最適レギュレータ課題

2007 年 5 月 15 日

『最適制御入門』(加藤寛一郎 著、東京大学出版会) の第 9 章にある「航空機の経路角最適制御問題」を、実際にプログラムを作って解いて下さい。

.... ではあまりにも惨いので、4 つの細かい課題に分けました。順次解いていってください。
なお提出してくれたプログラムは、1 回ごとにランダムに入れ替える (T 先生発案!!) ので、次の人を苦労させないようにがんばっください。

注意事項

- プログラミングが得意な人は何の言語を使っても構いません。でも、できれば関数型の言語 (Haskel とか Lisp とか ML) はやめてください。
- 課題は一回こっきりですが、ここで作ったものは実際に研究でも使える可能性が高い (行列とか複素数の計算は特に) ので、できるだけ汎用性をもたせてプログラムを組むようにしてください。
- 提出は D1 の成岡 (4F ドクター部屋) までお願いします。プログラムソースは fenrir.naru@gmail.com までお願いします。
- 意外と大変なので、日程は余裕をもって取り掛かりましょう。

1 課題その 1: 締切日: 4/27

3×3 行列

$$\begin{bmatrix} 1 & 2 & 2 \\ 3 & 4 & 11 \\ 7 & 3 & 1 \end{bmatrix} \tag{1.1}$$

の固有値、固有ベクトルを求めてください。

ヒント：以下に示すプログラムソースを利用して、埋めてねと書いてある部分を埋めれば完成します。このプログラムがファイルとして欲しい人は取りに来てください。

Listing 1: common.h

```
1  #ifndef __COMMON_H
2  #define __COMMON_H
3
4  #define SUCCESS 0
5  #define FAIL 1
6
7  typedef unsigned char BOOL;
8  #ifndef TRUE
9    #define TRUE 1
10 #endif
11 #ifndef FALSE
```

```
  #define FALSE 0
#endif

#endif
```

Listing 2: complex.h

```
#ifndef __COMPLEX_H
#define __COMPLEX_H

#include "common.h"

/**
 * 複素数を表す構造体 (struct)
 *
 */
typedef struct {
  double real; ///< 実部
  double imaginary; ///< 虚部
} complex_t;

void complex_zero(complex_t *target);
void complex_print(complex_t target);
void complex_copy(complex_t *src, complex_t *dist);
BOOL complex_is_equal(complex_t a, complex_t b);
complex_t complex_add(complex_t a, complex_t b);
complex_t complex_sub(complex_t a, complex_t b);
complex_t complex_multi(complex_t a, complex_t b);
complex_t complex_scalar_add(complex_t a, double b);
complex_t complex_scalar_sub(complex_t a, double b);
complex_t complex_scalar_multi(complex_t a, double b);
complex_t complex_scalar_div(complex_t a, double b);
double complex_abs2(complex_t target);
double complex_abs(complex_t target);

#endif
```

Listing 3: complex.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <memory.h>

#include "complex.h"

/**
 * 複素数を 0で初期化します。
 *
 * @param target 複素数
 */
void complex_zero(complex_t *target){
  {
    // 埋めてね
  }
}

/**
 * 複素数を見やすい形で表示します。
 *
 * @param target 複素数
 */
```

```
25 void complex_print(complex_t target){
26   {
27     // 埋めてね
28   }
29 }
30
31 /**
32  * 複素数のコピーを行います。
33  *
34  * @param src コピー元
35  * @param dist コピー先
36  */
37 void complex_copy(complex_t *src, complex_t *dist){
38   {
39     // 埋めてね
40   }
41 }
42
43 /**
44  * 2つの複素数の値が等しいか調べます。
45  *
46  * @param a 複素数その 1
47  * @param b 複素数その 2
48  * @return 等しい場合は TRUE。そうでない場合は FALSE。
49  */
50 BOOL complex_is_equal(complex_t a, complex_t b){
51   {
52     // 埋めてね
53   }
54   return TRUE;
55 }
56
57 /**
58  * 2つの複素数の加算を行います。
59  *
60  * @param a 複素数その 1
61  * @param b 複素数その 2
62  * @return 加算した結果
63  */
64 complex_t complex_add(complex_t a, complex_t b){
65   complex_t result;
66   {
67     // 埋めてね
68   }
69   return result;
70 }
71
72 /**
73  * 2つの複素数の減算を行います。
74  *
75  * @param a 複素数 (ひかれる側)
76  * @param b 複素数 (ひく側)
77  * @return 減算した結果
78  */
79 complex_t complex_sub(complex_t a, complex_t b){
80   complex_t result;
81   {
82     // 埋めてね
83   }
84   return result;
85 }
86
87 /**
```

```
 * 2つの複素数の積算を行います。
 *
 * @param a 複素数その 1
 * @param b 複素数その 2
 * @return 積算した結果
 */
complex_t complex_multi(complex_t a, complex_t b){
  complex_t result;
  {
    // 埋めてね
  }
  return result;
}

/**
 * 複素数と実数の加算を行います。
 *
 * @param a 複素数
 * @param b 実数
 * @return 加算した結果 (複素数)
 */
complex_t complex_scalar_add(complex_t a, double b){
  complex_t result;
  {
    // 埋めてね
  }
  return result;
}

/**
 * 複素数から実数の減算を行います。
 *
 * @param a 複素数
 * @param b 引く実数
 * @return 減算した結果 (複素数)
 */
complex_t complex_scalar_sub(complex_t a, double b){
  complex_t result;
  {
    // 埋めてね
  }
  return result;
}

/**
 * 複素数を実数倍します。
 *
 * @param a 複素数
 * @param b 実数
 * @return 実数倍した結果 (複素数)
 */
complex_t complex_scalar_multi(complex_t a, double b){
  complex_t result;
  {
    // 埋めてね
  }
  return result;
}

