Floquet Toolkit

Generated by Doxygen 1.8.17

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:

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

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
delta	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.

```
{
    // This program only initializes and provides and frees temp variables
    int ndim = x_i->size1;
    gsl_matrix** R1 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
    gsl_matrix** R2 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));

    for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
    {
        R1[i] = gsl_matrix_alloc(ndim,ndim);
        R2[i] = gsl_matrix_alloc(ndim,ndim);
    }

    gsl_matrix* y = gsl_matrix_alloc(ndim,ndim);
    gsl_matrix* eval = gsl_matrix_alloc(ndim,ndim);
    gsl_matrix* epsilon = gsl_matrix_calloc(ndim,ndim);

    __bulsto_final_matrix_floquet_type_real_runner(0, x_i, t_i, H, delta, A, x_f, params, y, eval, R1,
  R2, epsilon);
    //printf("%e %e %e\n",error, H, error/H);
    for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
    {
        gsl_matrix_free(R1[i]);
        gsl_matrix_free(R2[i]);
    }
    free(R1);
    free(R2);
    gsl_matrix_free(y);
    gsl_matrix_free(eval);
    gsl_matrix_free(epsilon);
}
```

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

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
delta	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.

```
{
    double h_min = H/RK4_MAX_SLICES;
    int nr = x_i->size1;
    int nc = x_i->size2;
    gsl_matrix* k[4];
    gsl_matrix* k_in[4];
    gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
    for (int i = 0; i < 4; ++i)
    {
        k[i] = gsl_matrix_calloc(nr,nc);
        k_in[i] = gsl_matrix_calloc(nr,nc);
    }

    double t = t_i;
    double t_f = t_i + H;
    double h = H/10.; // Initial h. This is a bit conservative, but will be refined further.
    gsl_matrix_memcpy(x_f,x_i);

    gsl_matrix* x1 = gsl_matrix_alloc(nr,nc);
    gsl_matrix* x2 = gsl_matrix_alloc(nr,nc);

    while (t<t_f)
    {
        // Evaluate x1
        gsl_matrix_memcpy(x1,x_f);

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x_f);

        A(t+0.5*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x_f);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x1,k_in[3]);

        t += h;

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x1);

        A(t+0.5*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x1);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x1);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x1,k_in[3]);

        t -= h;

        // Evaluate x2
        gsl_matrix_memcpy(x2,x_f);

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x_f);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x_f);

        A(t+2*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x2,k_in[3]);

        // Evaluate Error
        gsl_matrix_sub(x2,x1);
        double rho_to_the_fourth = (30.*h*delta)/GSL_MAX(gsl_matrix_max(x2), -1.*gsl_matrix_min(x2));
        rho_to_the_fourth = pow(rho_to_the_fourth,0.25);
        if (rho_to_the_fourth>1)
        {
            gsl_matrix_memcpy(x_f,x1);
            gsl_matrix_scale(x2, -1./15.);
            gsl_matrix_add(x_f,x2);
            t += 2*h;
            h = h*(GSL_MIN(rho_to_the_fourth, RK4_MAX_SCALE));
            h = GSL_MIN(0.5*(t_f-t),h);
        }
        else if(h > h_min)
        {
            h = h*(GSL_MAX(rho_to_the_fourth, RK4_MIN_SCALE));
        }
        else
        {
            gsl_matrix_memcpy(x_f,x1);
            gsl_matrix_scale(x2, -1./15.);
            gsl_matrix_add(x_f,x2);
            t += 2*h;
            h = h_min;
        }
    }

    gsl_matrix_free(A_val);

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_free(k[i]);
        gsl_matrix_free(k_in[i]);
    }

    gsl_matrix_free(x1);
    gsl_matrix_free(x2);
}
```

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
evol_func	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
params	Parameters to be passed to A(t)

Definition at line 102 of file diffEqSolvers.c.

```
{
    int nr = x_i->size1;
    int nc = x_i->size2;
    gsl_matrix_complex* k[4];
    gsl_matrix_complex* k_in[4];
    gsl_matrix_complex* A_val = gsl_matrix_complex_alloc(nr,nc);
    for (int i = 0; i < 4; ++i)
    {
        k[i] = gsl_matrix_complex_alloc(nr,nc);
        k_in[i] = gsl_matrix_complex_alloc(nr,nc);
    }

    gsl_complex h = gsl_complex_rect(h_,0.);
    gsl_complex zero = gsl_complex_rect(0.,0.);
    gsl_complex half = gsl_complex_rect(0.5,0.);
    gsl_complex one_sixth = gsl_complex_rect(1./6.,0.);

    double t = t_i;
    double t_f = t_i + H;
    gsl_matrix_complex_memcpy(x_f,x_i);
    while (t<t_f)
    {
        A(t,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, zero, k[0]);

        gsl_matrix_complex_memcpy(k_in[0], k[0]);
        gsl_matrix_complex_scale(k_in[0],half);
        gsl_matrix_complex_add(k_in[0],x_f);

        A(t+0.5*h_,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], zero, k[1]);

        gsl_matrix_complex_memcpy(k_in[1], k[1]);
        gsl_matrix_complex_scale(k_in[1],half);
        gsl_matrix_complex_add(k_in[1],x_f);

        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], zero, k[2]);

        gsl_matrix_complex_memcpy(k_in[2], k[2]);
        gsl_matrix_complex_add(k_in[2],x_f);

        A(t+h_,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], zero, k[3]);

        gsl_matrix_complex_memcpy(k_in[3],k[3]);
        gsl_matrix_complex_add(k_in[3],k[2]);
        gsl_matrix_complex_add(k_in[3],k[2]);
        gsl_matrix_complex_add(k_in[3],k[1]);
        gsl_matrix_complex_add(k_in[3],k[1]);
        gsl_matrix_complex_add(k_in[3],k[0]);
        gsl_matrix_complex_scale(k_in[3], one_sixth);
        gsl_matrix_complex_add(x_f,k_in[3]);
        t += h_;
    }

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_complex_free(k[i]);
        gsl_matrix_complex_free(k_in[i]);
    }
}
```

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
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 44 of file diffEqSolvers.c.

```
{
    int nr = x_i->size1;
    int nc = x_i->size2;
    gsl_matrix* k[4];
    gsl_matrix* k_in[4];
    gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
    for (int i = 0; i < 4; ++i)
    {
        k[i] = gsl_matrix_calloc(nr,nc);
        k_in[i] = gsl_matrix_calloc(nr,nc);
    }

    double t = t_i;
    double t_f = t_i + H;
    gsl_matrix_memcpy(x_f,x_i);
    while (t<t_f)
    {
        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x_f);

        A(t+0.5*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x_f);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x_f,k_in[3]);
        t += h;
    }

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_free(k[i]);
        gsl_matrix_free(k_in[i]);
    }
}
```

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

ndim	Dimensionality of x
x_i	Initial Vector $x_i \in \mathbb{R}^n$
t_i	Time when x_i is specified
H	Interval after which final x is required
h	Step size
evol_func	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)
x_f	Array to store the final x into. This should be preallocated
params	Parameters to be passed to evol_func

Definition at line 31 of file diffEqSolvers.c.

```
{
    double t = t_i;
    double t_f = t_i + H;

    cblas_dcopy(ndim,x_i,sizeof(x_i[0]),x_f,sizeof(x_f[0]));
    while(t<t_f)
    {
        __rk4_single_vector(ndim,x_f,t,h,evol_func,x_f, params);
        t += h;
    }
}
```

3.2 diffEqSolvers.h

```
#ifndef DIFFEQ_SOLVERS
```

```
#define DIFFEQ_SOLVERS

#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>

#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
#define BULSTO_MAX_LAYERS 32


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);

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);

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);

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);

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);

#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

n	Number of elements of vector that $A(t)$ operates on
A	$A(t)$ 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.
T	Period of the evolution function $A(t)$ s.t. $A(t+T) = A(t)\ \forall t \in \mathbb{R}$
start	Starting values of the parameters
end	Ending values of the parameters
nstep	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.
largest_multiplier	Matrix to store the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.
largest_multiplier_abs	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.

