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

```
                                fprintf(stderr, "Eliminazione Gaussiana non riuscita\n");
                                exit(EXIT_FAILURE);
                        }
                }
        }

        // Ritorno il vettore contenente le informazioni dei pivoting effettuati
        return pivot;
}

Vettore backwardSubstitution(const Matrice a, const Vettore y, const unsigned ndim)
{
        // Variabili
        Vettore x;
        int j;
        unsigned i;
        Elemento sum;

        // Alloco vettore delle incognite
        x = allocaVettore(ndim);

        // Eseguo sostituzione all'indietro
        for (j = ndim - 1; j >= 0; j--)
        {
                sum = 0.0;
                for (i = j + 1; i < ndim; ++i)
                        sum += a[j][i] * x[i];
                x[j] = (y[j] - sum) / a[j][j];
        }

        // Ritorno il vettore di soluzioni
        return x;
}

Vettore forwardSubstitution(const Matrice a, const Vettore y, const unsigned ndim)
{
        // Variabili
        Vettore x;
        unsigned i, j;
        Elemento sum;

        // Alloco vettore delle incognite
        x = allocaVettore(ndim);

        // Eseguo sostituzione in avanti
        for (j = 0; j < ndim; j++)
        {
                sum = 0;
                for (i = 0; i < j; ++i)
                        sum += a[j][i] * x[i];
                x[j] = (y[j] - sum);
```

```
154         }
155
156         // Ritorno il vettore di soluzioni
157         return x;
158 }
159
160 Vettore permVett(const int *const pivot, const Vettore y, const unsigned ndim)
161 {
162         // Variabili
163         unsigned i;
164         Vettore Py;
165
166         // Alloco il vettore permutato dei termini noti
167         Py = allocaVettore(ndim);
168
169         // Riempio Py in modo permutato secondo il vettore pivot
170         for (i = 0; i < ndim; i++)
171                 Py[pivot[i]] = y[i];
172
173         // Ritorno il vettore permutato
174         return Py;
175 }
176
177 Vettore risolviSistemaLineare(Matrice A, Vettore y, const unsigned ndim)
178 {
179         // Variabili
180         int *pivot;
181         Vettore Py;
182         Vettore tmp;
183         Vettore sol;
184
185         // Calcolo la fattorizzazione LU
186         pivot = fattorizzazioneLU(&A, ndim);
187
188         // Permutazione vett termini noti
189         Py = permVett(pivot, y, ndim);
190
191         // Risoluzione sitema lineare Lz = Py = permVett(pivot, &y)
192         tmp = forwardSubstitution(A, Py, ndim);
193
194         // Risoluzione sistema lineare Ux = z = sol
195         sol = backwardSubstitution(A, tmp, ndim);
196
197         free(pivot);
198         free(Py);
199         free(tmp);
200         return sol;
201 }
```