/**
 * 複素数を実数割します。
 *
 * @param a 複素数
```

```
 * @param b 実数
 * @return 実数割した結果 (複素数)
 */
complex_t complex_scalar_div(complex_t a, double b){
  return complex_scalar_multi(a, 1.0 / b);
}

/**
 * 複素数のノルムの 2乗を求めます。
 *
 * @param target 対称となる複素数
 * @return ノルムの 2乗 (実数)
 */
double complex_abs2(complex_t target){
  return pow(target.real, 2) + pow(target.imaginary, 2);
}

/**
 * 複素数のノルムを求めます。
 *
 * @param target 対称となる複素数
 * @return ノルム (実数)
 */
double complex_abs(complex_t target){
  return sqrt(complex_abs2(target));
}
```

Listing 4: matrix.h

```
#ifndef __MATRIX_H
#define __MATRIX_H

#include "common.h"
#include "complex.h"

typedef unsigned char error_code_t;

/**
 * 行列を表す構造体 (struct)
 *
 */
typedef struct {
  double *values; ///< 行列の成分を保存するメモリへのポインタ
  unsigned int rows; ///< 行数
  unsigned int columns; ///< 列数
} matrix_t;

void matrix_init(matrix_t *target);
error_code_t matrix_alloc(matrix_t *target, unsigned int rows, unsigned int columns);
error_code_t matrix_zero(matrix_t *target);
error_code_t matrix_free(matrix_t *target);
inline double *matrix_value(matrix_t *target, unsigned int row, unsigned int column);
error_code_t matrix_print(matrix_t *target);
BOOL matrix_is_square(matrix_t *target);
BOOL matrix_is_symmetric(matrix_t *target);
error_code_t matrix_copy(matrix_t *src, matrix_t *dist);
error_code_t matrix_unit(matrix_t *target, unsigned int size);
error_code_t matrix_transpose(matrix_t *target, matrix_t *result);
error_code_t matrix_swap_rows(matrix_t *target, unsigned int row1, unsigned int row2);
error_code_t matrix_swap_columns(matrix_t *target, unsigned int column1, unsigned int column2);
error_code_t matrix_inverse(matrix_t *target, matrix_t *result);
error_code_t matrix_add(matrix_t *target1, matrix_t *target2, matrix_t *result);
error_code_t matrix_sub(matrix_t *target1, matrix_t *target2, matrix_t *result);
error_code_t matrix_multi(matrix_t *target1, matrix_t *target2, matrix_t *result);
```