```
{
    gsl_complex** mult_temp;
    double** mult_abs_temp;

    if (largest_multiplier)
    {
        mult_temp = largest_multiplier;
    }
    else
    {
        mult_temp = (gsl_complex**) malloc(nstep[0]*sizeof(gsl_complex*));
        for (int i = 0; i < nstep[0]; ++i)
        {
            mult_temp[i] = (gsl_complex*) malloc(nstep[1]*sizeof(gsl_complex));
        }
    }

    if (largest_multiplier_abs)
    {
        mult_abs_temp = largest_multiplier_abs;
    }
    else
    {
        mult_abs_temp = (double**) malloc(nstep[0]*sizeof(double*));
        for (int i = 0; i < nstep[0]; ++i)
        {
            mult_abs_temp[i] = (double*) malloc(nstep[1]*sizeof(double));
        }
    }

    double step[2] = {(end[0]-start[0])/(nstep[0]-1), (end[1]-start[1])/(nstep[1]-1)};
    #pragma omp parallel for collapse(2) schedule(guided)
    for (int i = 0; i < nstep[0]; ++i)
    {
        for (int j = 0; j < nstep[1]; ++j)
        {
            double param[2] = {start[0] + step[0]*i, start[1] + step[1]*j };
            stability[i][j] =
       floquet_get_stability_reals_general(n,A,param,T,&mult_temp[i][j],&mult_abs_temp[i][j]);
        }
    }

    if(!largest_multiplier)
    {
        for (int i = 0; i < nstep[0]; ++i)
        {
            free(mult_temp[i]);
        }
        free(mult_temp);
    }

    if(!largest_multiplier_abs)
    {
        for (int i = 0; i < nstep[0]; ++i)
        {
            free(mult_abs_temp[i]);
        }
        free(mult_abs_temp);
    }
}
```

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

n	Number of elements of vector that $A(t)$ operates on

Parameters

A	$A(t)$ 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.
T	Period of the evolution function $A(t)$ s.t. $A(t+T) = A(t)\ \forall t \in \mathbb{R}$
start	Starting value of the parameter
end	Ending value of the parameter
nstep	Number of steps to take inclusive of the first and last values. Should be at least 2. Behaviour not defined otherwise.
largest_multiplier	Array to store the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.
largest_multiplier_abs	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.

```
{
    gsl_complex* mult_temp;
    double* mult_abs_temp;

    if (largest_multiplier)
    {
        mult_temp = largest_multiplier;
    }
    else
    {
        mult_temp = (gsl_complex*) malloc(nstep*sizeof(gsl_complex));
    }

    if (largest_multiplier_abs)
    {
        mult_abs_temp = largest_multiplier_abs;
    }
    else
    {
        mult_abs_temp = (double*) malloc(nstep*sizeof(double));
    }

    double step = (end-start)/(nstep-1);
    #pragma omp parallel for
    for (int i = 0; i < nstep; ++i)
    {
        double param = start + step*i;
        stability[i] = floquet_get_stability_reals_general(n,A,&param,T,mult_temp+i,mult_abs_temp+i);
    }

    if(!largest_multiplier)
    {
        free(mult_temp);
    }
    if(!largest_multiplier_abs)
    {
        free(mult_abs_temp);
    }
}
```

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

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.

```
{
    gsl_matrix* X = gsl_matrix_alloc(n,n);
    gsl_matrix_set_identity(X);

    gsl_matrix* B = gsl_matrix_alloc(n,n);
    bulsto_final_matrix_floquet_type_real(X, 0., T, ERR_TOL, A, B, params);

    gsl_vector_complex* eigenvals = gsl_vector_complex_alloc(n);

    gsl_eigen_nonsymm_workspace* w = gsl_eigen_nonsymm_alloc(n);

    int n_eigenvals_evaluated = n;

    int err_code = gsl_eigen_nonsymm(B,eigenvals,w);
    if(err_code)
    {
        n_eigenvals_evaluated = w->n_evals;
    }

    double mult_max_abs = -HUGE_VAL;
    gsl_complex mult_max;
    double ev_test;
    for (int i = 0; i < n_eigenvals_evaluated; ++i)
    {
        ev_test = gsl_complex_logabs(gsl_vector_complex_get(eigenvals,i));
        if (mult_max_abs < ev_test)
        {
            mult_max_abs = ev_test;
            mult_max = gsl_vector_complex_get(eigenvals,i);
        }
    }

    if (largest_multiplier != NULL)
    {
        *largest_multiplier = mult_max;
    }
    if (largest_multiplier_abs != NULL)
    {
        *largest_multiplier_abs = gsl_complex_abs(mult_max);
    }

    gsl_eigen_nonsymm_free(w);
    gsl_vector_complex_free(eigenvals);
    gsl_matrix_free(B);
    gsl_matrix_free(X);

    if (mult_max_abs > ERR_EIGEN_TOL)
    {
        return 1;
    }
    else if (mult_max_abs < (-ERR_EIGEN_TOL))
    {
        if (n_eigenvals_evaluated < n)
        {
            return 2;
        }
        return -1;
    }
    else
    {
        return 0;
    }
}
```

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

```
#ifndef FLOQUET
#define FLOQUET

#define ERR_TOL 1e-6
#define ERR_EIGEN_TOL 1e-5

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

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);

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);

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);

#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, 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
delta	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.

```
{
    // This program only initializes and provides and frees temp variables
    int ndim = x_i->size1;
    gsl_matrix** R1 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
    gsl_matrix** R2 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));

    for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
    {
        R1[i] = gsl_matrix_alloc(ndim,ndim);
        R2[i] = gsl_matrix_alloc(ndim,ndim);
    }

    gsl_matrix* y = gsl_matrix_alloc(ndim,ndim);
    gsl_matrix* eval = gsl_matrix_alloc(ndim,ndim);
    gsl_matrix* epsilon = gsl_matrix_calloc(ndim,ndim);

    __bulsto_final_matrix_floquet_type_real_runner(0, x_i, t_i, H, delta, A, x_f, params, y, eval, R1,
       R2, epsilon);
    //printf("%e %e %e\n",error, H, error/H);
    for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
    {
        gsl_matrix_free(R1[i]);
        gsl_matrix_free(R2[i]);
    }
    free(R1);
    free(R2);
    gsl_matrix_free(y);
    gsl_matrix_free(eval);
    gsl_matrix_free(epsilon);
}
```

