc(nr,nc);
00056     }
00057
00058     double t = t_i;
00059     double t_f = t_i + H;
00060     gsl_matrix_memcpy(x_f,x_i);
00061     while (t<t_f)
00062     {
00063         A(t,A_val,params);
00064         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00065
00066         gsl_matrix_memcpy(k_in[0], k[0]);
00067         gsl_matrix_scale(k_in[0],0.5);
00068         gsl_matrix_add(k_in[0],x_f);
00069
00070         A(t+0.5*h,A_val,params);
00071         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00072
00073         gsl_matrix_memcpy(k_in[1], k[1]);
00074         gsl_matrix_scale(k_in[1],0.5);
00075         gsl_matrix_add(k_in[1],x_f);
00076
00077         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00078
00079         gsl_matrix_memcpy(k_in[2], k[2]);
00080         gsl_matrix_scale(k_in[2],0.5);
00081         gsl_matrix_add(k_in[2],x_f);
00082
00083         A(t+h,A_val,params);
00084         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

```



```

00083
00084     gsl_matrix_memcpy(k_in[3], k[3]);
00085     gsl_matrix_add(k_in[3], k[2]);
00086     gsl_matrix_add(k_in[3], k[2]);
00087     gsl_matrix_add(k_in[3], k[1]);
00088     gsl_matrix_add(k_in[3], k[1]);
00089     gsl_matrix_add(k_in[3], k[0]);
00090     gsl_matrix_scale(k_in[3], 1./6.);
00091     gsl_matrix_add(x_f, k_in[3]);
00092     t += h;
00093 }
00094
00095 for (int i = 0; i < 4; ++i)
00096 {
00097     gsl_matrix_free(k[i]);
00098     gsl_matrix_free(k_in[i]);
00099 }
00100 }
00101
00102 void rk4_fixed_final_matrix_floquet_type_complex(gsl_matrix_complex* x_i, double t_i, double H, double
h_, void (*A)(double, gsl_matrix_complex*, void*), gsl_matrix_complex* x_f, void* params)
00103 {
00104     int nr = x_i->size1;
00105     int nc = x_i->size2;
00106     gsl_matrix_complex* k[4];
00107     gsl_matrix_complex* k_in[4];
00108     gsl_matrix_complex* A_val = gsl_matrix_complex_alloc(nr, nc);
00109     for (int i = 0; i < 4; ++i)
00110     {
00111         k[i] = gsl_matrix_complex_alloc(nr, nc);
00112         k_in[i] = gsl_matrix_complex_alloc(nr, nc);
00113     }
00114
00115     gsl_complex h = gsl_complex_rect(h_, 0.);
00116     gsl_complex zero = gsl_complex_rect(0., 0.);
00117     gsl_complex half = gsl_complex_rect(0.5, 0.);
00118     gsl_complex one_sixth = gsl_complex_rect(1./6., 0.);
00119
00120     double t = t_i;
00121     double t_f = t_i + H;
00122     gsl_matrix_complex_memcpy(x_f, x_i);
00123     while (t < t_f)
00124     {
00125         A(t, A_val, params);
00126         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, zero, k[0]);
00127
00128         gsl_matrix_complex_memcpy(k_in[0], k[0]);
00129         gsl_matrix_complex_scale(k_in[0], half);
00130         gsl_matrix_complex_add(k_in[0], x_f);
00131
00132         A(t+0.5*h_, A_val, params);
00133         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], zero, k[1]);
00134
00135         gsl_matrix_complex_memcpy(k_in[1], k[1]);
00136         gsl_matrix_complex_scale(k_in[1], half);
00137         gsl_matrix_complex_add(k_in[1], x_f);
00138
00139         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], zero, k[2]);
00140
00141         gsl_matrix_complex_memcpy(k_in[2], k[2]);
00142         gsl_matrix_complex_add(k_in[2], x_f);
00143
00144         A(t+h_, A_val, params);
00145         gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], zero, k[3]);
00146
00147         gsl_matrix_complex_memcpy(k_in[3], k[3]);
00148         gsl_matrix_complex_add(k_in[3], k[2]);
00149         gsl_matrix_complex_add(k_in[3], k[2]);
00150         gsl_matrix_complex_add(k_in[3], k[1]);
00151         gsl_matrix_complex_add(k_in[3], k[1]);
00152         gsl_matrix_complex_add(k_in[3], k[0]);
00153         gsl_matrix_complex_scale(k_in[3], one_sixth);
00154         gsl_matrix_complex_add(x_f, k_in[3]);
00155         t += h_;
00156     }
00157
00158     for (int i = 0; i < 4; ++i)
00159     {
00160         gsl_matrix_complex_free(k[i]);
00161         gsl_matrix_complex_free(k_in[i]);
00162     }
00163 }
00164
00165 void rk4_adaptive_final_matrix_floquet_type_real(gsl_matrix* x_i, double t_i, double H, double delta,
void (*A)(double, gsl_matrix*, void*), gsl_matrix* x_f, void* params)
00166 {
00167     double h_min = H/RK4_MAX_SLICES;

```

```

00168     int nr = x_i->size1;
00169     int nc = x_i->size2;
00170     gsl_matrix* k[4];
00171     gsl_matrix* k_in[4];
00172     gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
00173     for (int i = 0; i < 4; ++i)
00174     {
00175         k[i] = gsl_matrix_calloc(nr,nc);
00176         k_in[i] = gsl_matrix_calloc(nr,nc);
00177     }
00178
00179     double t = t_i;
00180     double t_f = t_i + H;
00181     double h = H/10.; // Initial h. This is a bit conservative, but will be refined further.
00182     gsl_matrix_memcpy(x_f,x_i);
00183
00184     gsl_matrix* x1 = gsl_matrix_alloc(nr,nc);
00185     gsl_matrix* x2 = gsl_matrix_alloc(nr,nc);
00186
00187     while (t<t_f)
00188     {
00189         // Evaluate x1
00190         gsl_matrix_memcpy(x1,x_f);
00191
00192         A(t,A_val,params);
00193         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00194
00195         gsl_matrix_memcpy(k_in[0], k[0]);
00196         gsl_matrix_scale(k_in[0],0.5);
00197         gsl_matrix_add(k_in[0],x_f);
00198
00199         A(t+0.5*h,A_val,params);
00200         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00201
00202         gsl_matrix_memcpy(k_in[1], k[1]);
00203         gsl_matrix_scale(k_in[1],0.5);
00204         gsl_matrix_add(k_in[1],x_f);
00205
00206         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00207
00208         gsl_matrix_memcpy(k_in[2], k[2]);
00209         gsl_matrix_add(k_in[2],x_f);
00210
00211         A(t+h,A_val,params);
00212         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);
00213
00214         gsl_matrix_memcpy(k_in[3],k[3]);
00215         gsl_matrix_add(k_in[3],k[2]);
00216         gsl_matrix_add(k_in[3],k[2]);
00217         gsl_matrix_add(k_in[3],k[1]);
00218         gsl_matrix_add(k_in[3],k[1]);
00219         gsl_matrix_add(k_in[3],k[0]);
00220         gsl_matrix_scale(k_in[3], 1./6.);
00221         gsl_matrix_add(x1,k_in[3]);
00222
00223         t += h;
00224
00225         A(t,A_val,params);
00226         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);
00227
00228         gsl_matrix_memcpy(k_in[0], k[0]);
00229         gsl_matrix_scale(k_in[0],0.5);
00230         gsl_matrix_add(k_in[0],x1);
00231
00232         A(t+0.5*h,A_val,params);
00233         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);
00234
00235         gsl_matrix_memcpy(k_in[1], k[1]);
00236         gsl_matrix_scale(k_in[1],0.5);
00237         gsl_matrix_add(k_in[1],x1);
00238
00239         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);
00240
00241         gsl_matrix_memcpy(k_in[2], k[2]);
00242         gsl_matrix_add(k_in[2],x1);
00243
00244         A(t+h,A_val,params);
00245         gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);
00246
00247         gsl_matrix_memcpy(k_in[3],k[3]);
00248         gsl_matrix_add(k_in[3],k[2]);
00249         gsl_matrix_add(k_in[3],k[2]);
00250         gsl_matrix_add(k_in[3],k[1]);
00251         gsl_matrix_add(k_in[3],k[1]);
00252         gsl_matrix_add(k_in[3],k[0]);
00253         gsl_matrix_scale(k_in[3], 1./6.);
00254         gsl_matrix_add(x1,k_in[3]);

```