```
36 error_code_t matrix_scalar_multi(matrix_t *target, double scalar);
37 error_code_t matrix_scalar_div(matrix_t *target, double scalar);
38
39 /**
40  * 行列を表す構造体 (struct)
41  * こちらは要素が複素数用。
42  *
43  */
44 typedef struct {
45   complex_t *values; ///< 要素を格納するためのメモリへのポインタ
46   unsigned int rows; ///< 行数
47   unsigned int columns; ///< 列数
48 } cmatrix_t;
49
50 void cmatrix_init(cmatrix_t *target);
51 error_code_t cmatrix_alloc(cmatrix_t *target, unsigned int rows, unsigned int columns);
52 error_code_t cmatrix_zero(cmatrix_t *target);
53 error_code_t cmatrix_free(cmatrix_t *target);
54 inline complex_t *cmatrix_value(cmatrix_t *target, unsigned int row, unsigned int column);
55
56 error_code_t cmatrix_print(cmatrix_t *target);
57
58 error_code_t matrix_eigen22(
59       matrix_t *target,
60       unsigned int row, unsigned int column,
61       complex_t *eigen_values);
62
63 error_code_t matrix_pivot_merge(
64         matrix_t *target,
65         unsigned int row, unsigned int column,
66         matrix_t *mergee);
67
68 error_code_t matrix_hessenberg(
69       matrix_t *target,
70       matrix_t *result,
71       matrix_t *transform);
72
73 #define EIGEN_VALUE_THRESHOLD 1E-10
74
75 error_code_t matrix_eigen(
76       matrix_t *target,
77       complex_t *eigen_values[],
78       cmatrix_t *eigen_vectors[]);
79
80 #endif /* __MATRIX_H */
```

Listing 5: matrix.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <memory.h>
6
7  #include "common.h"
8  #include "complex.h"
9  #include "matrix.h"
10
11 /**
12  * matrix_t 型を初期化します。
13  *
14  * @param target 対象とする行列
15  */
16 void matrix_init(matrix_t *target){
```

```
17    target−>values = NULL;
18    target−>rows = 0;
19    target−>columns = 0;
20  }
21
22  /**
23   * 行列にメモリを割り当てます。
24   *
25   * @param target 対象とする行列
26   * @param rows 行数
27   * @param columns 列数
28   * @return エラーがおきた際には SUCCESS 以外の値が返ります
29   */
30  error_code_t matrix_alloc(matrix_t *target, unsigned int rows, unsigned int columns){
31    if(!(target−>values = (double *)malloc(sizeof(double) * rows * columns))){
32      return FAIL;
33    }
34    target−>rows = rows;
35    target−>columns = columns;
36    return SUCCESS;
37  }
38
39  /**
40   * 行列の要素を 0で初期化します。
41   *
42   * @param target 対象とする行列
43   * @return エラーがおきた際には SUCCESS 以外の値が返ります
44   */
45  error_code_t matrix_zero(matrix_t *target){
46    {
47      // 埋めてね
48    }
49    return SUCCESS;
50  }
51
52  /**
53   * 行列に割り当てたメモリを解放します。
54   *
55   * @param target 行列
56   * @return エラーがおきた際には SUCCESS 以外の値が返ります
57   */
58  error_code_t matrix_free(matrix_t *target){
59    if(target−>values){
60      free(target−>values);
61      target−>values = NULL;
62      target−>rows = target−>columns = 0;
63    }
64    return SUCCESS;
65  }
66
67  /**
68   * 行列の成分を読み出し・書き込みする際に利用します。
69   * (高速化の為にinline 関数としています。)
70   *
71   * @param target 行列
72   * @param row 行番号 (開始はC 言語なので 0〜、つまり 1 行目にアクセスする場合は 0 を指定)
73   * @param column 列番号 (行番号と同じく 0〜)
74   * @return 要素へのポインタ
75   */
76  inline double *matrix_value(matrix_t *target, unsigned int row, unsigned int column){
77    return ((target−>values) + (row * target−>columns) + column);
78  }
79
```

```
80 /**
81  * 行列を見やすい形で出力します。
82  *
83  * @param target 行列
84  * @return エラーがおきた際には SUCCESS 以外の値が返ります
85  */
86 error_code_t matrix_print(matrix_t *target){
87   {
88     // 埋めてね
89   }
90   return SUCCESS;
91 }
92
93 /**
94  * 行列が正方行列であるか調べます。
95  *
96  * @return 正方行列である場合には TRUE、それ以外は FALSE
97  */
98 BOOL matrix_is_square(matrix_t *target){
99   {
100    // 埋めてね
101  }
102  return TRUE;
103 }
104
105 /**
106  * 行列が対称行列であるか調べます。
107  *
108  * @return 対称行列である場合には TRUE、それ以外は FALSE
109  */
110 BOOL matrix_is_symmetric(matrix_t *target){
111   if(!matrix_is_square(target)){return FALSE;}
112   {
113     // 埋めてね
114   }
115   return TRUE;
116 }
117
118 /**
119  * 行列のコピーをします。
120  *
121  * @param src コピー元
122  * @param dist コピー先 (matrix_alloc でメモリが自動的に割り当てられます、使い終わったら matrix_free を忘れずに)
123  * @return エラーがおきた際には SUCCESS 以外の値が返ります
124  */
125 error_code_t matrix_copy(matrix_t *src, matrix_t *dist){
126   if(matrix_alloc(dist, src->rows, src->columns)){return FAIL;}
127   {
128     // 埋めてね
129   }
130   return SUCCESS;
131 }
132
133 /**
134  * 単位行列を生成します。
135  *
136  * @param target 行列 (matrix_alloc でメモリが自動的に割り当てられます、使い終わったら matrix_free を忘れずに)
137  * @param size 行数、すなわち、列数
138  * @return エラーがおきた際には SUCCESS 以外の値が返ります
139  */
140 error_code_t matrix_unit(matrix_t *target, unsigned int size){
141   if(matrix_alloc(target, size, size)){return FAIL;}
142   {
```

```
143     // 埋めてね
144   }
145   return SUCCESS;
146 }
147
148 /**
149  * 行列の転置を行います。
150  *
151  * @param target 行列 (matrix_alloc でメモリが自動的に割り当てられます、使い終わったら matrix_free を忘れずに)
152  * @param result 結果の格納用
153  * @return エラーがおきた際には SUCCESS 以外の値が返ります
154  */
155 error_code_t matrix_transpose(matrix_t *target, matrix_t *result){
156   if(matrix_alloc(result, target->columns, target->rows)){return FALSE;}
157   {
158     // 埋めてね
159   }
160   return SUCCESS;
161 }
162
163 /**
164  * 行を入れ替えます。
165  *
166  * @param target 行列
167  * @param row1 行 1
168  * @param row2 行 2
169  * @return エラーがおきた際には SUCCESS 以外の値が返ります
170  */
171 error_code_t matrix_swap_rows(matrix_t *target, unsigned int row1, unsigned int row2){
172   if((row1 >= (target->rows)) || (row2 >= (target->rows))){return FAIL;}
173   {
174     double temp;
175     unsigned int j;
176     for(j = 0; j < target->columns; j++){
177       temp = *matrix_value(target, row1, j);
178       *matrix_value(target, row1, j) = *matrix_value(target, row2, j);
179       *matrix_value(target, row2, j) = temp;
180     }
181   }
182   return SUCCESS;
183 }
184
185 /**
186  * 列を入れ替えます。
187  *
188  * @param target 行列
189  * @param column1 列 1
190  * @param column2 列 2
191  * @return エラーがおきた際には SUCCESS 以外の値が返ります
192  */
193 error_code_t matrix_swap_columns(matrix_t *target, unsigned int column1, unsigned int column2){
194   if((column1 >= (target->columns)) || (column2 >= (target->columns))){return FAIL;}
195   {
196     double temp;
197     unsigned int i;
198     for(i = 0; i < target->rows; i++){
199       temp = *matrix_value(target, i, column1);