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
delta	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.

```
{
    double h_min = H/RK4_MAX_SLICES;
    int nr = x_i->size1;
    int nc = x_i->size2;
    gsl_matrix* k[4];
    gsl_matrix* k_in[4];
    gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
    for (int i = 0; i < 4; ++i)
    {
        k[i] = gsl_matrix_calloc(nr,nc);
        k_in[i] = gsl_matrix_calloc(nr,nc);
    }

    double t = t_i;
    double t_f = t_i + H;
    double h = H/10.; // Initial h. This is a bit conservative, but will be refined further.
    gsl_matrix_memcpy(x_f,x_i);

    gsl_matrix* x1 = gsl_matrix_alloc(nr,nc);
    gsl_matrix* x2 = gsl_matrix_alloc(nr,nc);

    while (t<t_f)
    {
        // Evaluate x1
        gsl_matrix_memcpy(x1,x_f);

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x_f);

        A(t+0.5*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x_f);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x1,k_in[3]);

        t += h;

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x1);

        A(t+0.5*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x1);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x1);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x1,k_in[3]);

        t -= h;

        // Evaluate x2
        gsl_matrix_memcpy(x2,x_f);

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x_f);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x_f);

        A(t+2*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x2,k_in[3]);

        // Evaluate Error
        gsl_matrix_sub(x2,x1);
        double rho_to_the_fourth = (30.*h*delta)/GSL_MAX(gsl_matrix_max(x2), -1.*gsl_matrix_min(x2));
        rho_to_the_fourth = pow(rho_to_the_fourth,0.25);
        if (rho_to_the_fourth>1)
        {
            gsl_matrix_memcpy(x_f,x1);
            gsl_matrix_scale(x2, -1./15.);
            gsl_matrix_add(x_f,x2);
            t += 2*h;
            h = h*(GSL_MIN(rho_to_the_fourth, RK4_MAX_SCALE));
            h = GSL_MIN(0.5*(t_f-t),h);
        }
        else if(h > h_min)
        {
            h = h*(GSL_MAX(rho_to_the_fourth, RK4_MIN_SCALE));
        }
        else
        {
            gsl_matrix_memcpy(x_f,x1);
            gsl_matrix_scale(x2, -1./15.);
            gsl_matrix_add(x_f,x2);
            t += 2*h;
            h = h_min;
        }
    }

    gsl_matrix_free(A_val);

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_free(k[i]);
        gsl_matrix_free(k_in[i]);
    }

    gsl_matrix_free(x1);
    gsl_matrix_free(x2);
}
```

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
evol_func	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
params	Parameters to be passed to A(t)

Definition at line 102 of file diffEqSolvers.c.

```
{
    int nr = x_i->size1;
    int nc = x_i->size2;
    gsl_matrix_complex* k[4];
    gsl_matrix_complex* k_in[4];
    gsl_matrix_complex* A_val = gsl_matrix_complex_alloc(nr,nc);
    for (int i = 0; i < 4; ++i)
    {
        k[i] = gsl_matrix_complex_alloc(nr,nc);
        k_in[i] = gsl_matrix_complex_alloc(nr,nc);
    }

    gsl_complex h = gsl_complex_rect(h_,0.);
    gsl_complex zero = gsl_complex_rect(0.,0.);
    gsl_complex half = gsl_complex_rect(0.5,0.);
    gsl_complex one_sixth = gsl_complex_rect(1./6.,0.);

    double t = t_i;
    double t_f = t_i + H;
    gsl_matrix_complex_memcpy(x_f,x_i);
    while (t<t_f)
    {
        A(t,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, zero, k[0]);

        gsl_matrix_complex_memcpy(k_in[0], k[0]);
        gsl_matrix_complex_scale(k_in[0],half);
        gsl_matrix_complex_add(k_in[0],x_f);

        A(t+0.5*h_,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], zero, k[1]);

        gsl_matrix_complex_memcpy(k_in[1], k[1]);
        gsl_matrix_complex_scale(k_in[1],half);
        gsl_matrix_complex_add(k_in[1],x_f);

        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], zero, k[2]);

        gsl_matrix_complex_memcpy(k_in[2], k[2]);
        gsl_matrix_complex_add(k_in[2],x_f);

        A(t+h_,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], zero, k[3]);

        gsl_matrix_complex_memcpy(k_in[3],k[3]);
        gsl_matrix_complex_add(k_in[3],k[2]);
        gsl_matrix_complex_add(k_in[3],k[2]);
        gsl_matrix_complex_add(k_in[3],k[1]);
        gsl_matrix_complex_add(k_in[3],k[1]);
        gsl_matrix_complex_add(k_in[3],k[0]);
        gsl_matrix_complex_scale(k_in[3], one_sixth);
        gsl_matrix_complex_add(x_f,k_in[3]);
        t += h_;
    }

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_complex_free(k[i]);
        gsl_matrix_complex_free(k_in[i]);
    }
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
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 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]);
```

```
        gsl_matrix_add(k_in[2],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x_f,k_in[3]);
        t += h;
    }

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_free(k[i]);
        gsl_matrix_free(k_in[i]);
    }
}
```

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

ndim	Dimensionality of x
x_i	Initial Vector $x_i \in \mathbb{R}^n$
t_i	Time when x_i is specified
H	Interval after which final x is required
h	Step size
evol_func	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)
x_f	Array to store the final x into. This should be preallocated
params	Parameters to be passed to evol_func

Definition at line 31 of file diffEqSolvers.c.

```
{
    double t = t_i;
    double t_f = t_i + H;

    cblas_dcopy(ndim,x_i,sizeof(x_i[0]),x_f,sizeof(x_f[0]));
    while(t<t_f)
    {
        __rk4_single_vector(ndim,x_f,t,h,evol_func,x_f, params);
        t += h;
    }
}
```