```

00255
00256     t -= h;
00257
00258     // Evaluate x2
00259     gsl_matrix_memcpy(x2,x_f);
00260
00261     A(t,A_val,params);
00262     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, x_f, 0., k[0]);
00263
00264     gsl_matrix_memcpy(k_in[0], k[0]);
00265     gsl_matrix_scale(k_in[0],0.5);
00266     gsl_matrix_add(k_in[0],x_f);
00267
00268     A(t+h,A_val,params);
00269     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[0], 0., k[1]);
00270
00271     gsl_matrix_memcpy(k_in[1], k[1]);
00272     gsl_matrix_scale(k_in[1],0.5);
00273     gsl_matrix_add(k_in[1],x_f);
00274
00275     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[1], 0., k[2]);
00276
00277     gsl_matrix_memcpy(k_in[2], k[2]);
00278     gsl_matrix_add(k_in[2],x_f);
00279
00280     A(t+2*h,A_val,params);
00281     gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[2], 0., k[3]);
00282
00283     gsl_matrix_memcpy(k_in[3],k[3]);
00284     gsl_matrix_add(k_in[3],k[2]);
00285     gsl_matrix_add(k_in[3],k[2]);
00286     gsl_matrix_add(k_in[3],k[1]);
00287     gsl_matrix_add(k_in[3],k[1]);
00288     gsl_matrix_add(k_in[3],k[0]);
00289     gsl_matrix_scale(k_in[3], 1./6.);
00290     gsl_matrix_add(x2,k_in[3]);
00291
00292     // Evaluate Error
00293     gsl_matrix_sub(x2,x1);
00294     double rho_to_the_fourth = (30.*h*delta)/GSL_MAX(gsl_matrix_max(x2), -1.*gsl_matrix_min(x2));
00295     rho_to_the_fourth = pow(rho_to_the_fourth,0.25);
00296     if (rho_to_the_fourth>1)
00297     {
00298         gsl_matrix_memcpy(x_f,x1);
00299         gsl_matrix_scale(x2, -1./15.);
00300         gsl_matrix_add(x_f,x2);
00301         t += 2*h;
00302         h = h*(GSL_MIN(rho_to_the_fourth, RK4_MAX_SCALE));
00303         h = GSL_MIN(0.5*(t_f-t),h);
00304     }
00305     else if(h > h_min)
00306     {
00307         h = h*(GSL_MAX(rho_to_the_fourth, RK4_MIN_SCALE));
00308     }
00309     else
00310     {
00311         gsl_matrix_memcpy(x_f,x1);
00312         gsl_matrix_scale(x2, -1./15.);
00313         gsl_matrix_add(x_f,x2);
00314         t += 2*h;
00315         h = h_min;
00316     }
00317 }
00318
00319 gsl_matrix_free(A_val);
00320
00321 for (int i = 0; i < 4; ++i)
00322 {
00323     gsl_matrix_free(k[i]);
00324     gsl_matrix_free(k_in[i]);
00325 }
00326
00327 gsl_matrix_free(x1);
00328 gsl_matrix_free(x2);
00329 }
00330
00331 void __midpoint_method(gsl_matrix* x, double t_i, double H, double h, void (*A)(double, gsl_matrix*,
void*), gsl_matrix* x_f, void* params, gsl_matrix* y, gsl_matrix* eval)
00332 {
00333     double t = t_i;
00334     double t_f = t_i + H;
00335     double h_2 = h/2.;
00336     gsl_matrix_memcpy(x_f,x);
00337
00338     A(t,eval,params);
00339     gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h_2,eval,x_f,0.,y);
00340     gsl_matrix_add(y,x_f);

```

```

00341
00342     A(t+h_2,eval,params);
00343     gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h,eval,y,1.,x_f);
00344
00345     t+=h;
00346
00347     while (t<t_f)
00348     {
00349         A(t,eval,params);
00350         gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h,eval,x_f,1.,y);
00351
00352         A(t+h_2,eval,params);
00353         gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h,eval,y,1.,x_f);
00354
00355         t+= h;
00356     }
00357
00358     A(t,eval,params);
00359     gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h_2,eval,x_f,1.,y);
00360     gsl_matrix_add(x_f,y);
00361     gsl_matrix_scale(x_f,0.5);
00362 }
00363 }
00364
00365 double __bulsto_final_matrix_floquet_type_real_main(gsl_matrix* x_i, double t_i, double H, double
    delta, void (*A)(double, gsl_matrix*, void*), gsl_matrix* x_f, void* params, gsl_matrix* y,
    gsl_matrix** eval, gsl_matrix** R1, gsl_matrix** R2, gsl_matrix* epsilon)
00366 {
00367     //int ndim = x_i->size1;
00368
00369
00370     //printf("0 ERR %e %e %e %e\n", gsl_matrix_get(x_i,0,0), gsl_matrix_get(x_i,0,1),
    gsl_matrix_get(x_i,1,0), gsl_matrix_get(x_i,1,1));
00371
00372     int n = 1;
00373     double h = H;
00374     __midpoint_method(x_i, t_i, H, h, A, R1[0], params, y, eval);
00375     double error = HUGE_VAL;
00376
00377     gsl_matrix** temp;
00378
00379     //printf("%d %e %e %e %e %e\n", n, error, gsl_matrix_get(R1[0],0,0), gsl_matrix_get(R1[0],0,1),
    gsl_matrix_get(R1[0],1,0), gsl_matrix_get(R1[0],1,1));
00380     while (error > H*delta && n<BULSTO_STEP_MAX)
00381     {
00382         n++;
00383         h = H/n;
00384
00385         // Swapping the arrays of matrices to save space
00386         temp = R2;
00387         R2 = R1;
00388         R1 = temp;
00389
00390         double scaler = n/(n-1.);
00391         scaler *= scaler;
00392
00393         double scale = 1.;
00394         __midpoint_method(x_i, t_i, H, h, A, R1[0], params, y, eval);
00395         for (int m = 1; m < n; ++m)
00396         {
00397             scale *= scaler;
00398             gsl_matrix_memcpy(epsilon,R1[m-1]);
00399             gsl_matrix_sub(epsilon,R2[m-1]);
00400             gsl_matrix_scale(epsilon, 1./(scale-1.));
00401
00402             gsl_matrix_memcpy(R1[m],R1[m-1]);
00403             gsl_matrix_add(R1[m],epsilon);
00404         }
00405         error = GSL_MAX(gsl_matrix_max(epsilon), -1.*gsl_matrix_min(epsilon));
00406         //printf("%d %e %e %e %e %e %e\n", n, H, error, gsl_matrix_get(R1[0],0,0),
    gsl_matrix_get(R1[0],0,1), gsl_matrix_get(R1[0],1,0), gsl_matrix_get(R1[0],1,1));
00407     }
00408
00409     gsl_matrix_memcpy(x_f,R1[n-1]);
00410
00411     return error;
00412 }
00413
00414 double __bulsto_final_matrix_floquet_type_real_runner(int nlayer, gsl_matrix* x_i, double t_i, double
    H, double delta, void (*A)(double, gsl_matrix*, void*), gsl_matrix* x_f, void* params, gsl_matrix* y,
    gsl_matrix** eval, gsl_matrix** R1, gsl_matrix** R2, gsl_matrix* epsilon)
00415 {
00416     int ndim = x_i->size1;
00417     gsl_matrix* xfin = gsl_matrix_alloc(ndim,ndim);
00418     double error = __bulsto_final_matrix_floquet_type_real_main(x_i, t_i, H, delta, A, xfin, params,
    y, eval, R1, R2, epsilon);
00419     nlayer++;

```

```

00420     if (error > H*delta && nlayer < BULSTO_MAX_LAYERS)
00421     {
00422         error = __bulsto_final_matrix_floquet_type_real_runner(nlayer, x_i, t_i, H/2., delta, A, xfin,
00423         params, y, eval, R1, R2, epsilon);
00424         error += __bulsto_final_matrix_floquet_type_real_runner(nlayer, xfin, t_i+(H/2.), H/2., delta,
00425         A, xfin, params, y, eval, R1, R2, epsilon);
00426     }
00427     gsl_matrix_memcpy(x_f,xfin);
00428     gsl_matrix_free(xfin);
00429     return error;
00430 }
00431 void bulsto_final_matrix_floquet_type_real(gsl_matrix* x_i, double t_i, double H, double delta, void
00432 (*A)(double, gsl_matrix*, void*), gsl_matrix* x_f, void* params)
00433 {
00434     // This program only initializes and provides and frees temp variables
00435     int ndim = x_i->size1;
00436     gsl_matrix** R1 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
00437     gsl_matrix** R2 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
00438     for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
00439     {
00440         R1[i] = gsl_matrix_alloc(ndim,ndim);
00441         R2[i] = gsl_matrix_alloc(ndim,ndim);
00442     }
00443     gsl_matrix* y = gsl_matrix_alloc(ndim,ndim);
00444     gsl_matrix* eval = gsl_matrix_alloc(ndim,ndim);
00445     gsl_matrix* epsilon = gsl_matrix_calloc(ndim,ndim);
00446     __bulsto_final_matrix_floquet_type_real_runner(0, x_i, t_i, H, delta, A, x_f, params, y, eval, R1,
00447     R2, epsilon);
00448     //printf("%e %e %e\n",error, H, error/H);
00449     for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
00450     {
00451         gsl_matrix_free(R1[i]);
00452         gsl_matrix_free(R2[i]);
00453     }
00454     free(R1);
00455     free(R2);
00456     gsl_matrix_free(y);
00457     gsl_matrix_free(eval);
00458     gsl_matrix_free(epsilon);
00459 }

```

3.7 src/floquet.c File Reference