200       *matrix_value(target, i, column1) = *matrix_value(target, i, column2);
201       *matrix_value(target, i, column2) = temp;
202     }
203   }
204   return SUCCESS;
205 }
```

```
206
207 /**
208  * 逆行列を求めます。
209  *
210  * @param target 行列 (matrix_alloc でメモリが自動的に割り当てられます、使い終わったら matrix_free を忘れずに)
211  * @param result 結果の格納用
212  * @return エラーがおきた際には SUCCESS 以外の値が返ります
213  */
214 error_code_t matrix_inverse(matrix_t *target, matrix_t *result){
215   if(!matrix_is_square(target)){return FAIL;}
216   if(matrix_unit(result, target->rows)){return FAIL;}
217   {
218     // 埋めてね
219   }
220   return SUCCESS;
221 }
222
223 /**
224  * 行列の和を求めます。
225  *
226  * @param target1 行列 1
227  * @param target2 行列 2
228  * @param result 結果の格納用 (matrix_alloc でメモリが自動的に割り当てられます、使い終わったら matrix_free を忘れずに
         )
229  * @return エラーがおきた際には SUCCESS 以外の値が返ります
230  */
231 error_code_t matrix_add(matrix_t *target1, matrix_t *target2, matrix_t *result){
232   if((target1->rows != target2->rows)
233           || (target1->columns != target2->columns)){
234     return FAIL;
235   }
236   if(matrix_alloc(result, target1->rows, target1->columns)){return FAIL;}
237   {
238     // 埋めてね
239   }
240   return SUCCESS;
241 }
242
243 /**
244  * 行列の差を求めます。
245  *
246  * @param target1 行列 1
247  * @param target2 行列 2
248  * @param result 結果の格納用 (matrix_alloc でメモリが自動的に割り当てられます、使い終わったら matrix_free を忘れずに
         )
249  * @return エラーがおきた際には SUCCESS 以外の値が返ります
250  */
251 error_code_t matrix_sub(matrix_t *target1, matrix_t *target2, matrix_t *result){
252   if((target1->rows != target2->rows)
253           || (target1->columns != target2->columns)){
254     return FAIL;
255   }
256   if(matrix_alloc(result, target1->rows, target1->columns)){return FAIL;}
257   {
258     // 埋めてね
259   }
260   return SUCCESS;
261 }
262
263 /**
264  * 行列の積を求めます。
265  *
266  * @param target1 行列 1
```

```
267  * @param target2 行列 2
268  * @param result 結果の格納用 (matrix_alloc でメモリが自動的に割り当てられます、使い終わったら matrix_free を忘れずに
         )
269  * @return エラーがおきた際には SUCCESS 以外の値が返ります
270  */
271 error_code_t matrix_multi(matrix_t *target1, matrix_t *target2, matrix_t *result){
272   if(target1->columns != target2->rows){return FAIL;}
273   if(matrix_alloc(result, target1->rows, target2->columns)){return FAIL;}
274   {
275     // 埋めてね
276   }
277   return SUCCESS;
278 }
279
280 /**
281  * 行列をスカラー倍します。
282  * target *= scalar; なイメージ
283  *
284  * @param target 行列 (これがスカラー倍されます)
285  * @param scalar かける値
286  * @return エラーがおきた際には SUCCESS 以外の値が返ります
287  */
288 error_code_t matrix_scalar_multi(matrix_t *target, double scalar){
289   {
290     // 埋めてね
291   }
292   return SUCCESS;
293 }
294
295 /**
296  * 行列をスカラー割します。
297  * target /= scalar; なイメージ
298  *
299  * @param target 行列 (これがスカラー割されます)
300  * @param scalar 割る値
301  * @return エラーがおきた際には SUCCESS 以外の値が返ります
302  */
303 error_code_t matrix_scalar_div(matrix_t *target, double scalar){
304   return matrix_scalar_multi(target, 1. / scalar);
305 }
306
307 /**
308  * cmatrix_t(要素が複素数版)を初期化します。
309  *
310  * @param target 対象とする行列
311  */
312 void cmatrix_init(cmatrix_t *target){
313   target->values = NULL;
314   target->rows = 0;
315   target->columns = 0;
316 }
317
318 /**
319  * 行列 (要素が複素数版)にメモリを割り当てます。
320  *
321  * @param target 対象とする行列
322  * @param rows 行数
323  * @param columns 列数
324  * @return エラーがおきた際には SUCCESS 以外の値が返ります
325  */
326 error_code_t cmatrix_alloc(cmatrix_t *target, unsigned int rows, unsigned int columns){
327   if(!(target->values = (complex_t *)malloc(sizeof(complex_t) * rows * columns))){
328     return FAIL;
```

```
329     }
330     target->rows = rows;
331     target->columns = columns;
332     return SUCCESS;
333 }
334 
335 /**
336  * 行列の要素を 0で初期化します。
337  *
338  * @param target 対象とする行列
339  * @return エラーがおきた際には SUCCESS 以外の値が返ります
340  */
341 error_code_t cmatrix_zero(cmatrix_t *target){
342     {
343       // 埋めてね
344     }
345     return SUCCESS;
346 }
347 
348 /**
349  * 行列 (要素が複素数版)に割り当てたメモリを解放します。
350  *
351  * @param target 行列
352  * @return エラーがおきた際には SUCCESS 以外の値が返ります
353  */
354 error_code_t cmatrix_free(cmatrix_t *target){
355     if(target->values){
356       free(target->values);
357       target->values = NULL;
358       target->rows = target->columns = 0;
359     }
360     return SUCCESS;
361 }
362 
363 /**
364  * 行列 (要素が複素数版)の成分を読み出し・書き込みする際に利用します。
365  * (高速化の為にinline 関数としています。)
366  *
367  * @param target 行列
368  * @param row 行番号 (開始はC 言語なので 0～、つまり 1 行目にアクセスする場合は 0 を指定)
369  * @param column 列番号 (行番号と同じく 0～)
370  * @return 要素へのポインタ
371  */
372 inline complex_t *cmatrix_value(cmatrix_t *target, unsigned int row, unsigned int column){
373     return ((target->values) + (row * target->columns) + column);
374 }
375 
376 /**
377  * 行列を見やすい形で出力します。
378  *
379  * @param target 行列
380  * @return エラーがおきた際には SUCCESS 以外の値が返ります
381  */
382 error_code_t cmatrix_print(cmatrix_t *target){
383     {
384       // 埋めてね
385     }
386     return SUCCESS;
387 }
388 
389 /**
390  * 2行 2列小行列の固有値を求めます。
391  *
```

```
392  * @param target 行列
393  * @param row 小行列の左上の行番号
394  * @param column 小行列の左上の列番号
395  * @param eigen_values 固有値 (NULL の場合は内部で malloc によって自動的にメモリが確保されます、free を忘れずに)
396  * @return エラーがおきた際には SUCCESS 以外の値が返ります
397  */
398 error_code_t matrix_eigen22(matrix_t *target, unsigned int row, unsigned int column, complex_t *eigen_values){
399   if(row + 1 >= target->rows || column + 1 >= target->columns){return FAIL;}
400   if(!eigen_values){
401     if(eigen_values = (complex_t *)malloc(sizeof(complex_t) * 2)){
402       return FAIL;
403     }
404   }
405   {
406     double a = *matrix_value(target, row, column),
407            b = *matrix_value(target, row, column + 1),
408            c = *matrix_value(target, row + 1, column),
409            d = *matrix_value(target, row + 1, column + 1);
410     double root = pow((a - d), 2) + b * c * 4;
411     if(root >= 0){
412       root = sqrt(root);
413       eigen_values[0].real = (a + d + root) / 2;
414       eigen_values[0].imaginary = 0;
415       eigen_values[1].real = (a + d - root) / 2;
416       eigen_values[1].imaginary = 0;
417     }else{
418       root = sqrt(root * -1);