3.6 diffEqSolvers.c

```c

#include "diffEqSolvers.h"

void __rk4_single_vector(int n, double* x, double t, double h, void (*evol_func)(double*, double, double*, void*), double* x_f, void* params)
{
    double k1[n], k2[n], k3[n], k4[n], k2_in[n], k3_in[n], k4_in[n];
    evol_func(x,t,k1,params);
    cblas_dscal(n,h,k1,sizeof(k1[0]));

    cblas_dcopy(n,x,sizeof(x[0]),k2_in,sizeof(k2_in[0]));
    cblas_daxpy(n,0.5,k1,sizeof(k1[0]),k2_in,sizeof(k2_in[0]));
    evol_func(k2_in,t+0.5*h,k2,params);
    cblas_dscal(n,h,k2,sizeof(k2[0]));

    cblas_dcopy(n,x,sizeof(x[0]),k3_in,sizeof(k3_in[0]));
    cblas_daxpy(n,0.5,k2,sizeof(k2[0]),k3_in,sizeof(k3_in[0]));
    evol_func(k3_in,t+0.5*h,k3,params);
    cblas_dscal(n,h,k3,sizeof(k3[0]));

    cblas_dcopy(n,x,sizeof(x[0]),k4_in,sizeof(k4_in[0]));
    cblas_daxpy(n,1.,k3,sizeof(k3[0]),k4_in,sizeof(k4_in[0]));
    evol_func(k4_in,t+h,k4,params);
    cblas_dscal(n,h,k4,sizeof(k4[0]));

    for (int i = 0; i < n; ++i)
    {
        x_f[i] = x[i] + (1./6.)*(k1[i] + 2*k2[i] + 2*k3[i] + k4[i]);
    }
}

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)
{
    double t = t_i;
    double t_f = t_i + H;

    cblas_dcopy(ndim,x_i,sizeof(x_i[0]),x_f,sizeof(x_f[0]));
    while(t<t_f)
    {
        __rk4_single_vector(ndim,x_f,t,h,evol_func,x_f, params);
        t += h;
    }
}

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)
{
    int nr = x_i->size1;
    int nc = x_i->size2;
    gsl_matrix* k[4];
    gsl_matrix* k_in[4];
    gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
    for (int i = 0; i < 4; ++i)
    {
        k[i] = gsl_matrix_calloc(nr,nc);
        k_in[i] = gsl_matrix_calloc(nr,nc);
    }

    double t = t_i;
    double t_f = t_i + H;
    gsl_matrix_memcpy(x_f,x_i);
    while (t<t_f)
    {
        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x_f);

        A(t+0.5*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x_f);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x_f,k_in[3]);
        t += h;
    }

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_free(k[i]);
        gsl_matrix_free(k_in[i]);
    }
}

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)
{
    int nr = x_i->size1;
    int nc = x_i->size2;
    gsl_matrix_complex* k[4];
    gsl_matrix_complex* k_in[4];
    gsl_matrix_complex* A_val = gsl_matrix_complex_alloc(nr,nc);
    for (int i = 0; i < 4; ++i)
    {
        k[i] = gsl_matrix_complex_alloc(nr,nc);
        k_in[i] = gsl_matrix_complex_alloc(nr,nc);
    }

    gsl_complex h = gsl_complex_rect(h_,0.);
    gsl_complex zero = gsl_complex_rect(0.,0.);
    gsl_complex half = gsl_complex_rect(0.5,0.);
    gsl_complex one_sixth = gsl_complex_rect(1./6.,0.);

    double t = t_i;
    double t_f = t_i + H;
    gsl_matrix_complex_memcpy(x_f,x_i);
    while (t<t_f)
    {
        A(t,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, zero, k[0]);

        gsl_matrix_complex_memcpy(k_in[0], k[0]);
        gsl_matrix_complex_scale(k_in[0],half);
        gsl_matrix_complex_add(k_in[0],x_f);

        A(t+0.5*h_,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], zero, k[1]);

        gsl_matrix_complex_memcpy(k_in[1], k[1]);
        gsl_matrix_complex_scale(k_in[1],half);
        gsl_matrix_complex_add(k_in[1],x_f);

        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], zero, k[2]);

        gsl_matrix_complex_memcpy(k_in[2], k[2]);
        gsl_matrix_complex_add(k_in[2],x_f);

        A(t+h_,A_val,params);
        gsl_blas_zgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], zero, k[3]);

        gsl_matrix_complex_memcpy(k_in[3],k[3]);
        gsl_matrix_complex_add(k_in[3],k[2]);
        gsl_matrix_complex_add(k_in[3],k[2]);
        gsl_matrix_complex_add(k_in[3],k[1]);
        gsl_matrix_complex_add(k_in[3],k[1]);
        gsl_matrix_complex_add(k_in[3],k[0]);
        gsl_matrix_complex_scale(k_in[3], one_sixth);
        gsl_matrix_complex_add(x_f,k_in[3]);
        t += h_;
    }

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_complex_free(k[i]);
        gsl_matrix_complex_free(k_in[i]);
    }
}

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)
{
    double h_min = H/RK4_MAX_SLICES;
    int nr = x_i->size1;
    int nc = x_i->size2;
    gsl_matrix* k[4];
    gsl_matrix* k_in[4];
    gsl_matrix* A_val = gsl_matrix_alloc(nr,nc);
    for (int i = 0; i < 4; ++i)
    {
        k[i] = gsl_matrix_calloc(nr,nc);
        k_in[i] = gsl_matrix_calloc(nr,nc);
    }

    double t = t_i;
    double t_f = t_i + H;
    double h = H/10.; // Initial h. This is a bit conservative, but will be refined further.
    gsl_matrix_memcpy(x_f,x_i);

    gsl_matrix* x1 = gsl_matrix_alloc(nr,nc);
    gsl_matrix* x2 = gsl_matrix_alloc(nr,nc);

    while (t<t_f)
    {
        // Evaluate x1
        gsl_matrix_memcpy(x1,x_f);

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x_f);

        A(t+0.5*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x_f);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x1,k_in[3]);

        t += h;

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x1);

        A(t+0.5*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x1);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x1);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x1,k_in[3]);

        t -= h;

        // Evaluate x2
        gsl_matrix_memcpy(x2,x_f);

        A(t,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, x_f, 0., k[0]);

        gsl_matrix_memcpy(k_in[0], k[0]);
        gsl_matrix_scale(k_in[0],0.5);
        gsl_matrix_add(k_in[0],x_f);

        A(t+h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[0], 0., k[1]);

        gsl_matrix_memcpy(k_in[1], k[1]);
        gsl_matrix_scale(k_in[1],0.5);
        gsl_matrix_add(k_in[1],x_f);

        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[1], 0., k[2]);

        gsl_matrix_memcpy(k_in[2], k[2]);
        gsl_matrix_add(k_in[2],x_f);

        A(t+2*h,A_val,params);
        gsl_blas_dgemm(CblasNoTrans, CblasNoTrans, 2*h, A_val, k_in[2], 0., k[3]);

        gsl_matrix_memcpy(k_in[3],k[3]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[2]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[1]);
        gsl_matrix_add(k_in[3],k[0]);
        gsl_matrix_scale(k_in[3], 1./6.);
        gsl_matrix_add(x2,k_in[3]);

        // Evaluate Error
        gsl_matrix_sub(x2,x1);
        double rho_to_the_fourth = (30.*h*delta)/GSL_MAX(gsl_matrix_max(x2), -1.*gsl_matrix_min(x2));
        rho_to_the_fourth = pow(rho_to_the_fourth,0.25);
        if (rho_to_the_fourth>1)
        {
            gsl_matrix_memcpy(x_f,x1);
            gsl_matrix_scale(x2, -1./15.);
            gsl_matrix_add(x_f,x2);
            t += 2*h;
            h = h*(GSL_MIN(rho_to_the_fourth, RK4_MAX_SCALE));
            h = GSL_MIN(0.5*(t_f-t),h);
        }
        else if(h > h_min)
        {
            h = h*(GSL_MAX(rho_to_the_fourth, RK4_MIN_SCALE));
        }
        else
        {
            gsl_matrix_memcpy(x_f,x1);
            gsl_matrix_scale(x2, -1./15.);
            gsl_matrix_add(x_f,x2);
            t += 2*h;
            h = h_min;
        }
    }

    gsl_matrix_free(A_val);

    for (int i = 0; i < 4; ++i)
    {
        gsl_matrix_free(k[i]);
        gsl_matrix_free(k_in[i]);
    }

    gsl_matrix_free(x1);
    gsl_matrix_free(x2);
}

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)
{
    double t = t_i;
    double t_f = t_i + H;
    double h_2 = h/2.;
    gsl_matrix_memcpy(x_f,x);

    A(t,eval,params);
    gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h_2,eval,x_f,0.,y);
    gsl_matrix_add(y,x_f);

    A(t+h_2,eval,params);
    gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h,eval,y,1.,x_f);

    t+=h;

    while(t<t_f)
    {
        A(t,eval,params);
        gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h,eval,x_f,1.,y);

        A(t+h_2,eval,params);
        gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h,eval,y,1.,x_f);

        t+= h;
    }

    A(t,eval,params);
    gsl_blas_dgemm(CblasNoTrans,CblasNoTrans,h_2,eval,x_f,1.,y);
    gsl_matrix_add(x_f,y);
    gsl_matrix_scale(x_f,0.5);

}

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)
{
    //int ndim = x_i->size1;


    //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));

    int n = 1;
    double h = H;
    __midpoint_method(x_i, t_i, H, h, A, R1[0], params, y, eval);
    double error = HUGE_VAL;

    gsl_matrix** temp;

    //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));
    while (error > H*delta && n<BULSTO_STEP_MAX)
    {
        n++;
        h = H/n;

        // Swapping the arrays of matrices to save space
        temp = R2;
        R2 = R1;
        R1 = temp;

        double scaler = n/(n-1.);
        scaler *= scaler;

        double scale = 1.;
        __midpoint_method(x_i, t_i, H, h, A, R1[0], params, y, eval);
        for (int m = 1; m < n; ++m)
        {
            scale *= scaler;
            gsl_matrix_memcpy(epsilon,R1[m-1]);
            gsl_matrix_sub(epsilon,R2[m-1]);
            gsl_matrix_scale(epsilon, 1./(scale-1.));

            gsl_matrix_memcpy(R1[m],R1[m-1]);
            gsl_matrix_add(R1[m],epsilon);
        }
        error = GSL_MAX(gsl_matrix_max(epsilon), -1.*gsl_matrix_min(epsilon));
        //printf("%d %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));
    }

    gsl_matrix_memcpy(x_f,R1[n-1]);

    return error;
}

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)
{
    int ndim = x_i->size1;
    gsl_matrix* xfin = gsl_matrix_alloc(ndim,ndim);
    double error = __bulsto_final_matrix_floquet_type_real_main(x_i, t_i, H, delta, A, xfin, params,
  y, eval, R1, R2, epsilon);
    nlayer++;
    if (error > H*delta && nlayer < BULSTO_MAX_LAYERS)
    {
        error = __bulsto_final_matrix_floquet_type_real_runner(nlayer, x_i, t_i, H/2., delta, A, xfin,
 params, y, eval, R1, R2, epsilon);
        error += __bulsto_final_matrix_floquet_type_real_runner(nlayer, xfin, t_i+(H/2.), H/2., delta,
 A, xfin, params, y, eval, R1, R2, epsilon);
    }
    gsl_matrix_memcpy(x_f,xfin);
    gsl_matrix_free(xfin);
    return error;
}

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)
{
    // This program only initializes and provides and frees temp variables
    int ndim = x_i->size1;
    gsl_matrix** R1 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));
    gsl_matrix** R2 = (gsl_matrix**) malloc((BULSTO_STEP_MAX+1)*sizeof(gsl_matrix*));

    for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
    {
        R1[i] = gsl_matrix_alloc(ndim,ndim);
        R2[i] = gsl_matrix_alloc(ndim,ndim);
    }

    gsl_matrix* y = gsl_matrix_alloc(ndim,ndim);
    gsl_matrix* eval = gsl_matrix_alloc(ndim,ndim);
    gsl_matrix* epsilon = gsl_matrix_calloc(ndim,ndim);

    __bulsto_final_matrix_floquet_type_real_runner(0, x_i, t_i, H, delta, A, x_f, params, y, eval, R1,
 R2, epsilon);
    //printf("%e %e %e\n",error, H, error/H);
    for (int i = 0; i <= BULSTO_STEP_MAX; ++i)
    {
        gsl_matrix_free(R1[i]);
        gsl_matrix_free(R2[i]);
    }
    free(R1);
    free(R2);
    gsl_matrix_free(y);
    gsl_matrix_free(eval);
    gsl_matrix_free(epsilon);
}
```

