```c
#include "tool.h"

Matrice allocaMatrice(const unsigned ndim)
{
	// Variabili
	unsigned i;
	Matrice m;

	// Alloco memoria
	m = (Matrice)malloc(ndim * sizeof(Vettore));

	// Controllo che l'allocazione sia andata a buon fine
	if (m == NULL)
	{
		fprintf(stderr, "Allocazione non riuscita\n");
		exit(EXIT_FAILURE);
	}

	// Alloco i vettori riga
	for (i = 0; i < ndim; i++)
		m[i] = allocaVettore(ndim);

	// Ritorno la matrice allocata
	return m;
}

void liberaMatrice(Matrice m, const unsigned ndim)
{
	// Variabili
	unsigned i;

	// Libero ogni vettore riga
	for (i = 0; i < ndim; i++)
		free(m[i]);

	// Libero la matrice (l'unico vettore colonna)
	free(m);
}

Vettore allocaVettore(const unsigned ndim)
{
	// Variabili
	Vettore v;

	// Alloco memoria
	v = (Vettore)malloc(ndim * sizeof(Elemento));

	// Controllo che l'allocazione sia andata a buon fine
	if (v == NULL)
	{
		fprintf(stderr, "Allocazione non riuscita\n");
		exit(EXIT_FAILURE);
```

```
53          }
54
55          // Ritorno il vettore allocata
56          return v;
57  }
58
59  int *fattorizzazioneLU(Matrice *m, const unsigned ndim)
60  {
61          // Variabili
62          unsigned i, j, k;
63          int *pivot;
64          Vettore tmp;
65
66          // Istruzioni
67          // Alloco la memoria per il vettore delle informazioni per ricostruire il
    pivoting
68          pivot = (int *)malloc(ndim * sizeof(int));
69          // Riempio il pivot con [0,1,2,3,...] in modo da poi saper riordinarlo
70          for (i = 0; i < ndim; i++)
71                  pivot[i] = i;
72
73          for (k = 0; k < ndim - 1; k++)
74          {
75                  // Variabili
76                  int temp, ind_max;
77
78                  //Istruzioni
79                  // Cerco l'indice dell'elemento per effettuare il pivoting
80                  ind_max = k;
81                  for (i = k + 1; i < ndim; i++)
82                          if (fabs((*m)[i][k]) > fabs((*m)[ind_max][k]))
83                                  ind_max = i;
84
85                  // Effettuo il pivoting tra la riga k e la riga ind_max
86                  tmp = (*m)[ind_max];
87                  (*m)[ind_max] = (*m)[k];
88                  (*m)[k] = tmp;
89
90                  // Effettuo gli stessi cambi sul vettore pivot
91                  temp = pivot[ind_max];
92                  pivot[ind_max] = pivot[k];
93                  pivot[k] = temp;
94
95                  // Eseguo l'eliminazione gaussiana
96                  for (i = k + 1; i < ndim; i++)
97                  {
98                          if (fabs((*m)[k][k]) > TOL)
99                          {
100                                 (*m)[i][k] = (*m)[i][k] / (*m)[k][k];
101                                 for (j = k + 1; j < ndim; j++)
102                                         (*m)[i][j] -= (*m)[i][k] * (*m)[k][j];
103                         }
```

```
104                     else
105                     {
106                             fprintf(stderr, "Eliminazione Gaussiana non
    riuscita\n");
107                             exit(EXIT_FAILURE);
108                     }
109             }
110     }
111
112     // Ritorno il vettore contenente le informazioni dei pivoting effettuati
113     return pivot;
114 }
115
116 Vettore backwardSubstitution(const Matrice a, const Vettore y, const unsigned ndim)
117 {
118     // Variabili
119     Vettore x;
120     int j;
121     unsigned i;
122
123     // Alloco vettore delle incognite
124     x = allocaVettore(ndim);
125
126     // Eseguo sostituzione all'indietro
127     for (j = ndim - 1; j >= 0; j--)
128     {
129             Elemento sum;
130             sum = 0.0;
131             for (i = j + 1; i < ndim; ++i)
132                     sum += a[j][i] * x[i];
133             x[j] = (y[j] - sum) / a[j][j];
134     }
135
136     // Ritorno il vettore di soluzioni
137     return x;
138 }
139
140 Vettore forwardSubstitution(const Matrice a, const Vettore y, const unsigned ndim)
141 {
142     // Variabili
143     Vettore x;
144     unsigned i, j;
145
146     // Alloco vettore delle incognite
147     x = allocaVettore(ndim);
148
149     // Eseguo sostituzione in avanti
150     for (j = 0; j < ndim; j++)
151     {
152             Elemento sum;
153             sum = 0;
154             for (i = 0; i < j; ++i)
```

```
155                 {
156                         sum += a[j][i] * x[i];
157                 }
158                 x[j] = (y[j] - sum);
159         }
160
161         // Ritorno il vettore di soluzioni
162         return x;
163 }
164
165 Vettore permVett(const int *const pivot, const Vettore y, const unsigned ndim)
166 {
167         // Variabili
168         unsigned i;
169         Vettore Py;
170
171         // Alloco il vettore permutato dei termini noti
172         Py = allocaVettore(ndim);
173
174         // Riempio Py in modo permutato secondo il vettore pivot
175         for (i = 0; i < ndim; i++)
176                 Py[pivot[i]] = y[i];
177
178         // Ritorno il vettore permutato
179         return Py;
180 }
181
182 Vettore risolviSistemaLineare(Matrice A, Vettore y, const unsigned ndim)
183 {
184         // Variabili
185         int *pivot;
186         Vettore Py;
187         Vettore tmp;
188         Vettore sol;
189
190         // Calcolo la fattorizzazione LU
191         pivot = fattorizzazioneLU(&A, ndim);
192
193         // Permutazione vett termini noti
194         Py = permVett(pivot, y, ndim);
195
196         // Risoluzione sitema lineare Lz = Py = permVett(pivot, &y)
197         tmp = forwardSubstitution(A, Py, ndim);
198
199         // Risoluzione sistema lineare Ux = z = sol
200         sol = backwardSubstitution(A, tmp, ndim);
201
202         free(pivot);
203         free(Py);
204         free(tmp);
205         return sol;
206 }
```