419       eigen_values[0].real = (a + d) / 2;
420       eigen_values[0].imaginary = root / 2;
421       eigen_values[1].real = (a + d) / 2;
422       eigen_values[1].imaginary = -root / 2;
423     }
424   }
425   return SUCCESS;
426 }
427
428 /**
429  * ピボットを指定して､加算します。
430  * イメージとしては (target の部分行列) += mergee;
431  *
432  * @param target 行列
433  * @param row 行インデックス
434  * @param column 列インデックス
435  * @param mergee 足す行列
436  * @return エラーがおきた際には SUCCESS 以外の値が返ります
437  */
438 error_code_t matrix_pivot_merge(
439         matrix_t *target,
440         unsigned int row, unsigned int column,
441         matrix_t *mergee){
442   unsigned int i, j;
443   for(i = 0; i < mergee->rows; i++){
444     if(row + i < 0){continue;}
445     else if(row + i >= target->rows){break;}
446     for(j = 0; j < mergee->columns; j++){
447       if(column + j < 0){continue;}
448       else if(column + j >= target->columns){break;}
449       *matrix_value(target, row + i, column + j) += (*matrix_value(mergee, i, j));
450     }
451   }
452   return SUCCESS;
453 }
454
```

```
455 /**
456  * ハウスホルダー変換をしてヘッセンベルク行列を得ます。
457  *
458  * @param target 対象となる行列
459  * @param result 求めたヘッセンベルク行列を格納するためのポインタ
460  * @param transform 変換に用いた行列の積を格納するポインタ
461  * @return エラーがおきた際には SUCCESS 以外の値が返ります
462  */
463 error_code_t matrix_hessenberg(
464     matrix_t *target,
465     matrix_t *result,
466     matrix_t *transform){
467
468   unsigned int i, j;
469
470   if(!matrix_is_square(target)){return FAIL;}
471
472   matrix_copy(target, result);
473   matrix_unit(transform, target->rows);
474   for(j = 0; j < target->columns - 2; j++){
475     double t, s;
476     t = 0;
477     for(i = j + 1; i < target->rows; i++){
478       t += pow(*matrix_value(result, i, j), 2);
479     }
480     s = sqrt(t);
481     if(*matrix_value(result, j + 1, j) < 0){s *= -1;}
482
483     {
484       matrix_t omega, omega_trans, omega_multi_omega_trans;
485       matrix_t p, p_result, p_result_p;
486       matrix_t transform_p;
487
488       matrix_alloc(&omega, target->rows - (j+1), 1);
489       for(i = 0; i < omega.rows; i++){
490         *matrix_value(&omega, i, 0) = *matrix_value(result, j+i+1, j);
491       }
492       *matrix_value(&omega, 0, 0) += s;
493
494       matrix_transpose(&omega, &omega_trans);
495       matrix_multi(&omega, &omega_trans, &omega_multi_omega_trans);
496
497       matrix_unit(&p, target->rows);
498       matrix_scalar_div(
499           &omega_multi_omega_trans,
500           -(t + (*matrix_value(result, j + 1, j)) * s)
501         );
502       matrix_pivot_merge(
503           &p,
504           j+1,
505           j+1,
506           &omega_multi_omega_trans
507         );
508
509       matrix_multi(&p, result, &p_result);
510       matrix_multi(&p_result, &p, &p_result_p);
511
512       matrix_free(result);
513       matrix_copy(&p_result_p, result);
514
515       matrix_multi(transform, &p, &transform_p);
516       matrix_free(transform);
517       matrix_copy(&transform_p, transform);
```

```
518
519         matrix_free(&omega);
520         matrix_free(&omega_trans);
521         matrix_free(&omega_multi_omega_trans);
522         matrix_free(&p);
523         matrix_free(&p_result);
524         matrix_free(&p_result_p);
525         matrix_free(&transform_p);
526       }
527     }
528
529     //ゼロ処理
530     {
531       BOOL sym = matrix_is_symmetric(target);
532       for(j = 0; j < target->columns - 2; j++){
533         for(i = j + 2; i < target->rows; i++){
534           *matrix_value(result, i, j) = 0;
535           if(sym){*matrix_value(result, j, i) = 0;}
536         }
537       }
538     }
539
540     return SUCCESS;
541 }
542
543 /**
544  * 固有値、固有ベクトルを求めます。
545  *
546  * @param target 行列
547  * @param eigen_values 結果の格納用 (固有値)
548  * @param eigne_vectors 結果の格納用 (固有ベクトル)
549  * @return エラーがおきた際には SUCCESS 以外の値が返ります
550  */
551 error_code_t matrix_eigen(
552       matrix_t *target,
553       complex_t *eigen_values[],
554       cmatrix_t *eigen_vectors[]){
555
556     if(!matrix_is_square(target)){return FAIL;}
557
558     //ダブル QR 法
559     /* <手順>
560      * ハウスホルダー法を適用して、上ヘッセンベルク行列に置換後、
561      * ダブルQR 法を適用。
562      * 結果、固有値が得られるので、固有ベクトルを計算。
563      */
564
565     // 結果格納用メモリの確保
566     {
567       *eigen_values = (complex_t *)malloc(sizeof(complex_t) * target->rows);
568       *eigen_vectors = (cmatrix_t *)malloc(sizeof(cmatrix_t) * target->rows);
569     }
570
571     // 固有値の計算
572     {
573       // 変数の宣言
574       double mu_sum, mu_multi;
575       complex_t p1, p2;
576       int m;
577       BOOL first;
578       matrix_t transform, a, a_backup;
579
580       // 初期化
```

```
581     {
582       mu_sum = 0; mu_multi = 0;
583       complex_zero(&p1);
584       complex_zero(&p2);
585       m = target->rows;
586       first = TRUE;
587
588       matrix_hessenberg(target, &a, &transform);
589       matrix_copy(&a, &a_backup);
590     }
591
592     while(TRUE){
593
594       //m = 1 or m = 2
595       if(m == 1){
596         eigen_values[0]->real = *matrix_value(&a, 0, 0);
597         eigen_values[0]->imaginary = 0;
598         break;
599       }else if(m == 2){
600         matrix_eigen22(&a, 0, 0, eigen_values[0]);
601         break;
602       }
603
604       //μ、μ*の更新 (4.143)
605       {
606         complex_t p_new[2];
607         matrix_eigen22(&a, m-2, m-2, p_new);
608         if(first ? (first = FALSE) : TRUE){
609           if(complex_abs(complex_sub(p_new[0], p1)) > complex_abs(p_new[0]) / 2){
610             if(complex_abs(complex_sub(p_new[1], p2)) > complex_abs(p_new[1]) / 2){
611               mu_sum = complex_add(p1, p2).real;
612               mu_multi = complex_multi(p1, p2).real;
613             }else{
614               mu_sum = p_new[1].real * 2;
615               mu_multi = pow(p_new[1].real, 2);
616             }
617           }else{
618             if(complex_abs(complex_sub(p_new[1], p2)) > complex_abs(p_new[1]) / 2){
619               mu_sum = p_new[0].real * 2;
620               mu_multi = pow(p_new[0].real, 2);
621             }else{
622               mu_sum = complex_add(p_new[0], p_new[1]).real;