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

n	Number of elements of vector that $A(t)$ operates on
A	$A(t)$ 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.
T	Period of the evolution function $A(t)$ s.t. $A(t+T) = A(t)\ \forall t \in \mathbb{R}$
start	Starting values of the parameters
end	Ending values of the parameters
nstep	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.
largest_multiplier	Matrix to store the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.
largest_multiplier_abs	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.

```
{
    gsl_complex** mult_temp;
    double** mult_abs_temp;

    if (largest_multiplier)
    {
        mult_temp = largest_multiplier;
    }
    else
    {
        mult_temp = (gsl_complex**) malloc(nstep[0]*sizeof(gsl_complex*));
        for (int i = 0; i < nstep[0]; ++i)
        {
            mult_temp[i] = (gsl_complex*) malloc(nstep[1]*sizeof(gsl_complex));
        }
    }

    if (largest_multiplier_abs)
    {
        mult_abs_temp = largest_multiplier_abs;
    }
    else
    {
        mult_abs_temp = (double**) malloc(nstep[0]*sizeof(double*));
        for (int i = 0; i < nstep[0]; ++i)
        {
            mult_abs_temp[i] = (double*) malloc(nstep[1]*sizeof(double));
        }
    }

    double step[2] = {(end[0]-start[0])/(nstep[0]-1), (end[1]-start[1])/(nstep[1]-1)};
    #pragma omp parallel for collapse(2) schedule(guided)
    for (int i = 0; i < nstep[0]; ++i)
    {
        for (int j = 0; j < nstep[1]; ++j)
        {
            double param[2] = {start[0] + step[0]*i, start[1] + step[1]*j };
            stability[i][j] =
       floquet_get_stability_reals_general(n,A,param,T,&mult_temp[i][j],&mult_abs_temp[i][j]);
        }
    }

    if(!largest_multiplier)
    {
        for (int i = 0; i < nstep[0]; ++i)
        {
            free(mult_temp[i]);
        }
        free(mult_temp);
    }

    if(!largest_multiplier_abs)
    {
        for (int i = 0; i < nstep[0]; ++i)
        {
            free(mult_abs_temp[i]);
        }
        free(mult_abs_temp);
    }
}
```