```
#include "floquet.h"
```

Functions

- int [floquet_get_stability_reals_general](#) (int n, void(*A)(double, gsl_matrix *, void *), void *params, double T, gsl_complex *largest_multiplier, double *largest_multiplier_abs)

Function which checks if the function is Floquet Stable or unstable for a general function.

- void [floquet_get_stability_array_real_single_param_general](#) (int n, void(*A)(double, gsl_matrix *, void *), double T, double start, double end, int nstep, int *stability, gsl_complex *largest_multiplier, double *largest_multiplier_abs)

CPU Parallelized Function which iterates over a range of a parameter and checks if the function is Floquet Stable or unstable for a general function.

- void [floquet_get_stability_array_real_double_param_general](#) (int n, void(*A)(double, gsl_matrix *, void *), double T, double *start, double *end, int *nstep, int **stability, gsl_complex **largest_multiplier, double **largest_multiplier_abs)

CPU Parallelized Function which iterates over the ranges of 2 parameters and checks if the function is Floquet Stable or unstable for a general function.

3.7.1 Function Documentation

3.7.1.1 floquet_get_stability_array_real_double_param_general()

```
void floquet_get_stability_array_real_double_param_general (
    int n,
    void(*) (double, gsl_matrix *, void *) A,
    double T,
    double * start,
    double * end,
    int * nstep,
    int ** stability,
    gsl_complex ** largest_multiplier,
    double ** largest_multiplier_abs )
```

CPU Parallelized Function which iterates over the ranges of 2 parameters and checks if the function is Floquet Stable or unstable for a general function.

Run `floquet_get_stability_reals_general` on the ranges of 2 parameters and stores the stability, floquet multiplier corresponding to the largest absolute value, and the absolute value of the largest floquet multiplier. This is a memory naive implementation.

Parameters

<i>n</i>	Number of elements of vector that $A(t)$ operates on
<i>A</i>	$A(t)$ matrix corresponding to the equation $\dot{x} = Ax$. The void* should be resolved to a double inside the function because a double* with 2 doubles would be passed.
<i>T</i>	Period of the evolution function $A(t)$ s.t. $A(t + T) = A(t) \forall t \in \mathbb{R}$
<i>start</i>	Starting values of the parameters
<i>end</i>	Ending values of the parameters
<i>nstep</i>	Number of steps to take inclusive of the first and last values keeping the other parameter constant. Should be at least 2. Behaviour not defined otherwise.
<i>largest_multiplier</i>	Matrix to store the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.
<i>largest_multiplier_abs</i>	Matrix to store the absolute value of the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.

Definition at line 110 of file `floquet.c`.

```
00111 {
00112     gsl_complex** mult_temp;
00113     double** mult_abs_temp;
00114
00115     if (largest_multiplier)
00116     {
00117         mult_temp = largest_multiplier;
00118     }
00119     else
00120     {
00121         mult_temp = (gsl_complex**) malloc(nstep[0]*sizeof(gsl_complex*));
00122         for (int i = 0; i < nstep[0]; ++i)
00123         {
00124             mult_temp[i] = (gsl_complex*) malloc(nstep[1]*sizeof(gsl_complex));
00125         }
00126     }
00127
00128     if (largest_multiplier_abs)
00129     {
```

```

00130     mult_abs_temp = largest_multiplier_abs;
00131 }
00132 else
00133 {
00134     mult_abs_temp = (double**) malloc(nstep[0]*sizeof(double*));
00135     for (int i = 0; i < nstep[0]; ++i)
00136     {
00137         mult_abs_temp[i] = (double*) malloc(nstep[1]*sizeof(double));
00138     }
00139 }
00140
00141 double step[2] = {(end[0]-start[0])/(nstep[0]-1), (end[1]-start[1])/(nstep[1]-1)};
00142 #pragma omp parallel for collapse(2) schedule(guided)
00143 for (int i = 0; i < nstep[0]; ++i)
00144 {
00145     for (int j = 0; j < nstep[1]; ++j)
00146     {
00147         double param[2] = {start[0] + step[0]*i, start[1] + step[1]*j };
00148         stability[i][j] =
floquet_get_stability_reals_general(n,A,param,T,&mult_temp[i][j],&mult_abs_temp[i][j]);
00149     }
00150 }
00151
00152 if(!largest_multiplier)
00153 {
00154     for (int i = 0; i < nstep[0]; ++i)
00155     {
00156         free(mult_temp[i]);
00157     }
00158     free(mult_temp);
00159 }
00160
00161 if(!largest_multiplier_abs)
00162 {
00163     for (int i = 0; i < nstep[0]; ++i)
00164     {
00165         free(mult_abs_temp[i]);
00166     }
00167     free(mult_abs_temp);
00168 }
00169 }

```

References [floquet_get_stability_reals_general\(\)](#).

3.7.1.2 floquet_get_stability_array_real_single_param_general()

```

void floquet_get_stability_array_real_single_param_general (
    int n,
    void(*) (double, gsl_matrix *, void *) A,
    double T,
    double start,
    double end,
    int nstep,
    int * stability,
    gsl_complex * largest_multiplier,
    double * largest_multiplier_abs )

```

CPU Parallelized Function which iterates over a range of a parameter and checks if the function is Floquet Stable or unstable for a general function.

Run `floquet_get_stability_reals_general` on a range of a parameter and stores the stability, floquet multiplier corresponding to the largest absolute value, and the absolute value of the largest floquet multiplier. This is a memory naive implementation.

Parameters

<i>n</i>	Number of elements of vector that $A(t)$ operates on
----------	--

Parameters

<i>A</i>	$A(t)$ matrix corresponding to the equation $\dot{x} = Ax$. The void* should be resolved to a double inside the function because a double* with a single double would be passed.
<i>T</i>	Period of the evolution function $A(t)$ s.t. $A(t + T) = A(t) \forall t \in \mathbb{R}$
<i>start</i>	Starting value of the parameter
<i>end</i>	Ending value of the parameter
<i>nstep</i>	Number of steps to take inclusive of the first and last values. Should be at least 2. Behaviour not defined otherwise.
<i>largest_multiplier</i>	Array to store the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.
<i>largest_multiplier_abs</i>	Array to store the absolute value of the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.

Definition at line 69 of file [floquet.c](#).

```

00070 {
00071     gsl_complex* mult_temp;
00072     double* mult_abs_temp;
00073
00074     if (largest_multiplier)
00075     {
00076         mult_temp = largest_multiplier;
00077     }
00078     else
00079     {
00080         mult_temp = (gsl_complex*) malloc(nstep*sizeof(gsl_complex));
00081     }
00082
00083     if (largest_multiplier_abs)
00084     {
00085         mult_abs_temp = largest_multiplier_abs;
00086     }
00087     else
00088     {
00089         mult_abs_temp = (double*) malloc(nstep*sizeof(double));
00090     }
00091
00092     double step = (end-start)/(nstep-1);
00093     #pragma omp parallel for
00094     for (int i = 0; i < nstep; ++i)
00095     {
00096         double param = start + step*i;
00097         stability[i] = floquet_get_stability_reals_general(n,A,&param,T,mult_temp+i,mult_abs_temp+i);
00098     }
00099
00100     if(!largest_multiplier)
00101     {
00102         free(mult_temp);
00103     }
00104     if(!largest_multiplier_abs)
00105     {
00106         free(mult_abs_temp);
00107     }
00108 }

```

References [floquet_get_stability_reals_general\(\)](#).

3.7.1.3 floquet_get_stability_reals_general()

```

int floquet_get_stability_reals_general (
    int n,
    void(*) (double, gsl_matrix *, void *) A,
    void * params,
    double T,

```



```
gsl_complex * largest_multiplier,  
double * largest_multiplier_abs )
```

Function which checks if the function is Floquet Stable or unstable for a general function.

Naive implementation with the elements of the $X(T)$ matrix calculated to precision ERR_TOL

Parameters

n	Number of elements of vector that $A(t)$ operates on
A	$A(t)$ matrix corresponding to the equation $\dot{x} = Ax$
$params$	Parameters to be passed to $A(t)$
T	Period of the evolution function $A(t)$ s.t. $A(t + T) = A(t) \forall t \in \mathbb{R}$
$largest_multiplier$	Pointer to store the largest (by abs) computed Floquet multiplier into. No multiplier will be stored if NULL is passed.
$largest_multiplier_abs$	Pointer to store the absolute value of the largest (by abs) computed Floquet multiplier into. No multiplier will be stored if NULL is passed.

Returns

1 if Stable, -1 if unstable, 0 if periodic or indeterminate to accuracy ERR_EIGEN_TOL. In rare cases, 2 would be returned if none of the computed floquet multipliers lead to instability, but not all of multipliers could be computed.

Definition at line 4 of file [floquet.c](#).