623               mu_multi = complex_multi(p_new[0], p_new[1]).real;
624             }
625           }
626         }
627         complex_copy(&p_new[0], &p1), complex_copy(&p_new[1], &p2);
628       }
629
630       //ハウスホルダー変換を繰り返す
631       {
632         double b1, b2, b3, r;
633         int i;
634         for(i = 0; i < m - 1; i++){
635           if(i == 0){
636             b1 = pow(*matrix_value(&a, 0, 0), 2)
637               - mu_sum * (*matrix_value(&a, 0, 0))
638               + mu_multi
639               + *matrix_value(&a, 0, 1) * *matrix_value(&a, 1, 0);
640             b2 = *matrix_value(&a, 1, 0)
641               * ((*matrix_value(&a, 0, 0)) + (*matrix_value(&a, 1, 1)) - mu_sum);
642             b3 = *matrix_value(&a, 2, 1) * (*matrix_value(&a, 1, 0));
643           }else{
```

```
644             b1 = *matrix_value(&a, i - 1);
645             b2 = *matrix_value(&a, i + 1, i - 1);
646             b3 = (i == m - 2 ? 0 : *matrix_value(&a, i + 2, i - 1));
647           }
648           /*printf("m: %d", m);
649           printf(" b1: %f", b1);
650           printf(" b2: %f", b2);
651           printf(" b3: %f\n", b3);*/
652
653           r = sqrt(pow(b1, 2) + pow(b2, 2) + pow(b3, 2));
654
655           {
656             matrix_t omega, omega_trans;
657             matrix_t omega_multi_omega_trans, omega_trans_multi_omega;
658             matrix_t p, pa, pap;
659
660             matrix_alloc(&omega, 3, 1);
661             {
662               *matrix_value(&omega, 0, 0) = b1 + r * (b1 >= 0 ? 1 : -1);
663               *matrix_value(&omega, 1, 0) = b2;
664               *matrix_value(&omega, 2, 0) = b3;
665             }
666             matrix_transpose(&omega, &omega_trans);
667
668             matrix_unit(&p, target->rows);
669             matrix_multi(&omega, &omega_trans, &omega_multi_omega_trans);
670             matrix_multi(&omega_trans, &omega, &omega_trans_multi_omega);
671             matrix_scalar_multi(
672                 &omega_multi_omega_trans,
673                 -2.0 / *matrix_value(&omega_trans_multi_omega, 0, 0));
674             matrix_pivot_merge(&p, i, i, &omega_multi_omega_trans);
675
676             matrix_multi(&p, &a, &pa);
677             matrix_multi(&pa, &p, &pap);
678
679             matrix_free(&a);
680             matrix_copy(&pap, &a);
681
682             matrix_free(&omega);
683             matrix_free(&omega_trans);
684             matrix_free(&omega_multi_omega_trans);
685             matrix_free(&omega_trans_multi_omega);
686             matrix_free(&p);
687             matrix_free(&pa);
688             matrix_free(&pap);
689           }
690         }
691       }
692
693       //収束判定
694       {
695         double epsilon = EIGEN_VALUE_THRESHOLD
696           * (1.
697               + fabs(
698                   fabs(*matrix_value(&a, m-2, m-2)) < fabs(*matrix_value(&a, m-1, m-1))
699                     ? *matrix_value(&a, m-2, m-2)
700                     : *matrix_value(&a, m-1, m-1)
701               ));
702         //printf("eps: %0.20f\n", epsilon);
703         if(fabs(*matrix_value(&a, m-1, m-2)) < epsilon){
704           m--;
705           (*eigen_values)[m].real = *matrix_value(&a, m, m);
706           (*eigen_values)[m].imaginary = 0;
```

```
707         }else if(fabs(*matrix_value(&a, m-2, m-3)) < epsilon){
708           m -= 2;
709           matrix_eigen22(&a, m, m, &((*eigen_values)[m]));
710         }
711       }
712     }
713
714     //固有ベクトルの計算
715     {
716       unsigned int i, j, k;
717       cmatrix_t x;
718
719       matrix_free(&a);
720       matrix_copy(&a_backup, &a);
721       cmatrix_alloc(&x, target->rows, target->rows); //固有ベクトル
722       cmatrix_zero(&x);
723
724       for(j = 0; j < target->rows; j++){
725         int n = target->rows;
726         for(i = 0; i < j; i++){
727           if(complex_is_equal((*eigen_values)[j], (*eigen_values)[i])){--n;}
728         }
729         cmatrix_value(&x, --n, j)->real = 1;
730         cmatrix_value(&x, n, j)->imaginary = 0;
731         while(n-- > 0){
732           *cmatrix_value(&x, n, j) = complex_multi(
733                 *cmatrix_value(&x, n+1, j),
734                 complex_scalar_sub((*eigen_values)[j], *matrix_value(&a, n+1, n+1))
735               );
736           for(i = n+2; i < target->rows; i++){
737             cmatrix_value(&x, n, j)->real
738               -= cmatrix_value(&x, i, j)->real * (*matrix_value(&a, n+1, i));
739             cmatrix_value(&x, n, j)->imaginary
740               -= cmatrix_value(&x, i, j)->imaginary * (*matrix_value(&a, n+1, i));
741           }
742           *cmatrix_value(&x, n, j)
743             = complex_scalar_div(
744                   *cmatrix_value(&x, n, j),
745                   *matrix_value(&a, n+1, n)
746                 );
747         }
748       }
749
750       //結果の格納
751       for(j = 0; j < x.columns; j++){
752         cmatrix_alloc(&((*eigen_vectors)[j]), target->rows, 1);
753         cmatrix_zero(&((*eigen_vectors)[j]));
754         for(i = 0; i < x.rows; i++){
755           for(k = 0; k < transform.columns; k++){
756             cmatrix_value(&((*eigen_vectors)[j]), i, 0)->real
757               += cmatrix_value(&x, k, j)->real
758                   * (*matrix_value(&transform, i, k));
759             cmatrix_value(&((*eigen_vectors)[j]), i, 0)->imaginary
760               += cmatrix_value(&x, k, j)->imaginary
761                   * (*matrix_value(&transform, i, k));
762           }
763         }
764
765         //正規化
766         {
767           double norm = 0;
768           for(i = 0; i < target->rows; i++){
769             norm += complex_abs2(*cmatrix_value(&((*eigen_vectors)[j]), i, 0));
```

```
            }
            norm = sqrt(norm);
            for(i = 0; i < target->rows; i++){
              cmatrix_value(&((*eigen_vectors)[j]), i, 0)->real /= norm;
              cmatrix_value(&((*eigen_vectors)[j]), i, 0)->imaginary /= norm;
            }
          }
        }

        cmatrix_free(&x);
      }

      // メモリの解放
      {
        matrix_free(&transform);
        matrix_free(&a);
        matrix_free(&a_backup);
      }
    }

    return SUCCESS;
}
```