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

n	Number of elements of vector that $A(t)$ operates on

Parameters

A	$A(t)$ 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.
T	Period of the evolution function $A(t)$ s.t. $A(t+T) = A(t)\ \forall t \in \mathbb{R}$
start	Starting value of the parameter
end	Ending value of the parameter
nstep	Number of steps to take inclusive of the first and last values. Should be at least 2. Behaviour not defined otherwise.
largest_multiplier	Array to store the largest (by abs) computed Floquet multipliers into. No multiplier will be stored if NULL is passed.
largest_multiplier_abs	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.

```
{
    gsl_complex* mult_temp;
    double* mult_abs_temp;

    if (largest_multiplier)
    {
        mult_temp = largest_multiplier;
    }
    else
    {
        mult_temp = (gsl_complex*) malloc(nstep*sizeof(gsl_complex));
    }

    if (largest_multiplier_abs)
    {
        mult_abs_temp = largest_multiplier_abs;
    }
    else
    {
        mult_abs_temp = (double*) malloc(nstep*sizeof(double));
    }

    double step = (end-start)/(nstep-1);
    #pragma omp parallel for
    for (int i = 0; i < nstep; ++i)
    {
        double param = start + step*i;
        stability[i] = floquet_get_stability_reals_general(n,A,&param,T,mult_temp+i,mult_abs_temp+i);
    }

    if(!largest_multiplier)
    {
        free(mult_temp);
    }
    if(!largest_multiplier_abs)
    {
        free(mult_abs_temp);
    }
}
```

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.

```
{
    gsl_matrix* X = gsl_matrix_alloc(n,n);
    gsl_matrix_set_identity(X);

    gsl_matrix* B = gsl_matrix_alloc(n,n);
    bulsto_final_matrix_floquet_type_real(X, 0., T, ERR_TOL, A, B, params);

    gsl_vector_complex* eigenvals = gsl_vector_complex_alloc(n);

    gsl_eigen_nonsymm_workspace* w = gsl_eigen_nonsymm_alloc(n);

    int n_eigenvals_evaluated = n;

    int err_code = gsl_eigen_nonsymm(B,eigenvals,w);
    if(err_code)
    {
        n_eigenvals_evaluated = w->n_evals;
    }

    double mult_max_abs = -HUGE_VAL;
    gsl_complex mult_max;
    double ev_test;
    for (int i = 0; i < n_eigenvals_evaluated; ++i)
    {
        ev_test = gsl_complex_logabs(gsl_vector_complex_get(eigenvals,i));
        if (mult_max_abs < ev_test)
        {
            mult_max_abs = ev_test;
            mult_max = gsl_vector_complex_get(eigenvals,i);
        }
    }

    if (largest_multiplier != NULL)
    {
        *largest_multiplier = mult_max;
    }
    if (largest_multiplier_abs != NULL)
    {
        *largest_multiplier_abs = gsl_complex_abs(mult_max);
    }

    gsl_eigen_nonsymm_free(w);
    gsl_vector_complex_free(eigenvals);
    gsl_matrix_free(B);
    gsl_matrix_free(X);

    if (mult_max_abs > ERR_EIGEN_TOL)
    {
        return 1;
    }
    else if (mult_max_abs < (-ERR_EIGEN_TOL))
    {
        if (n_eigenvals_evaluated < n)
        {
            return 2;
        }
        return -1;
    }
    else
    {
        return 0;
    }
}
```

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

```

#include "floquet.h"

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)
{
    gsl_matrix* X = gsl_matrix_alloc(n,n);
    gsl_matrix_set_identity(X);

    gsl_matrix* B = gsl_matrix_alloc(n,n);
    bulsto_final_matrix_floquet_type_real(X, 0., T, ERR_TOL, A, B, params);

    gsl_vector_complex* eigenvals = gsl_vector_complex_alloc(n);

    gsl_eigen_nonsymm_workspace* w = gsl_eigen_nonsymm_alloc(n);

    int n_eigenvals_evaluated = n;

    int err_code = gsl_eigen_nonsymm(B,eigenvals,w);
    if(err_code)
    {
        n_eigenvals_evaluated = w->n_evals;
    }

    double mult_max_abs = -HUGE_VAL;
    gsl_complex mult_max;
    double ev_test;
    for (int i = 0; i < n_eigenvals_evaluated; ++i)
    {
        ev_test = gsl_complex_logabs(gsl_vector_complex_get(eigenvals,i));
        if (mult_max_abs < ev_test)
        {
            mult_max_abs = ev_test;
            mult_max = gsl_vector_complex_get(eigenvals,i);
        }
    }

    if (largest_multiplier != NULL)
    {
        *largest_multiplier = mult_max;
    }
    if (largest_multiplier_abs != NULL)
    {
        *largest_multiplier_abs = gsl_complex_abs(mult_max);
    }

    gsl_eigen_nonsymm_free(w);
    gsl_vector_complex_free(eigenvals);
    gsl_matrix_free(B);
    gsl_matrix_free(X);

    if (mult_max_abs > ERR_EIGEN_TOL)
    {
        return 1;
    }
    else if (mult_max_abs < (-ERR_EIGEN_TOL))
    {
        if (n_eigenvals_evaluated < n)
        {
            return 2;
        }
        return -1;
    }
    else
    {
        return 0;
    }
}

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)
{
    gsl_complex* mult_temp;
    double* mult_abs_temp;

    if (largest_multiplier)
    {
        mult_temp = largest_multiplier;
    }
    else
    {
        mult_temp = (gsl_complex*) malloc(nstep*sizeof(gsl_complex));
    }

    if (largest_multiplier_abs)
    {
        mult_abs_temp = largest_multiplier_abs;
    }
    else
    {
        mult_abs_temp = (double*) malloc(nstep*sizeof(double));
    }

    double step = (end-start)/(nstep-1);
    #pragma omp parallel for
    for (int i = 0; i < nstep; ++i)
    {
        double param = start + step*i;
        stability[i] = floquet_get_stability_reals_general(n,A,&param,T,mult_temp+i,mult_abs_temp+i);
    }

    if(!largest_multiplier)
    {
        free(mult_temp);
    }
    if(!largest_multiplier_abs)
    {
        free(mult_abs_temp);
    }
}

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)
{
    gsl_complex** mult_temp;
    double** mult_abs_temp;

    if (largest_multiplier)
    {
        mult_temp = largest_multiplier;
    }
    else
    {
        mult_temp = (gsl_complex**) malloc(nstep[0]*sizeof(gsl_complex*));
        for (int i = 0; i < nstep[0]; ++i)
        {
            mult_temp[i] = (gsl_complex*) malloc(nstep[1]*sizeof(gsl_complex));
        }
    }

    if (largest_multiplier_abs)
    {
        mult_abs_temp = largest_multiplier_abs;
    }
    else
    {
        mult_abs_temp = (double**) malloc(nstep[0]*sizeof(double*));
        for (int i = 0; i < nstep[0]; ++i)
        {
            mult_abs_temp[i] = (double*) malloc(nstep[1]*sizeof(double));
        }
    }

    double step[2] = {(end[0]-start[0])/(nstep[0]-1), (end[1]-start[1])/(nstep[1]-1)};
    #pragma omp parallel for collapse(2) schedule(guided)
    for (int i = 0; i < nstep[0]; ++i)
    {
        for (int j = 0; j < nstep[1]; ++j)
        {
            double param[2] = {start[0] + step[0]*i, start[1] + step[1]*j };
            stability[i][j] =
floquet_get_stability_reals_general(n,A,param,T,&mult_temp[i][j],&mult_abs_temp[i][j]);
        }
    }

    if(!largest_multiplier)
    {
        for (int i = 0; i < nstep[0]; ++i)
        {
            free(mult_temp[i]);
        }
        free(mult_temp);
    }

    if(!largest_multiplier_abs)
    {
        for (int i = 0; i < nstep[0]; ++i)
        {
            free(mult_abs_temp[i]);
        }
        free(mult_abs_temp);
    }
}
```