```

00005 {
00006     gsl_matrix* X = gsl_matrix_alloc(n,n);
00007     gsl_matrix_set_identity(X);
00008
00009     gsl_matrix* B = gsl_matrix_alloc(n,n);
00010     bulsto_final_matrix_floquet_type_real(X, 0., T, ERR_TOL, A, B, params);
00011
00012     gsl_vector_complex* eigenvals = gsl_vector_complex_alloc(n);
00013
00014     gsl_eigen_nonsymm_workspace* w = gsl_eigen_nonsymm_alloc(n);
00015
00016     int n_eigenvals_evaluated = n;
00017
00018     int err_code = gsl_eigen_nonsymm(B,eigenvals,w);
00019     if(err_code)
00020     {
00021         n_eigenvals_evaluated = w->n_evals;
00022     }
00023
00024     double mult_max_abs = -HUGE_VAL;
00025     gsl_complex mult_max;
00026     double ev_test;
00027     for (int i = 0; i < n_eigenvals_evaluated; ++i)
00028     {
00029         ev_test = gsl_complex_logabs(gsl_vector_complex_get(eigenvals,i));
00030         if (mult_max_abs < ev_test)
00031         {
00032             mult_max_abs = ev_test;
00033             mult_max = gsl_vector_complex_get(eigenvals,i);
00034         }
00035     }
00036
00037     if (largest_multiplier != NULL)
00038     {
00039         *largest_multiplier = mult_max;
00040     }
00041     if (largest_multiplier_abs != NULL)
00042     {
00043         *largest_multiplier_abs = gsl_complex_abs(mult_max);
00044     }
00045
00046     gsl_eigen_nonsymm_free(w);
00047     gsl_vector_complex_free(eigenvals);
00048     gsl_matrix_free(B);
00049     gsl_matrix_free(X);
00050
00051     if (mult_max_abs > ERR_EIGEN_TOL)
00052     {
00053         return 1;
00054     }
00055     else if (mult_max_abs < (-ERR_EIGEN_TOL))
00056     {
00057         if (n_eigenvals_evaluated < n)
00058         {
00059             return 2;

```

```

00060     }
00061     return -1;
00062 }
00063 else
00064 {
00065     return 0;
00066 }
00067 }

```

References [bulsto_final_matrix_floquet_type_real\(\)](#), [ERR_EIGEN_TOL](#), and [ERR_TOL](#).

Referenced by [floquet_get_stability_array_real_double_param_general\(\)](#), and [floquet_get_stability_array_real_single_param_general\(\)](#).

3.8 floquet.c

```

00001
00002 #include "floquet.h"
00003
00004 int floquet_get_stability_reals_general(int n, void (*A)(double, gsl_matrix*, void*), void* params,
    double T, gsl_complex* largest_multiplier, double* largest_multiplier_abs)
00005 {
00006     gsl_matrix* X = gsl_matrix_alloc(n,n);
00007     gsl_matrix_set_identity(X);
00008
00009     gsl_matrix* B = gsl_matrix_alloc(n,n);
00010     bulsto_final_matrix_floquet_type_real(X, 0., T, ERR_TOL, A, B, params);
00011
00012     gsl_vector_complex* eigenvals = gsl_vector_complex_alloc(n);
00013
00014     gsl_eigen_nonsymm_workspace* w = gsl_eigen_nonsymm_alloc(n);
00015
00016     int n_eigenvals_evaluated = n;
00017
00018     int err_code = gsl_eigen_nonsymm(B,eigenvals,w);
00019     if(err_code)
00020     {
00021         n_eigenvals_evaluated = w->n_evals;
00022     }
00023
00024     double mult_max_abs = -HUGE_VAL;
00025     gsl_complex mult_max;
00026     double ev_test;
00027     for (int i = 0; i < n_eigenvals_evaluated; ++i)
00028     {
00029         ev_test = gsl_complex_logabs(gsl_vector_complex_get(eigenvals,i));
00030         if (mult_max_abs < ev_test)
00031         {
00032             mult_max_abs = ev_test;
00033             mult_max = gsl_vector_complex_get(eigenvals,i);
00034         }
00035     }
00036
00037     if (largest_multiplier != NULL)
00038     {
00039         *largest_multiplier = mult_max;
00040     }
00041     if (largest_multiplier_abs != NULL)
00042     {
00043         *largest_multiplier_abs = gsl_complex_abs(mult_max);
00044     }
00045
00046     gsl_eigen_nonsymm_free(w);
00047     gsl_vector_complex_free(eigenvals);
00048     gsl_matrix_free(B);
00049     gsl_matrix_free(X);
00050
00051     if (mult_max_abs > ERR_EIGEN_TOL)
00052     {
00053         return 1;
00054     }
00055     else if (mult_max_abs < (-ERR_EIGEN_TOL))
00056     {
00057         if (n_eigenvals_evaluated < n)
00058         {
00059             return 2;
00060         }
00061         return -1;
00062     }
00063     else
00064     {
00065         return 0;

```

```

00066     }
00067 }
00068
00069 void floquet_get_stability_array_real_single_param_general(int n, void (*A)(double, gsl_matrix*,
    void*), double T, double start, double end, int nstep, int* stability, gsl_complex*
    largest_multiplier, double* largest_multiplier_abs)
00070 {
00071     gsl_complex* mult_temp;
00072     double* mult_abs_temp;
00073
00074     if (largest_multiplier)
00075     {
00076         mult_temp = largest_multiplier;
00077     }
00078     else
00079     {
00080         mult_temp = (gsl_complex*) malloc(nstep*sizeof(gsl_complex));
00081     }
00082
00083     if (largest_multiplier_abs)
00084     {
00085         mult_abs_temp = largest_multiplier_abs;
00086     }
00087     else
00088     {
00089         mult_abs_temp = (double*) malloc(nstep*sizeof(double));
00090     }
00091
00092     double step = (end-start)/(nstep-1);
00093     #pragma omp parallel for
00094     for (int i = 0; i < nstep; ++i)
00095     {
00096         double param = start + step*i;
00097         stability[i] = floquet_get_stability_reals_general(n,A,&param,T,mult_temp+i,mult_abs_temp+i);
00098     }
00099
00100     if(!largest_multiplier)
00101     {
00102         free(mult_temp);
00103     }
00104     if(!largest_multiplier_abs)
00105     {
00106         free(mult_abs_temp);
00107     }
00108 }
00109
00110 void floquet_get_stability_array_real_double_param_general(int n, void (*A)(double, gsl_matrix*,
    void*), double T, double* start, double* end, int* nstep, int** stability, gsl_complex**
    largest_multiplier, double** largest_multiplier_abs)
00111 {
00112     gsl_complex** mult_temp;
00113     double** mult_abs_temp;
00114
00115     if (largest_multiplier)
00116     {
00117         mult_temp = largest_multiplier;
00118     }
00119     else
00120     {
00121         mult_temp = (gsl_complex**) malloc(nstep[0]*sizeof(gsl_complex));
00122         for (int i = 0; i < nstep[0]; ++i)
00123         {
00124             mult_temp[i] = (gsl_complex*) malloc(nstep[1]*sizeof(gsl_complex));
00125         }
00126     }
00127
00128     if (largest_multiplier_abs)
00129     {
00130         mult_abs_temp = largest_multiplier_abs;
00131     }
00132     else
00133     {
00134         mult_abs_temp = (double**) malloc(nstep[0]*sizeof(double));
00135         for (int i = 0; i < nstep[0]; ++i)
00136         {
00137             mult_abs_temp[i] = (double*) malloc(nstep[1]*sizeof(double));
00138         }
00139     }
00140
00141     double step[2] = {(end[0]-start[0])/(nstep[0]-1), (end[1]-start[1])/(nstep[1]-1)};
00142     #pragma omp parallel for collapse(2) schedule(guided)
00143     for (int i = 0; i < nstep[0]; ++i)
00144     {
00145         for (int j = 0; j < nstep[1]; ++j)
00146         {
00147             double param[2] = {start[0] + step[0]*i, start[1] + step[1]*j };
00148             stability[i][j] =

```

