```cpp
#include "FFT.h"

namespace SoilMath
{
    FFT::FFT()
    {
    }


    FFT::~FFT()
    {
    }

    ComplexVect_t FFT::GetDescriptors(const cv::Mat &img)
    {
        if (!fftDescriptors.empty()) { return fftDescriptors; }

        complexcontour = Contour2Complex(img, img.cols / 2, img.rows / 2);

        // Supplement the vector of complex numbers so that N = 2^m
        uint32_t N = complexcontour.size();
        double logN = log(static_cast<double>(N)) / log(2.0);
        if (floor(logN) !=  logN)
        {
            // Get the next power of 2
            double nextLogN = floor(logN + 1.0);
            N = static_cast<uint32_t>(pow(2, nextLogN));

            uint32_t i = complexcontour.size();
            // Append the vector with zeros
            while (i++ < N) { complexcontour.push_back(Complex_t(0.0, 0.0)); }
        }

        ComplexArray_t ca(complexcontour.data(), complexcontour.size());
        fft(ca);
        fftDescriptors.assign(std::begin(ca), std::end(ca));
        return fftDescriptors;
    }

    iContour_t FFT::Neighbors(uchar *O, int pixel, uint32_t columns, uint32_t rows)
    {
        //long int LUT_nBore[8] = { 1, 1 + columns, columns, columns - 1, -1, -columns - 1, -columns, -columns + 1 };
        long int LUT_nBore[8] = { -columns + 1, -columns, -columns - 1, -1, columns - 1, columns, 1 + columns, 1 };
```

```
44          iContour_t neighbors;
45          uint32_t pEnd = rows * columns;
46          uint32_t count = 0;
47          for (uint32_t i = 0; i < 8; i++)
48          {
49              count = pixel + LUT_nBore[i];
50              while ((count < 0 || count >= pEnd) && i < 8) { count = pixel + LUT_nBore[++i]; }
51              if (i >= 8) { break; }
52              if (O[count] == 1) neighbors.push_back(count);
53          }
54          return neighbors;
55      }
56
57      // Depth first search with extension list,
58      // based upon: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/
          lecture-videos/lecture-4-search-depth-first-hill-climbing-beam/
59      ComplexVect_t FFT::Contour2Complex(const cv::Mat &img, float centerCol, float centerRow)
60      {
61          uchar *O = img.data;
62          uint32_t pEnd = img.cols * img.rows;
63
64          std::deque<std::deque<uint32_t>> sCont;
65          std::deque<uint32_t> eList;
66
67          //Initialize the queue
68          for (uint32_t i = 0; i < pEnd; i++)
69          {
70              if (O[i] == 1)
71              {
72                  std::deque<uint32_t> tmpQ;
73                  tmpQ.push_back(i);
74                  sCont.push_back(tmpQ);
75                  break;
76              }
77          }
78
79          if (sCont.front().size() < 1) { throw Exception::MathException("No contour found in image!"); } // Exception handling
80
81          uint32_t prev = -1;
82
83          // Extend path on queue
84          for (uint32_t i = sCont.front().front(); i < pEnd;)
```

```
85      {
86          iContour_t nBors = Neighbors(O, i, img.cols, img.rows); // find neighboring pixels
87          std::deque<uint32_t> cQ = sCont.front(); //store first queue;
88          sCont.erase(sCont.begin()); // erase first queue from beginning
89          if (cQ.size() > 1) { prev = cQ.size() - 2; }
90          else { prev = 0; }
91          // Loop through each neighbor
92          for (uint32_t j = 0; j < nBors.size(); j++)
93          {
94              if (nBors[j] != cQ[prev]) // No backtracking
95              {
96                  if (nBors[j] == cQ.front() && cQ.size() > 8) { i = pEnd; } // Back at first node
97                  if (std::find(eList.begin(), eList.end(), nBors[j]) == eList.end()) // Check if this current route is extended
                        elsewhere
98                  {
99                      std::deque<uint32_t> nQ = cQ;
100                     nQ.push_back(nBors[j]); // Add the neighbor to the queue
101                     sCont.push_front(nQ); // add the sequence to the front of the queue
102                 }
103             }
104         }
105         if (nBors.size() > 2) { eList.push_back(i); } // if there are multiple choices put current node in extension List
106         if (i != pEnd) { i = sCont.front().back(); } // If it isn't the end set i to the last node of the first queue
107         if (sCont.size() == 0) { throw Exception::MathException("No continuous contour found, or less then 8 pixels long!"); }
108     }
109
110     // convert the first queue to a complex normalized vector
111     Complex_t cPoint;
112     ComplexVect_t contour;
113     float col = 0.0;
114     //Normalize and convert the complex function
115     for_each(sCont.front().begin(), sCont.front().end(), [&img, &cPoint, &contour, &centerCol, &centerRow, &col](uint32_t &e)
116     {
117         col = (float)((e % img.cols) - centerCol);
118         if (col == 0.0) {   cPoint.real(1.0);}
119         else { cPoint.real((float)(col / centerCol)); }
120         cPoint.imag((float)((floorf(e / img.cols) - centerRow) / centerRow));
121         contour.push_back(cPoint);
122     });
123
124     return contour;
```

```
125     
126     }
127     
128     void FFT::fft(ComplexArray_t &CA)
129     {
130         const size_t N = CA.size();
131         if (N <= 1) { return; }
132     
133         ComplexArray_t even = CA[std::slice(0, N / 2, 2)];
134         ComplexArray_t  odd = CA[std::slice(1, N / 2, 2)];
135     
136         fft(even);
137         fft(odd);
138     
139         for (size_t k = 0; k < N / 2; ++k)
140         {
141             Complex_t ct = std::polar(1.0, -2 * M_PI * k / N) * odd[k];
142             CA[k] = even[k] + ct;
143             CA[k + N / 2] = even[k] - ct;
144         }
145     }
146     
147     void FFT::ifft(ComplexArray_t &CA)
148     {
149         CA = CA.apply(std::conj);
150         fft(CA);
151         CA = CA.apply(std::conj);
152         CA /= CA.size();
153     }
154 }
```