```
#include <cstddef>
#include <stdlib.h>
#include "DGM.h"

using namespace DirectGraphicalModels;

class Foo {

	public:

		Foo(byte n_states,  Size img_size):
			dense_graph(n_states) ,
			inferer(dense_graph),
			ext_graph(dense_graph)
		{

			/*Arguments:
					n_states: the number of depth states.
					img_size: size of image as (cols,rows).
			*/

			// build a 2D graph with image size and default potentials
			ext_graph.buildGraph(img_size);
		}
		~Foo(void) = default;

		void add_unary_pot(Mat unary_pot);

		void set_smooth_kernel(Vec2f sigma_smooth,  float w_smooth);

		void add_appearence_kernel(Mat feature, Vec2f sigma_xy, float sigma_feat, float
w_kernel);

		CGraphDense&	getGraph() { return dense_graph; }

		CInferDense& getInfer() { return inferer; }

		CGraphDenseExt& getGraphExt() { return ext_graph; }

	private:
		CGraphDense dense_graph;
		CInferDense inferer;
		CGraphDenseExt ext_graph;
};


#include "depth_regression.h"

void Foo::add_unary_pot(Mat unary_pot) {
```

```
	/* Arguments:
		unary_pot: unary potential matrix with the shape of (n_nodes, n_states).
	*/

	// adds the graph nodes with potentials.
	getGraph().addNodes(unary_pot);

	return;

}

void Foo::set_smooth_kernel(Vec2f sigma_xy,  float w_kernel){
	/*Arguments:
		sigma_xy: x-y std deviations of the 2D smoothness kernel.
		w_kernel: wighting parameter of the smoothness term.
	*/

	// add smoothness potential
	getGraphExt().addGaussianEdgeModel(sigma_xy, w_kernel);

	return;
}

void Foo::add_appearence_kernel(Mat feature, Vec2f sigma_xy, float sigma_feat, float w_kernel){
	/*Arguments:
		feature: a multi-channel feature matrix with shape of (cols,rows).
		sigma_xy: x-y std deviations of the 2D smoothness kernel,
		sigma_feat: std deviation of the feature.
		w_kernel: wighting parameter of the smoothness term.
	*/

	// add a Bilateral pairwise potentia
	getGraphExt().addBilateralEdgeModel(feature, sigma_xy, sigma_feat, w_kernel);

	return;
}
```

Unit Test:

```
#include<iostream>
#include<string>
#include "depth_regression.h"

void show(std::string name, Mat img)
{
	namedWindow(name, WINDOW_NORMAL);
	imshow(name, img);
```

```
	return;
}

int main(void){

	byte n_states = 1;

	const int cols = 9;

	const int rows = 9;

	const Size img_size = Size (cols,rows);

	Mat gt_seg(rows,  cols, CV_32FC1, Scalar(0));

	// draw a 3x3 rectangle
	for( int i = 3; i<6; i++){

		float * row_i = gt_seg.ptr<float>(i);

		for(int j=3; j<6; j++){

			row_i[j] = 1;

		}
	}

	Mat  gt_img;
	gt_seg.convertTo(gt_img, CV_8U, 255, 0);

	show("gt_img", gt_img);

	Foo fcrf {n_states, img_size};

	Mat unary_pot;
	// blur gt_seg to get foreground predictions
	GaussianBlur(gt_seg, unary_pot, Size(3, 3), 1.0, 1.0);

	show("unary_pot", unary_pot);

	unary_pot = unary_pot.reshape(1, cols*rows);

	fcrf.add_unary_pot(unary_pot);

	fcrf.set_smooth_kernel(Vec2f(3,3), 1);

	Mat features;
	// obtain features
	Canny(gt_img, features, 0, 1, 7);
```

```
        show("features", features);

        fcrf.add_appearence_kernel(features, Vec2f(3,3),  5, 2);

        // estimates the marginal potentials for each graph node
        vec_byte_t optimalDecoding = depth_fcrf.getInfer().decode(10);

        waitKey(0);

        return 0;

}
```