```

    floquet_get_stability_reals_general(n,A,param,T,&mult_temp[i][j],&mult_abs_temp[i][j]);
00149     }
00150 }
00151
00152 if(!largest_multiplier)
00153 {
00154     for (int i = 0; i < nstep[0]; ++i)
00155     {
00156         free(mult_temp[i]);
00157     }
00158     free(mult_temp);
00159 }
00160
00161 if(!largest_multiplier_abs)
00162 {
00163     for (int i = 0; i < nstep[0]; ++i)
00164     {
00165         free(mult_abs_temp[i]);
00166     }
00167     free(mult_abs_temp);
00168 }
00169 }

```

3.9 src/hill_meissner.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Functions

- void **print_int_matrix_to_file_csv** (int n1, int n2, int **m, FILE *file)
- void **print_double_matrix_to_file_csv** (int n1, int n2, double **m, FILE *file)
- void **hill_meissner** (double t, gsl_matrix *A_val, void *param)
- int **main** ()

3.10 hill_meissner.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00008 {
00009     for (int i = 0; i < n1; ++i)
00010     {
00011         for (int j = 0; j < n2-1; ++j)
00012         {
00013             fprintf(file,"%d,",m[i][j]);
00014         }
00015         fprintf(file,"%d\n",m[i][n2-1]);
00016     }
00017 }
00018
00019 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00020 {
00021     for (int i = 0; i < n1; ++i)
00022     {
00023         for (int j = 0; j < n2-1; ++j)
00024         {
00025             fprintf(file,"%lf,",m[i][j]);
00026         }
00027         fprintf(file,"%lf\n",m[i][n2-1]);
00028     }
00029 }

```

```

00030
00031
00032 void hill_meissner(double t, gsl_matrix* A_val, void* param)
00033 {
00034     double* par_temp = (double*) param;
00035     gsl_matrix_set_zero(A_val);
00036     gsl_matrix_set(A_val,0,1,1.);
00037     gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*GSL_SIGN(t-M_PI)));
00038 }
00039
00040 int main()
00041 {
00042     int n = 2;
00043     double T = 2*M_PI;
00044
00045     double start[2] = {9.,-1.};
00046     double end[2] = {0., 9.};
00047     int nstep[2] = {64,64};
00048     int** stability = (int**) malloc(nstep[0]*sizeof(int*));
00049     gsl_complex** largest_multiplier = NULL;
00050     double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));
00051
00052     for (int i = 0; i < nstep[0]; ++i)
00053     {
00054         stability[i] = (int*) malloc(nstep[1]*sizeof(int));
00055         largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
00056     }
00057
00058     floquet_get_stability_array_real_double_param_general(n, hill_meissner, T, start, end, nstep,
00059 stability, largest_multiplier, largest_multiplier_abs);
00060
00061     FILE* file = fopen("stability.csv","w");
00062     print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
00063     fclose(file);
00064
00065     file = fopen("largest_multiplier_abs.csv","w");
00066     print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
00067     fclose(file);
00068
00069     file = fopen("extraparams.txt","w");
00070     fprintf(file,"Hill-Meissner Stability Plot\n");
00071     fprintf(file,"$\omega^2$\n");
00072     fprintf(file,"$\alpha^2$\n");
00073     fprintf(file,"%lf %lf\n",start[0],start[1]);
00074     fprintf(file,"%lf %lf\n",end[0],end[1]);
00075     fprintf(file,"%d %d\n",nstep[0], nstep[1]);
00076     fclose(file);
00077
00078     for (int i = 0; i < nstep[0]; ++i)
00079     {
00080         free(stability[i]);
00081         free(largest_multiplier_abs[i]);
00082     }
00083
00084     free(stability);
00085     free(largest_multiplier_abs);
00086     return 0;
00087 }

```

3.11 src/hill_meissner_damped1.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Macros

- `#define DELTA 0.0456`

Functions

- void **print_int_matrix_to_file_csv** (int n1, int n2, int **m, FILE *file)
- void **print_double_matrix_to_file_csv** (int n1, int n2, double **m, FILE *file)
- void **hill_meissner** (double t, gsl_matrix *A_val, void *param)
- int **main** ()

3.12 hill_meissner_damped1.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 #define DELTA 0.0456
00008
00009 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00010 {
00011     for (int i = 0; i < n1; ++i)
00012     {
00013         for (int j = 0; j < n2-1; ++j)
00014         {
00015             fprintf(file, "%d", m[i][j]);
00016         }
00017         fprintf(file, "%d\n", m[i][n2-1]);
00018     }
00019 }
00020
00021 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00022 {
00023     for (int i = 0; i < n1; ++i)
00024     {
00025         for (int j = 0; j < n2-1; ++j)
00026         {
00027             fprintf(file, "%lf", m[i][j]);
00028         }
00029         fprintf(file, "%lf\n", m[i][n2-1]);
00030     }
00031 }
00032
00033
00034 void hill_meissner(double t, gsl_matrix* A_val, void* param)
00035 {
00036     double* par_temp = (double*) param;
00037     gsl_matrix_set_zero(A_val);
00038     gsl_matrix_set(A_val, 0, 1, 1.);
00039     gsl_matrix_set(A_val, 1, 0, -(par_temp[1] + par_temp[0]*GSL_SIGN(t-M_PI)));
00040     gsl_matrix_set(A_val, 1, 1, -DELTA);
00041 }
00042
00043 int main()
00044 {
00045     int n = 2;
00046     double T = 2*M_PI;
00047
00048     double start[2] = {12., 2.};
00049     double end[2] = {0., 10.};
00050     int nstep[2] = {320, 320};
00051     int** stability = (int**) malloc(nstep[0]*sizeof(int*));
00052     gsl_complex** largest_multiplier = NULL;
00053     double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));
00054
00055     for (int i = 0; i < nstep[0]; ++i)
00056     {
00057         stability[i] = (int*) malloc(nstep[1]*sizeof(int));
00058         largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
00059     }
00060
00061
00062     floquet_get_stability_array_real_double_param_general(n, hill_meissner, T, start, end, nstep,
00063     stability, largest_multiplier, largest_multiplier_abs);
00064
00065     FILE* file = fopen("stability.csv", "w");
00066     print_int_matrix_to_file_csv(nstep[0], nstep[1], stability, file);
00067     fclose(file);
00068
00069     file = fopen("largest_multiplier_abs.csv", "w");
00070     print_double_matrix_to_file_csv(nstep[0], nstep[1], largest_multiplier_abs, file);
00071     fclose(file);

```

```

00071
00072     file = fopen("extraparams.txt","w");
00073     fprintf(file,"Damped Hill-Meissner Stability Plot ($\\delta=0.0456$)\n");
00074     fprintf(file,"$\\omega^2$\n");
00075     fprintf(file,"$\\alpha^2$\n");
00076     fprintf(file,"%lf %lf\n",start[0],start[1]);
00077     fprintf(file,"%lf %lf\n",end[0],end[1]);
00078     fprintf(file,"%d %d\n",nstep[0], nstep[1]);
00079     fclose(file);
00080
00081     for (int i = 0; i < nstep[0]; ++i)
00082     {
00083         free(stability[i]);
00084         free(largest_multiplier_abs[i]);
00085     }
00086
00087     free(stability);
00088     free(largest_multiplier_abs);
00089     return 0;
00090 }

```

3.13 src/hill_meissner_damped2.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Macros

- `#define DELTA 0.0465`

Functions

- void `print_int_matrix_to_file_csv` (int n1, int n2, int **m, FILE *file)
- void `print_double_matrix_to_file_csv` (int n1, int n2, double **m, FILE *file)
- void `hill_meissner` (double t, gsl_matrix *A_val, void *param)
- int `main` ()

3.14 hill_meissner_damped2.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 #define DELTA 0.0465
00008
00009 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00010 {
00011     for (int i = 0; i < n1; ++i)
00012     {
00013         for (int j = 0; j < n2-1; ++j)
00014         {
00015             fprintf(file,"%d,",m[i][j]);
00016         }
00017         fprintf(file,"%d\n",m[i][n2-1]);
00018     }
00019 }
00020
00021 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00022 {
00023     for (int i = 0; i < n1; ++i)