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

```

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

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}
```

```


void hill_meissner(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*GSL_SIGN(t-M_PI)));
}

int main()
{
    int n = 2;
    double T = 2*M_PI;

    double start[2] = {9.,-1.};
    double end[2] = {0., 9.};
    int nstep[2] = {64,64};
    int** stability = (int**) malloc(nstep[0]*sizeof(int*));
    gsl_complex** largest_multiplier = NULL;
    double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));

    for (int i = 0; i < nstep[0]; ++i)
    {
        stability[i] = (int*) malloc(nstep[1]*sizeof(int));
        largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
    }


    floquet_get_stability_array_real_double_param_general(n, hill_meissner, T, start, end, nstep,
     stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file,"Hill-Meissner Stability Plot\n");
    fprintf(file,"$\\omega^2$\n");
    fprintf(file,"$\\alpha^2$\n");
    fprintf(file,"%lf %lf\n",start[0],start[1]);
    fprintf(file,"%lf %lf\n",end[0],end[1]);
    fprintf(file,"%d %d\n",nstep[0], nstep[1]);
    fclose(file);

    for (int i = 0; i < nstep[0]; ++i)
    {
        free(stability[i]);
        free(largest_multiplier_abs[i]);
    }

    free(stability);
    free(largest_multiplier_abs);
    return 0;
}
```

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

```

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

#define DELTA 0.0456

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}


void hill_meissner(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*GSL_SIGN(t-M_PI)));
    gsl_matrix_set(A_val,1,1,-DELTA);
}

int main()
{
    int n = 2;
    double T = 2*M_PI;

    double start[2] = {12.,2.};
    double end[2] = {0., 10.};
    int nstep[2] = {320,320};
    int** stability = (int**) malloc(nstep[0]*sizeof(int*));
    gsl_complex** largest_multiplier = NULL;
    double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));

    for (int i = 0; i < nstep[0]; ++i)
    {
        stability[i] = (int*) malloc(nstep[1]*sizeof(int));
        largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
    }


    floquet_get_stability_array_real_double_param_general(n, hill_meissner, T, start, end, nstep,
  stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file,"Damped Hill-Meissner Stability Plot ($\\delta=0.0456$)\n");
    fprintf(file,"$\\omega^2$\n");
    fprintf(file,"$\\alpha^2$\n");
    fprintf(file,"%lf %lf\n",start[0],start[1]);
    fprintf(file,"%lf %lf\n",end[0],end[1]);
    fprintf(file,"%d %d\n",nstep[0], nstep[1]);
    fclose(file);

    for (int i = 0; i < nstep[0]; ++i)
    {
        free(stability[i]);
        free(largest_multiplier_abs[i]);
    }

    free(stability);
    free(largest_multiplier_abs);
    return 0;
}
```

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

```

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

#define DELTA 0.0465

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}


void hill_meissner(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*GSL_SIGN(t-M_PI)));
    gsl_matrix_set(A_val,1,1,-DELTA);
}

int main()
{
    int n = 2;
    double T = 2*M_PI;

    double start[2] = {12.,2.};
    double end[2] = {0., 10.};
    int nstep[2] = {320,320};
    int** stability = (int**) malloc(nstep[0]*sizeof(int*));
    gsl_complex** largest_multiplier = NULL;
    double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));

    for (int i = 0; i < nstep[0]; ++i)
    {
        stability[i] = (int*) malloc(nstep[1]*sizeof(int));
        largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
    }


    floquet_get_stability_array_real_double_param_general(n, hill_meissner, T, start, end, nstep,
     stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file,"Damped Hill-Meissner Stability Plot ($\\delta=0.0456$)\n");
    fprintf(file,"$\\omega^2$\n");
    fprintf(file,"$\\alpha^2$\n");
    fprintf(file,"%lf %lf\n",start[0],start[1]);
    fprintf(file,"%lf %lf\n",end[0],end[1]);
    fprintf(file,"%d %d\n",nstep[0], nstep[1]);
    fclose(file);

    for (int i = 0; i < nstep[0]; ++i)
    {
        free(stability[i]);
        free(largest_multiplier_abs[i]);
    }

    free(stability);
    free(largest_multiplier_abs);
    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

```

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

#define K_DAMP 0.1

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}


void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
}

void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
    gsl_matrix_set(A_val,1,1,-(K_DAMP));
}

int main()
{
    int n = 2;
    double T = M_PI;

    double start[2] = {60.,-5.};
    double end[2] = {0., 20.};
    int nstep[2] = {320,320};
    int** stability = (int**) malloc(nstep[0]*sizeof(int*));
    gsl_complex** largest_multiplier = NULL;
    double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));

    for (int i = 0; i < nstep[0]; ++i)
    {
        stability[i] = (int*) malloc(nstep[1]*sizeof(int));
        largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
    }


    floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
nstep, stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file,"Damped Mathieu Stability Plot ($k=0.1$)\n");
    fprintf(file,"$\\delta$\n");
    fprintf(file,"$\\epsilon$\n");
    fprintf(file,"%lf %lf\n",start[0],start[1]);
    fprintf(file,"%lf %lf\n",end[0],end[1]);
    fprintf(file,"%d %d\n",nstep[0], nstep[1]);
    fclose(file);

    for (int i = 0; i < nstep[0]; ++i)
    {
        free(stability[i]);
        free(largest_multiplier_abs[i]);
    }

    free(stability);
    free(largest_multiplier_abs);
    return 0;
}
```

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

```

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

#define K_DAMP 0.5

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}


void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
}

void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
    gsl_matrix_set(A_val,1,1,-(K_DAMP));
}

int main()
{
    int n = 2;
    double T = M_PI;

    double start[2] = {60.,-5.};
    double end[2] = {0., 20.};
    int nstep[2] = {320,320};
    int** stability = (int**) malloc(nstep[0]*sizeof(int*));
    gsl_complex** largest_multiplier = NULL;
    double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));

    for (int i = 0; i < nstep[0]; ++i)
    {
        stability[i] = (int*) malloc(nstep[1]*sizeof(int));
        largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
    }


    floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
 nstep, stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file,"Damped Mathieu Stability Plot ($k=0.5$)\n");
    fprintf(file,"$\\delta$\n");
    fprintf(file,"$\\epsilon$\n");
    fprintf(file,"%lf %lf\n",start[0],start[1]);
    fprintf(file,"%lf %lf\n",end[0],end[1]);
    fprintf(file,"%d %d\n",nstep[0], nstep[1]);
    fclose(file);

    for (int i = 0; i < nstep[0]; ++i)
    {
        free(stability[i]);
        free(largest_multiplier_abs[i]);
    }

    free(stability);
    free(largest_multiplier_abs);
    return 0;
}
```

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

```

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

#define K_DAMP 1.