Listing 6: OptimizedRegulator.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <memory.h>

#include "common.h"
#include "complex.h"
#include "matrix.h"

#define A_SIZE 3

#define X_u −0.00643
#define Z_u −0.0941
#define M_u −0.000222
#define X_w 0.0263
#define Z_w −0.624
#define M_w −0.00153
#define M_q −0.668
#define M_de −2.08
#define U_0 830
#define g 32.18
#define c_bar 27.31

#define rho 1E−4

#define theta_0 −0.01

void stage1(){
  unsigned int i;

  matrix_t a;
  complex_t *a_eigen_values;
  cmatrix_t *a_eigen_vectors;

  matrix_alloc(&a, A_SIZE, A_SIZE);

  *matrix_value(&a, 0, 0) = 1.;
  *matrix_value(&a, 0, 1) = 2.;
```

```
40    *matrix_value(&a, 0, 2) = 2.;
41
42    *matrix_value(&a, 1, 0) = 3.;
43    *matrix_value(&a, 1, 1) = 4.;
44    *matrix_value(&a, 1, 2) = 11.;
45
46    *matrix_value(&a, 2, 0) = 7.;
47    *matrix_value(&a, 2, 1) = 3;
48    *matrix_value(&a, 2, 2) = 1.;
49
50    matrix_eigen(&a, &a_eigen_values, &a_eigen_vectors);
51
52    for(i = 0; i < a.rows; i++){
53      printf("eigen_value [%d] : ", i);
54      complex_print(a_eigen_values[i]);
55      printf("\r\n");
56      printf("eigen_vector [%d]\r\n", i);
57      cmatrix_print(&a_eigen_vectors[i]);
58      printf("\r\n");
59      cmatrix_free(&a_eigen_vectors[i]);
60    }
61
62    free(a_eigen_values);
63    free(a_eigen_vectors);
64
65    matrix_free(&a);
66  }
67
68  void stage2(){
69
70    matrix_t a, b, c;
71    matrix_t r_1, r_2, r_3;
72
73    matrix_alloc(&a, 4, 4);
74    matrix_zero(&a);
75    {
76      *matrix_value(&a, 0, 0) = X_u;
77      *matrix_value(&a, 0, 1) = X_w;
78      *matrix_value(&a, 0, 2) = -g;
79
80      *matrix_value(&a, 1, 0) = Z_u;
81      *matrix_value(&a, 1, 1) = Z_w;
82      *matrix_value(&a, 1, 3) = U_0;
83
84      *matrix_value(&a, 2, 3) = 1;
85
86      *matrix_value(&a, 3, 0) = M_u;
87      *matrix_value(&a, 3, 1) = M_w;
88      *matrix_value(&a, 3, 3) = M_q;
89    }
90
91    matrix_alloc(&b, 4, 1);
92    matrix_zero(&b);
93    {
94      *matrix_value(&b, 3, 0) = M_de;
95    }
96
97    matrix_alloc(&c, 1, 4);
98    matrix_zero(&c);
99    {
100     *matrix_value(&c, 0, 1) = -1./U_0;
101     *matrix_value(&c, 0, 2) = 1;
102   }
```