```



```

00024     {
00025         for (int j = 0; j < n2-1; ++j)
00026         {
00027             fprintf(file, "%lf", m[i][j]);
00028         }
00029         fprintf(file, "%lf\n", m[i][n2-1]);
00030     }
00031 }
00032
00033
00034 void hill_meissner(double t, gsl_matrix* A_val, void* param)
00035 {
00036     double* par_temp = (double*) param;
00037     gsl_matrix_set_zero(A_val);
00038     gsl_matrix_set(A_val, 0, 1, 1.);
00039     gsl_matrix_set(A_val, 1, 0, -(par_temp[1] + par_temp[0]*GSL_SIGN(t-M_PI)));
00040     gsl_matrix_set(A_val, 1, 1, -DELTA);
00041 }
00042
00043 int main()
00044 {
00045     int n = 2;
00046     double T = 2*M_PI;
00047
00048     double start[2] = {12., 2.};
00049     double end[2] = {0., 10.};
00050     int nstep[2] = {320, 320};
00051     int** stability = (int**) malloc(nstep[0]*sizeof(int*));
00052     gsl_complex** largest_multiplier = NULL;
00053     double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));
00054
00055     for (int i = 0; i < nstep[0]; ++i)
00056     {
00057         stability[i] = (int*) malloc(nstep[1]*sizeof(int));
00058         largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
00059     }
00060
00061
00062     floquet_get_stability_array_real_double_param_general(n, hill_meissner, T, start, end, nstep,
00063 stability, largest_multiplier, largest_multiplier_abs);
00064
00065     FILE* file = fopen("stability.csv", "w");
00066     print_int_matrix_to_file_csv(nstep[0], nstep[1], stability, file);
00067     fclose(file);
00068
00069     file = fopen("largest_multiplier_abs.csv", "w");
00070     print_double_matrix_to_file_csv(nstep[0], nstep[1], largest_multiplier_abs, file);
00071     fclose(file);
00072
00073     file = fopen("extraparams.txt", "w");
00074     fprintf(file, "Damped Hill-Meissner Stability Plot ($\\delta=0.0456$)\n");
00075     fprintf(file, "$\\omega^2$\n");
00076     fprintf(file, "$\\alpha^2$\n");
00077     fprintf(file, "%lf %lf\n", start[0], start[1]);
00078     fprintf(file, "%lf %lf\n", end[0], end[1]);
00079     fprintf(file, "%d %d\n", nstep[0], nstep[1]);
00080     fclose(file);
00081
00082     for (int i = 0; i < nstep[0]; ++i)
00083     {
00084         free(stability[i]);
00085         free(largest_multiplier_abs[i]);
00086     }
00087
00088     free(stability);
00089     free(largest_multiplier_abs);
00090     return 0;

```

3.15 src/mathieu_damped_k_0_1.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Macros

- `#define K_DAMP 0.1`

Functions

- `void print_int_matrix_to_file_csv (int n1, int n2, int **m, FILE *file)`
- `void print_double_matrix_to_file_csv (int n1, int n2, double **m, FILE *file)`
- `void mathieu_undamped (double t, gsl_matrix *A_val, void *param)`
- `void mathieu_damped_fixed_k (double t, gsl_matrix *A_val, void *param)`
- `int main ()`

3.16 mathieu_damped_k_0_1.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 #define K_DAMP 0.1
00008
00009 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00010 {
00011     for (int i = 0; i < n1; ++i)
00012     {
00013         for (int j = 0; j < n2-1; ++j)
00014         {
00015             fprintf(file,"%d",m[i][j]);
00016         }
00017         fprintf(file,"%d\n",m[i][n2-1]);
00018     }
00019 }
00020
00021 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00022 {
00023     for (int i = 0; i < n1; ++i)
00024     {
00025         for (int j = 0; j < n2-1; ++j)
00026         {
00027             fprintf(file,"%lf",m[i][j]);
00028         }
00029         fprintf(file,"%lf\n",m[i][n2-1]);
00030     }
00031 }
00032
00033
00034 void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
00035 {
00036     double* par_temp = (double*) param;
00037     gsl_matrix_set_zero(A_val);
00038     gsl_matrix_set(A_val,0,1,1.);
00039     gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
00040 }
00041
00042 void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
00043 {
00044     double* par_temp = (double*) param;
00045     gsl_matrix_set_zero(A_val);
00046     gsl_matrix_set(A_val,0,1,1.);
00047     gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
00048     gsl_matrix_set(A_val,1,1,-(K_DAMP));
00049 }
00050
00051 int main()
00052 {
00053     int n = 2;
00054     double T = M_PI;
00055
00056     double start[2] = {60.,-5.};
00057     double end[2] = {0., 20.};
00058     int nstep[2] = {320,320};
00059     int** stability = (int**) malloc(nstep[0]*sizeof(int*));
00060     gsl_complex** largest_multiplier = NULL;
00061     double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));
00062

```

```

00063     for (int i = 0; i < nstep[0]; ++i)
00064     {
00065         stability[i] = (int*) malloc(nstep[1]*sizeof(int));
00066         largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
00067     }
00068
00069
00070     floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
nstep, stability, largest_multiplier, largest_multiplier_abs);
00071
00072     FILE* file = fopen("stability.csv","w");
00073     print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
00074     fclose(file);
00075
00076     file = fopen("largest_multiplier_abs.csv","w");
00077     print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
00078     fclose(file);
00079
00080     file = fopen("extraparams.txt","w");
00081     fprintf(file,"Damped Mathieu Stability Plot ($k=0.1$)\n");
00082     fprintf(file,"$\delta$\n");
00083     fprintf(file,"$\epsilon$\n");
00084     fprintf(file,"%lf %lf\n",start[0],start[1]);
00085     fprintf(file,"%lf %lf\n",end[0],end[1]);
00086     fprintf(file,"%d %d\n",nstep[0], nstep[1]);
00087     fclose(file);
00088
00089     for (int i = 0; i < nstep[0]; ++i)
00090     {
00091         free(stability[i]);
00092         free(largest_multiplier_abs[i]);
00093     }
00094
00095     free(stability);
00096     free(largest_multiplier_abs);
00097     return 0;
00098 }

```

3.17 src/mathieu_damped_k_0_5.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Macros

- `#define K_DAMP 0.5`

Functions

- void `print_int_matrix_to_file_csv` (int n1, int n2, int **m, FILE *file)
- void `print_double_matrix_to_file_csv` (int n1, int n2, double **m, FILE *file)
- void `mathieu_undamped` (double t, gsl_matrix *A_val, void *param)
- void `mathieu_damped_fixed_k` (double t, gsl_matrix *A_val, void *param)
- int `main` ()

3.18 mathieu_damped_k_0_5.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 #define K_DAMP 0.5
00008
00009 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00010 {
00011     for (int i = 0; i < n1; ++i)
00012     {
00013         for (int j = 0; j < n2-1; ++j)
00014         {
00015             fprintf(file, "%d, ", m[i][j]);
00016         }
00017         fprintf(file, "%d\n", m[i][n2-1]);
00018     }
00019 }
00020
00021 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00022 {
00023     for (int i = 0; i < n1; ++i)
00024     {
00025         for (int j = 0; j < n2-1; ++j)
00026         {
00027             fprintf(file, "%lf, ", m[i][j]);
00028         }
00029         fprintf(file, "%lf\n", m[i][n2-1]);
00030     }
00031 }
00032
00033
00034 void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
00035 {
00036     double* par_temp = (double*) param;
00037     gsl_matrix_set_zero(A_val);
00038     gsl_matrix_set(A_val, 0, 1, 1.);
00039     gsl_matrix_set(A_val, 1, 0, -(par_temp[1] + par_temp[0]*cos(2.*t)));
00040 }
00041
00042 void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
00043 {
00044     double* par_temp = (double*) param;
00045     gsl_matrix_set_zero(A_val);
00046     gsl_matrix_set(A_val, 0, 1, 1.);
00047     gsl_matrix_set(A_val, 1, 0, -(par_temp[1] + par_temp[0]*cos(2.*t)));
00048     gsl_matrix_set(A_val, 1, 1, -(K_DAMP));
00049 }
00050
00051 int main()
00052 {
00053     int n = 2;
00054     double T = M_PI;
00055
00056     double start[2] = {60., -5.};
00057     double end[2] = {0., 20.};
00058     int nstep[2] = {320, 320};
00059     int** stability = (int**) malloc(nstep[0]*sizeof(int));
00060     gsl_complex** largest_multiplier = NULL;
00061     double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double));
00062
00063     for (int i = 0; i < nstep[0]; ++i)
00064     {
00065         stability[i] = (int*) malloc(nstep[1]*sizeof(int));
00066         largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
00067     }
00068
00069
00070     floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
nstep, stability, largest_multiplier, largest_multiplier_abs);
00071
00072     FILE* file = fopen("stability.csv", "w");
00073     print_int_matrix_to_file_csv(nstep[0], nstep[1], stability, file);
00074     fclose(file);
00075
00076     file = fopen("largest_multiplier_abs.csv", "w");
00077     print_double_matrix_to_file_csv(nstep[0], nstep[1], largest_multiplier_abs, file);
00078     fclose(file);
00079
00080     file = fopen("extraparams.txt", "w");
00081     fprintf(file, "Damped Mathieu Stability Plot ($k=0.5$)\n");
00082     fprintf(file, "$\\delta$\n");
00083     fprintf(file, "$\\epsilon$\n");
00084     fprintf(file, "%lf %lf\n", start[0], start[1]);

```

```

00085     fprintf(file, "%lf %lf\n", end[0], end[1]);
00086     fprintf(file, "%d %d\n", nstep[0], nstep[1]);
00087     fclose(file);
00088
00089     for (int i = 0; i < nstep[0]; ++i)
00090     {
00091         free(stability[i]);
00092         free(largest_multiplier_abs[i]);
00093     }
00094
00095     free(stability);
00096     free(largest_multiplier_abs);
00097     return 0;
00098 }