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}


void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
}

void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
    gsl_matrix_set(A_val,1,1,-(K_DAMP));
}

int main()
{
    int n = 2;
    double T = M_PI;

    double start[2] = {60.,-5.};
    double end[2] = {0., 20.};
    int nstep[2] = {320,320};
    int** stability = (int**) malloc(nstep[0]*sizeof(int*));
    gsl_complex** largest_multiplier = NULL;
    double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));

    for (int i = 0; i < nstep[0]; ++i)
    {
        stability[i] = (int*) malloc(nstep[1]*sizeof(int));
        largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
    }


    floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
 nstep, stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file,"Damped Mathieu Stability Plot ($k=1$)\n");
    fprintf(file,"$\\delta$\n");
    fprintf(file,"$\\epsilon$\n");
    fprintf(file,"%lf %lf\n",start[0],start[1]);
    fprintf(file,"%lf %lf\n",end[0],end[1]);
    fprintf(file,"%d %d\n",nstep[0], nstep[1]);
    fclose(file);

    for (int i = 0; i < nstep[0]; ++i)
    {
        free(stability[i]);
        free(largest_multiplier_abs[i]);
    }

    free(stability);
    free(largest_multiplier_abs);
    return 0;
}
```

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

```

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

#define K_DAMP 10.

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}


void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
}

void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
    gsl_matrix_set(A_val,1,1,-(K_DAMP));
}

int main()
{
    int n = 2;
    double T = M_PI;

    double start[2] = {60.,-5.};
    double end[2] = {0., 20.};
    int nstep[2] = {320,320};
    int** stability = (int**) malloc(nstep[0]*sizeof(int*));
    gsl_complex** largest_multiplier = NULL;
    double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));

    for (int i = 0; i < nstep[0]; ++i)
    {
        stability[i] = (int*) malloc(nstep[1]*sizeof(int));
        largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
    }


    floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
    nstep, stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file,"Damped Mathieu Stability Plot ($k=10$)\n");
    fprintf(file,"$\\delta$\n");
    fprintf(file,"$\\epsilon$\n");
    fprintf(file,"%lf %lf\n",start[0],start[1]);
    fprintf(file,"%lf %lf\n",end[0],end[1]);
    fprintf(file,"%d %d\n",nstep[0], nstep[1]);
    fclose(file);

    for (int i = 0; i < nstep[0]; ++i)
    {
        free(stability[i]);
        free(largest_multiplier_abs[i]);
    }

    free(stability);
    free(largest_multiplier_abs);
    return 0;
}
```

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

```c

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

#define K_DAMP 0.1

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}


void mathieu_undamped(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
}

void mathieu_damped_fixed_k(double t, gsl_matrix* A_val, void* param)
{
    double* par_temp = (double*) param;
    gsl_matrix_set_zero(A_val);
    gsl_matrix_set(A_val,0,1,1.);
    gsl_matrix_set(A_val,1,0,-(par_temp[1] + par_temp[0]*cos(2.*t)));
    gsl_matrix_set(A_val,1,1,-(K_DAMP));
}

int main()
{
    int n = 2;
    double T = M_PI;

    double start[2] = {60.,-5.};
    double end[2] = {0., 20.};
    int nstep[2] = {320,320};
    int** stability = (int**) malloc(nstep[0]*sizeof(int*));
    gsl_complex** largest_multiplier = NULL;
    double** largest_multiplier_abs = (double**) malloc(nstep[0]*sizeof(double*));

    for (int i = 0; i < nstep[0]; ++i)
    {
        stability[i] = (int*) malloc(nstep[1]*sizeof(int));
        largest_multiplier_abs[i] = (double*) malloc(nstep[1]*sizeof(double));
    }


    floquet_get_stability_array_real_double_param_general(n, mathieu_damped_fixed_k, T, start, end,
 nstep, stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    print_int_matrix_to_file_csv(nstep[0],nstep[1],stability,file);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    print_double_matrix_to_file_csv(nstep[0],nstep[1],largest_multiplier_abs,file);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file,"Damped Mathieu Stability Plot ($k=0.1$)\n");
    fprintf(file,"$\\delta$\n");
    fprintf(file,"$\\epsilon$\n");
    fprintf(file,"%lf %lf\n",start[0],start[1]);
    fprintf(file,"%lf %lf\n",end[0],end[1]);
    fprintf(file,"%d %d\n",nstep[0], nstep[1]);
    fclose(file);

    for (int i = 0; i < nstep[0]; ++i)
    {
        free(stability[i]);
        free(largest_multiplier_abs[i]);
    }

    free(stability);
    free(largest_multiplier_abs);
    return 0;
}
```

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

```

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

void print_int_matrix_to_file_csv(int n1, int n2, int** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%d,",m[i][j]);
        }
        fprintf(file,"%d\n",m[i][n2-1]);
    }
}

void print_double_matrix_to_file_csv(int n1, int n2, double** m, FILE* file)
{
    for (int i = 0; i < n1; ++i)
    {
        for (int j = 0; j < n2-1; ++j)
        {
            fprintf(file,"%lf,",m[i][j]);
        }
        fprintf(file,"%lf\n",m[i][n2-1]);
    }
}


void fitness_periodic(double t, gsl_matrix* A_val, void* param)
{
    double d = *((double*) param);
    double sint = sin(2.*M_PI*t);
    gsl_matrix_set(A_val,0,0,sint-d);
    gsl_matrix_set(A_val,0,1,d);
    gsl_matrix_set(A_val,1,0,d);
    gsl_matrix_set(A_val,1,1,-sint-d);
}

int main()
{
    int n = 2;
    double T = M_PI;

    double start = 0.;
    double end = 100.;
    int nstep = 1024;
    int* stability = (int*) malloc(nstep*sizeof(int));
    gsl_complex* largest_multiplier = NULL;
    double* largest_multiplier_abs = (double*) malloc(nstep*sizeof(double));

    floquet_get_stability_array_real_single_param_general(n, fitness_periodic, T, start, end, nstep,
   stability, largest_multiplier, largest_multiplier_abs);

    FILE* file = fopen("stability.csv","w");
    for (int i = 0; i < nstep-1; ++i)
    {
        fprintf(file,"%d,",stability[i]);
    }
    fprintf(file,"%d",stability[nstep-1]);
    fclose(file);

    file = fopen("largest_multiplier_abs.csv","w");
    for (int i = 0; i < nstep-1; ++i)
    {
        fprintf(file,"%lf,",largest_multiplier_abs[i]);
    }
    fprintf(file,"%lf",largest_multiplier_abs[nstep-1]);
    fclose(file);

    file = fopen("extraparams.txt","w");
    fprintf(file," Dominant Floquet Multiplier as a function of Dispersal rate $d$\n");
    fprintf(file,"$d$\n");
    fprintf(file,"$\\max(|\\rho|)$\n");
    fprintf(file,"%lf\n",start);
    fprintf(file,"%lf\n",end);
    fprintf(file,"%d\n",nstep);
    fclose(file);

    free(stability);
    free(largest_multiplier_abs);
    return 0;
}
```

Index