```
103
104 printf("\nA"); matrix_print(&a);
105 printf("\nB"); matrix_print(&b);
106 printf("\nC"); matrix_print(&c);
107
108 matrix_alloc(&r_2, 1, 1);
109 matrix_zero(&r_2);
110 {
111   *matrix_value(&r_2, 0, 0) = rho;
112 }
113
114 matrix_alloc(&r_3, 1, 1);
115 matrix_zero(&r_3);
116 {
117   *matrix_value(&r_3, 0, 0) = 1;
118 }
119
120 {
121   matrix_t c_trans, c_trans_r_3;
122   matrix_transpose(&c, &c_trans);
123   matrix_multi(&c_trans, &r_3, &c_trans_r_3);
124   matrix_multi(&c_trans_r_3, &c, &r_1);
125   matrix_free(&c_trans);
126   matrix_free(&c_trans_r_3);
127 }
128
129
130 printf("\nR_1"); matrix_print(&r_1);
131 printf("\nR_2"); matrix_print(&r_2);
132 printf("\nR_3"); matrix_print(&r_3);
133
134 {
135   matrix_t z;
136   matrix_alloc(&z, a.rows + r_1.rows, a.columns * 2);
137   matrix_zero(&z);
138
139   // z の左上
140   {
141     matrix_pivot_merge(&z, 0, 0, &a);
142   }
143
144   // z の右上
145   {
146     matrix_t r_2_inv;
147     matrix_t b_trans;
148     matrix_t b_r_2_inv;
149     matrix_t b_r_2_inv_b_trans;
150
151     matrix_inverse(&r_2, &r_2_inv);
152     matrix_transpose(&b, &b_trans);
153     matrix_multi(&b, &r_2_inv, &b_r_2_inv);
154     matrix_multi(&b_r_2_inv, &b_trans, &b_r_2_inv_b_trans);
155     matrix_scalar_multi(&b_r_2_inv_b_trans, -1);
156
157     matrix_pivot_merge(&z, 0, a.columns, &b_r_2_inv_b_trans);
158
159     matrix_free(&r_2_inv);
160     matrix_free(&b_trans);
161     matrix_free(&b_r_2_inv);
162     matrix_free(&b_r_2_inv_b_trans);
163   }
164
165   // z の左下
```

```
166     {
167       matrix_t copy_r_1;
168       matrix_copy(&r_1, &copy_r_1);
169       matrix_scalar_multi(&copy_r_1, −1);
170 
171       matrix_pivot_merge(&z, a.rows, 0, &copy_r_1);
172 
173       matrix_free(&copy_r_1);
174     }
175 
176     // z の右下
177     {
178       matrix_t a_trans;
179       matrix_transpose(&a, &a_trans);
180       matrix_scalar_multi(&a_trans, −1);
181 
182       matrix_pivot_merge(&z, a.rows, a.columns, &a_trans);
183 
184       matrix_free(&a_trans);
185     }
186 
187     printf("\nz"); matrix_print(&z);
188 
189     {
190       int i;
191       complex_t ∗z_eigen_values;
192       cmatrix_t ∗z_eigen_vectors;
193 
194       matrix_eigen(&z, &z_eigen_values, &z_eigen_vectors);
195 
196       for(i = 0; i < z.rows; i++){
197         printf("eigen_value␣[%d]␣:␣", i);
198         complex_print(z_eigen_values[i]);
199         printf("\r\n");
200         printf("eigen_vector␣[%d]\r\n", i);
201         cmatrix_print(&z_eigen_vectors[i]);
202         printf("\r\n");
203         cmatrix_free(&z_eigen_vectors[i]);
204       }
205 
206       free(z_eigen_values);
207       free(z_eigen_vectors);
208     }
209   }
210 
211   matrix_free(&a);
212   matrix_free(&b);
213   matrix_free(&c);
214   matrix_free(&r_1);
215   matrix_free(&r_2);
216   matrix_free(&r_3);
217 }
218 
219 int main(){
220   stage1();
221   stage2();
222   return 0;
223 }
```

2　課題その 2: 締切日: 5/4

課題 1 を利用して、「航空機の経路角最適制御問題」にある行列 A, B, C を入力して、 Z の固有値と固有ベクトルを求めてください。

- 機体緒言は p.180 の値を用い、$\rho = 10^{-4}$ とする。
- 提出物はプログラムソース、固有値、固有ベクトルの計算結果。

3　課題その 3: 締切日: 5/11

課題 2 に引き続き、行列 W を作って、行列 P, F(フィードバックゲイン) を求めるプログラムを書いてください。

- 提出物はプログラムソース、および行列 P, F の数値。

4　課題その 4: 締切日: 5/18

課題 3 に引き続き、初期値応答 (初期値は　$\theta(0) = -0.01[\mathrm{rad}]$) を求めるプログラムを作成してください。

- Y の行き過ぎ量と定常値、静定時間を求めるプログラムも作成してください。定常値、静定時間の定義は各自で決めてください。
- 提出物はプログラムソース、および δ_e, γ, u のグラフ出力結果と、y の行き過ぎ量と定常値、及び静定時間。
- 4 次の Runge-Kutta 法により微分方程式を数値積分して系の応答を求めてください。
- グラフの出力にはできれば Matlab, Gnuplot のいずれかを使ってください。Excel はできれば避けましょう。