```

3.19 src/mathieu_damped_k_1.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Macros

- `#define K_DAMP 1.`

Functions

- `void print_int_matrix_to_file_csv(int n1, int n2, int **m, FILE *file)`
- `void print_double_matrix_to_file_csv(int n1, int n2, double **m, FILE *file)`
- `void mathieu_undamped(double t, gsl_matrix *A_val, void *param)`
- `void mathieu_damped_fixed_k(double t, gsl_matrix *A_val, void *param)`
- `int main ()`

3.20 mathieu_damped_k_1.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 #define K_DAMP 1.
00008
00009 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00010 {
00011     for (int i = 0; i < n1; ++i)
00012     {
00013         for (int j = 0; j < n2-1; ++j)
00014         {
00015             fprintf(file, "%d,", m[i][j]);
00016         }
00017         fprintf(file, "%d\n", m[i][n2-1]);
00018     }
00019 }
00020
00021 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00022 {
00023     for (int i = 0; i < n1; ++i)
00024     {
00025         for (int j = 0; j < n2-1; ++j)
00026         {
00027             fprintf(file, "%lf,", m[i][j]);
00028         }

```

```

00029         fprintf(file, "%lf\n", m[i][n2-1]);
00030     }
00031 }
00032
00033
00034 void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
00035 {
00036     double* par_temp = (double*) param;
00037     gsl_matrix_set_zero(A_val);
00038     gsl_matrix_set(A_val, 0, 1, 1.);
00039     gsl_matrix_set(A_val, 1, 0, -(par_temp[1] + par_temp[0]*cos(2.*t)));
00040 }
00041
00042 void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
00043 {
00044     double* par_temp = (double*) param;
00045     gsl_matrix_set_zero(A_val);
00046     gsl_matrix_set(A_val, 0, 1, 1.);
00047     gsl_matrix_set(A_val, 1, 0, -(par_temp[1] + par_temp[0]*cos(2.*t)));
00048     gsl_matrix_set(A_val, 1, 1, -(K_DAMP));
00049 }
00050
00051 int main()
00052 {
00053     int n = 2;
00054     double T = M_PI;
00055
00056     double start[2] = {60., -5.};
00057     double end[2] = {0., 20.};
00058     int nstep[2] = {320, 320};
00059     int** stability = (int**) malloc(nstep[0]*sizeof(int*));
00060     gsl_complex** largest_multiplier = NULL;
00061     double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));
00062
00063     for (int i = 0; i < nstep[0]; ++i)
00064     {
00065         stability[i] = (int*) malloc(nstep[1]*sizeof(int));
00066         largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
00067     }
00068
00069     floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
00070 nstep, stability, largest_multiplier, largest_multiplier_abs);
00071
00072     FILE* file = fopen("stability.csv", "w");
00073     print_int_matrix_to_file_csv(nstep[0], nstep[1], stability, file);
00074     fclose(file);
00075
00076     file = fopen("largest_multiplier_abs.csv", "w");
00077     print_double_matrix_to_file_csv(nstep[0], nstep[1], largest_multiplier_abs, file);
00078     fclose(file);
00079
00080     file = fopen("extraparams.txt", "w");
00081     fprintf(file, "Damped Mathieu Stability Plot ($k=1$)\n");
00082     fprintf(file, "$\\delta$\n");
00083     fprintf(file, "$\\epsilon$\n");
00084     fprintf(file, "%lf %lf\n", start[0], start[1]);
00085     fprintf(file, "%lf %lf\n", end[0], end[1]);
00086     fprintf(file, "%d %d\n", nstep[0], nstep[1]);
00087     fclose(file);
00088
00089     for (int i = 0; i < nstep[0]; ++i)
00090     {
00091         free(stability[i]);
00092         free(largest_multiplier_abs[i]);
00093     }
00094
00095     free(stability);
00096     free(largest_multiplier_abs);
00097     return 0;
00098 }

```

3.21 src/mathieu_damped_k_10.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Macros

- `#define K_DAMP 10.`

Functions

- `void print_int_matrix_to_file_csv (int n1, int n2, int **m, FILE *file)`
- `void print_double_matrix_to_file_csv (int n1, int n2, double **m, FILE *file)`
- `void mathieu_undamped (double t, gsl_matrix *A_val, void *param)`
- `void mathieu_damped_fixed_k (double t, gsl_matrix *A_val, void *param)`
- `int main ()`

3.22 mathieu_damped_k_10.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 #define K_DAMP 10.
00008
00009 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00010 {
00011     for (int i = 0; i < n1; ++i)
00012     {
00013         for (int j = 0; j < n2-1; ++j)
00014         {
00015             fprintf(file,"%d",m[i][j]);
00016         }
00017         fprintf(file,"%d\n",m[i][n2-1]);
00018     }
00019 }
00020
00021 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00022 {
00023     for (int i = 0; i < n1; ++i)
00024     {
00025         for (int j = 0; j < n2-1; ++j)
00026         {
00027             fprintf(file,"%lf",m[i][j]);
00028         }
00029         fprintf(file,"%lf\n",m[i][n2-1]);
00030     }
00031 }
00032
00033
00034 void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
00035 {
00036     double* par_temp = (double*) param;
00037     gsl_matrix_set_zero(A_val);
00038     gsl_matrix_set(A_val,0,1,1.);
00039     gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
00040 }
00041
00042 void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
00043 {
00044     double* par_temp = (double*) param;
00045     gsl_matrix_set_zero(A_val);
00046     gsl_matrix_set(A_val,0,1,1.);
00047     gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
00048     gsl_matrix_set(A_val,1,1,-(K_DAMP));
00049 }
00050
00051 int main()
00052 {
00053     int n = 2;
00054     double T = M_PI;
00055
00056     double start[2] = {60.,-5.};
00057     double end[2] = {0., 20.};
00058     int nstep[2] = {320,320};
00059     int** stability = (int**) malloc(nstep[0]*sizeof(int*));
00060     gsl_complex** largest_multiplier = NULL;
00061     double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));
00062

```

```

00063     for (int i = 0; i < nstep[0]; ++i)
00064     {
00065         stability[i] = (int*) malloc(nstep[1]*sizeof(int));
00066         largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
00067     }
00068
00069
00070     floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
nstep, stability, largest_multiplier, largest_multiplier_abs);
00071
00072     FILE* file = fopen("stability.csv", "w");
00073     print_int_matrix_to_file_csv(nstep[0], nstep[1], stability, file);
00074     fclose(file);
00075
00076     file = fopen("largest_multiplier_abs.csv", "w");
00077     print_double_matrix_to_file_csv(nstep[0], nstep[1], largest_multiplier_abs, file);
00078     fclose(file);
00079
00080     file = fopen("extraparams.txt", "w");
00081     fprintf(file, "Damped Mathieu Stability Plot ($k=10$)\n");
00082     fprintf(file, "$\\delta$\n");
00083     fprintf(file, "$\\epsilon$\n");
00084     fprintf(file, "%lf %lf\n", start[0], start[1]);
00085     fprintf(file, "%lf %lf\n", end[0], end[1]);
00086     fprintf(file, "%d %d\n", nstep[0], nstep[1]);
00087     fclose(file);
00088
00089     for (int i = 0; i < nstep[0]; ++i)
00090     {
00091         free(stability[i]);
00092         free(largest_multiplier_abs[i]);
00093     }
00094
00095     free(stability);
00096     free(largest_multiplier_abs);
00097     return 0;
00098 }

```

3.23 src/mathieu_undamped.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Macros

- `#define K_DAMP 0.1`

Functions

- void `print_int_matrix_to_file_csv` (int n1, int n2, int **m, FILE *file)
- void `print_double_matrix_to_file_csv` (int n1, int n2, double **m, FILE *file)
- void `mathieu_undamped` (double t, gsl_matrix *A_val, void *param)
- void `mathieu_damped_fixed_k` (double t, gsl_matrix *A_val, void *param)
- int `main` ()

3.24 mathieu_undamped.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 #define K_DAMP 0.1
00008
00009 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00010 {
00011     for (int i = 0; i < n1; ++i)
00012     {
00013         for (int j = 0; j < n2-1; ++j)
00014         {
00015             fprintf(file, "%d, ", m[i][j]);
00016         }
00017         fprintf(file, "%d\n", m[i][n2-1]);
00018     }
00019 }
00020
00021 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00022 {
00023     for (int i = 0; i < n1; ++i)
00024     {
00025         for (int j = 0; j < n2-1; ++j)
00026         {
00027             fprintf(file, "%lf, ", m[i][j]);
00028         }
00029         fprintf(file, "%lf\n", m[i][n2-1]);
00030     }
00031 }
00032
00033
00034 void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
00035 {
00036     double* par_temp = (double*) param;
00037     gsl_matrix_set_zero(A_val);
00038     gsl_matrix_set(A_val, 0, 1, 1.);
00039     gsl_matrix_set(A_val, 1, 0, -(par_temp[1] + par_temp[0]*cos(2.*t)));
00040 }
00041
00042 void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
00043 {
00044     double* par_temp = (double*) param;
00045     gsl_matrix_set_zero(A_val);
00046     gsl_matrix_set(A_val, 0, 1, 1.);
00047     gsl_matrix_set(A_val, 1, 0, -(par_temp[1] + par_temp[0]*cos(2.*t)));
00048     gsl_matrix_set(A_val, 1, 1, -(K_DAMP));
00049 }
00050
00051 int main()
00052 {
00053     int n = 2;
00054     double T = M_PI;
00055
00056     double start[2] = {60., -5.};
00057     double end[2] = {0., 20.};
00058     int nstep[2] = {320, 320};
00059     int** stability = (int**) malloc(nstep[0]*sizeof(int*));
00060     gsl_complex** largest_multiplier = NULL;
00061     double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));
00062
00063     for (int i = 0; i < nstep[0]; ++i)
00064     {
00065         stability[i] = (int*) malloc(nstep[1]*sizeof(int));
00066         largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
00067     }
00068
00069
00070     floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
nstep, stability, largest_multiplier, largest_multiplier_abs);
00071
00072     FILE* file = fopen("stability.csv", "w");
00073     print_int_matrix_to_file_csv(nstep[0], nstep[1], stability, file);
00074     fclose(file);
00075
00076     file = fopen("largest_multiplier_abs.csv", "w");
00077     print_double_matrix_to_file_csv(nstep[0], nstep[1], largest_multiplier_abs, file);
00078     fclose(file);
00079
00080     file = fopen("extraparams.txt", "w");
00081     fprintf(file, "Damped Mathieu Stability Plot ($k=0.1$)\n");
00082     fprintf(file, "$\\delta$\n");
00083     fprintf(file, "$\\epsilon$\n");
00084     fprintf(file, "%lf %lf\n", start[0], start[1]);

```

```

00085     fprintf(file,"%lf %lf\n",end[0],end[1]);
00086     fprintf(file,"%d %d\n",nstep[0], nstep[1]);
00087     fclose(file);
00088
00089     for (int i = 0; i < nstep[0]; ++i)
00090     {
00091         free(stability[i]);
00092         free(largest_multiplier_abs[i]);
00093     }
00094
00095     free(stability);
00096     free(largest_multiplier_abs);
00097     return 0;
00098 }

```

3.25 src/population_dynamics.c File Reference

```

#include <stdio.h>
#include <gsl/gsl_math.h>
#include <math.h>
#include "floquet.h"

```

Functions

- void **print_int_matrix_to_file_csv** (int n1, int n2, int **m, FILE *file)
- void **print_double_matrix_to_file_csv** (int n1, int n2, double **m, FILE *file)
- void **fitness_periodic** (double t, gsl_matrix *A_val, void *param)
- int **main** ()

3.26 population_dynamics.c

```

00001
00002 #include <stdio.h>
00003 #include <gsl/gsl_math.h>
00004 #include <math.h>
00005 #include "floquet.h"
00006
00007 void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
00008 {
00009     for (int i = 0; i < n1; ++i)
00010     {
00011         for (int j = 0; j < n2-1; ++j)
00012         {
00013             fprintf(file,"%d,",m[i][j]);
00014         }
00015         fprintf(file,"%d\n",m[i][n2-1]);
00016     }
00017 }
00018
00019 void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
00020 {
00021     for (int i = 0; i < n1; ++i)
00022     {
00023         for (int j = 0; j < n2-1; ++j)
00024         {
00025             fprintf(file,"%lf,",m[i][j]);
00026         }
00027         fprintf(file,"%lf\n",m[i][n2-1]);
00028     }
00029 }
00030
00031
00032 void fitness_periodic(double t, gsl_matrix* A_val, void* param)
00033 {
00034     double d = *((double*) param);
00035     double sint = sin(2.*M_PI*t);
00036     gsl_matrix_set(A_val,0,0,sint-d);
00037     gsl_matrix_set(A_val,0,1,d);

```

```

00038     gsl_matrix_set(A_val,1,0,d);
00039     gsl_matrix_set(A_val,1,1,-sint-d);
00040 }
00041
00042 int main()
00043 {
00044     int n = 2;
00045     double T = M_PI;
00046
00047     double start = 0.;
00048     double end = 100.;
00049     int nstep = 1024;
00050     int* stability = (int*) malloc(nstep*sizeof(int));
00051     gsl_complex* largest_multiplier = NULL;
00052     double* largest_multiplier_abs = (double*) malloc(nstep*sizeof(double));
00053
00054     floquet_get_stability_array_real_single_param_general(n, fitness_periodic, T, start, end, nstep,
00055 stability, largest_multiplier, largest_multiplier_abs);
00056
00056     FILE* file = fopen("stability.csv","w");
00057     for (int i = 0; i < nstep-1; ++i)
00058     {
00059         fprintf(file,"%d",stability[i]);
00060     }
00061     fprintf(file,"%d",stability[nstep-1]);
00062     fclose(file);
00063
00064     file = fopen("largest_multiplier_abs.csv","w");
00065     for (int i = 0; i < nstep-1; ++i)
00066     {
00067         fprintf(file,"%lf",largest_multiplier_abs[i]);
00068     }
00069     fprintf(file,"%lf",largest_multiplier_abs[nstep-1]);
00070     fclose(file);
00071
00072     file = fopen("extraparams.txt","w");
00073     fprintf(file," Dominant Floquet Multiplier as a function of Dispersal rate $d$\n");
00074     fprintf(file,"$d$\n");
00075     fprintf(file,"$\max(|\rho|)$\n");
00076     fprintf(file,"%lf\n",start);
00077     fprintf(file,"%lf\n",end);
00078     fprintf(file,"%d\n",nstep);
00079     fclose(file);
00080
00081     free(stability);
00082     free(largest_multiplier_abs);
00083     return 0;
00084 }

```


Index

bulsto_final_matrix_floquet_type_real
diffEqSolvers.c, [21](#)
diffEqSolvers.h, [6](#)

diffEqSolvers.c
bulsto_final_matrix_floquet_type_real, [21](#)
rk4_adaptive_final_matrix_floquet_type_real, [22](#)
rk4_fixed_final_matrix_floquet_type_complex, [24](#)
rk4_fixed_final_matrix_floquet_type_real, [26](#)
rk4_fixed_final_vector_real, [27](#)

diffEqSolvers.h
bulsto_final_matrix_floquet_type_real, [6](#)
rk4_adaptive_final_matrix_floquet_type_real, [7](#)
rk4_fixed_final_matrix_floquet_type_complex, [9](#)
rk4_fixed_final_matrix_floquet_type_real, [11](#)
rk4_fixed_final_vector_real, [13](#)

diffEqSolvers.h, [11](#)
rk4_fixed_final_vector_real
diffEqSolvers.c, [27](#)
diffEqSolvers.h, [13](#)

src/diffEqSolvers.c, [20](#), [28](#)
src/floquet.c, [33](#), [39](#)
src/hill_meissner.c, [41](#)
src/hill_meissner_damped1.c, [42](#), [43](#)
src/hill_meissner_damped2.c, [44](#)
src/mathieu_damped_k_0_1.c, [45](#), [46](#)
src/mathieu_damped_k_0_5.c, [47](#), [48](#)
src/mathieu_damped_k_1.c, [49](#)
src/mathieu_damped_k_10.c, [50](#), [51](#)
src/mathieu_undamped.c, [52](#), [53](#)
src/population_dynamics.c, [54](#)

floquet.c
floquet_get_stability_array_real_double_param_general,
[34](#)
floquet_get_stability_array_real_single_param_general,
[35](#)
floquet_get_stability_reals_general, [36](#)

floquet.h
floquet_get_stability_array_real_double_param_general,
[15](#)
floquet_get_stability_array_real_single_param_general,
[16](#)
floquet_get_stability_reals_general, [17](#)

floquet_get_stability_array_real_double_param_general
floquet.c, [34](#)
floquet.h, [15](#)

floquet_get_stability_array_real_single_param_general
floquet.c, [35](#)
floquet.h, [16](#)

floquet_get_stability_reals_general
floquet.c, [36](#)
floquet.h, [17](#)

include/diffEqSolvers.h, [5](#), [13](#)
include/floquet.h, [14](#), [20](#)

rk4_adaptive_final_matrix_floquet_type_real
diffEqSolvers.c, [22](#)
diffEqSolvers.h, [7](#)

rk4_fixed_final_matrix_floquet_type_complex
diffEqSolvers.c, [24](#)
diffEqSolvers.h, [9](#)

rk4_fixed_final_matrix_floquet_type_real
diffEqSolvers.c, [26](#)