JOURNAL OF APPLIED
COMPUTER SCIENCE
Vol. 22 No. 1 (2014), pp. 29-48

# New Deformable Models Development Using the MESA Environment

**Cezary Bołdak, Daniel Reska, Marek Krętowski**

*Bialystok University of Technology*
*Faculty of Computer Science*
*Wiejska 45A, 15-351 Bialystok*
*c.boldak@pb.edu.pl*

**Abstract.** *In this work we present capabilities of a new environment called Medical Segmentation Arena (MESA) in developing new segmentation methods based on the deformable models. The MESA environment was created in frame of the project "Information Platform TEWI" to facilitate researches in the medical image processing domain. The operator can formulate new segmentation algorithms based on the deformable models theory (active contours - snakes) by composing them from ready-to-use blocks. He can also develop new blocks with a simple Java-based programming mechanism. Then he can easily evaluate these algorithms with many offered tools (image management and visualization, batch experiment planning and running, parametric studies, virtual phantom generation, segmentation quality assessment, distributing of computations). We give also some examples of the snake energies and models implemented in the MESA environment presenting its capabilities in practice.*
**Keywords:** *image segmentation, deformable models, segmentation methods development, image processing environment, Java.*

## 1. Motivations

The image processing domain is very challenging for researchers in terms of quantity of the initial work needed to start the experiments. The main difficulty

consists in setting up an environment allowing to manage (acquire, store, display) the input images and to visualize the effects of the processing. Two main approaches exist here, each one with its strong and weak sides.

- The first one is to write a stand-alone application using a general purpose programming language (C/C++, Java, C#, Python, ...). This approach gives the maximum of control for the developer, allowing him to perform all necessary operations on the image. But the main drawback, besides the need of the programming skills, is the amount of work to create such a program. Even if there exist libraries (e.g. OpenCV, VTK, ITK) to automate the most common image processing operations, it is still tedious and time-consuming.

- The second one it to use existing image processing tools (GIMP) performing all the standard image operations - the researcher can start working much faster focusing on his main goal. But unfortunately, possibilities of the image treatment given by such tools are in general much more limited in comparison to the first option.

There exist also solutions trying to combine the advantages of the both groups - one can see among them specialized languages with dedicated image processing toolboxes (Matlab, Scilab), but they still demand a lot of work from the researcher.

What would be necessary here is a complete framework dealing with all these image management operations and giving the (almost) complete control on the image processing. Moreover, in the case of a platform devoted to a certain class of problems, it could organize the whole work (workflow) allowing to customize only the steps under research. Simpler image operations should be easy to implement, even on the fly and without specialized programming skills. In the same time, assuming deeper IT knowledge, the system should be open to almost unlimited modifications, but with a moderate effort.

## 2. Deformable models

The image segmentation is one of the most common tasks performed while dealing with the visual information. The medical images stand out here as very challenging and of the great significance. Many techniques have been proposed so far, and those based on the deformable models (active contours, levels sets, with very numerous modifications and ameliorations) gained a lot of popularity thanks to their flexibility and quality of results [1].

The active contour is a shape that evolves under influence of applied forces to minimize its energy [2]:

$$E^*_{snake} = \int_0^1 E_{snake}(v(s))ds = \int_0^1 E_{int}(v(s)) + E_{image}(v(s)) + E_{con}(v(s))ds, \quad (1)$$

where $E_{int}$ is the internal energy (controlling the shape form), $E_{image}$ is the image energy (based most often on the image intensity or gradient and moving the snake toward the segmented object boundaries) and $E_{con}$ represents other possible constraints (*a priori* knowledge).

We limit our work to the discrete version of the active contours family, where the shape is represented as a collection of $N$ points - snaxels $s_i$ interconnected with primitives (lines in 2D, triangles in 3D). With the discrete formulation the integrals change to the sums calculated only in the snaxel positions (however the energy can be also calculated on the connecting primitives):

$$E^*_{snake} = \sum_{i=0}^{N-1} E_{snake}(s_i). \quad (2)$$

During the evolution the snake looks for a position and shape minimizing its energy. Considering the numerical realization two approaches can be applied:

- based directly on the energies - to examine the neighbourhood of each snaxel and to choose its new location that minimizes the local sum energy;
- based on the forces acting on each snaxel (computed directly or from the energy spatial distribution) - to calculate a local vector being sum of all forces and to deplace the snaxel by some distance along this vector.

This process is repeated iteratively until some stop condition is reached (e.g. threshold on the global energy drop).

The original snake model has significant limitations that have led (and are still leading) to different modifications. It should be initialized close to the segmented object boundaries and has natural tendency to shrink. An extra inflation (balloon) force [3], a local gradient expansion [4] allowing the snake to "see" more distant borders or a dual contour snake [5] help to overcome these drawbacks. The contour dynamic reformulation methods [6] or alternative shape representations (level sets [7], electric snake [8]) were also introduced to overcome the inflexibility of the standard snake fixed topology.

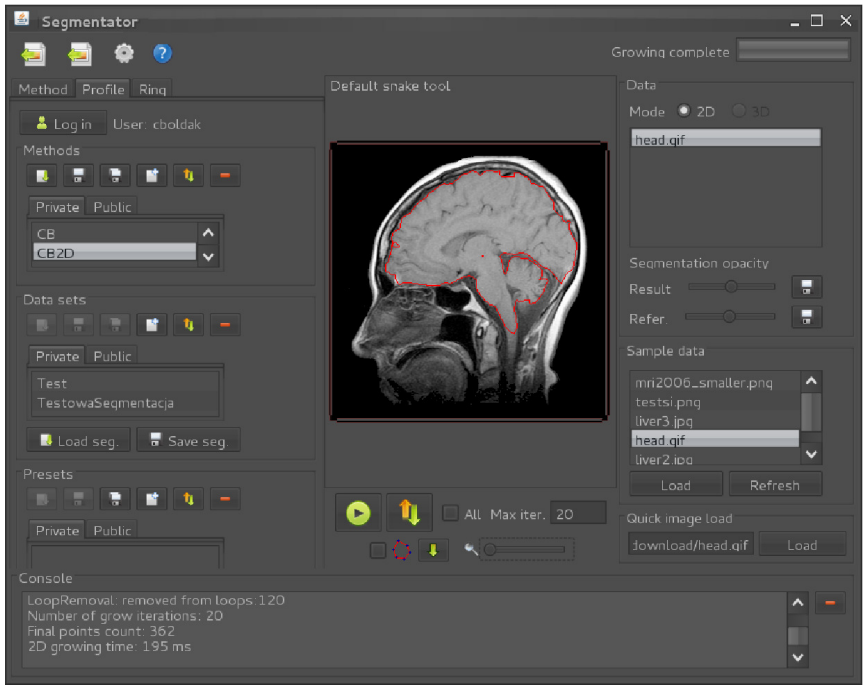


Figure 1. MESA environment Graphical User Interface (GUI)

## 3. MESA - Medical Segmentation Arena

The MESA - MEdical Segmentation Arena (Fig. 1) is a web platform for design and evaluation of segmentation methods based on the active contour theory [9]. Its main focus is on the medical images, 2D and 3D, however it can be also applied to images from other domains. The system is built in the client-server architecture, where a rich client (Java applet executed within the operator web browser) is a front-end to the powerful server machines serving computations in form of the web services. Thanks to this feature the operator can use this tool even when equiped with a moderate client machine (however in the 3D mode some hardware acceleration is needed to visualize 3D scenes efficiently). Moreover, the computations can be easily distributed and/or parallelized [10]. Our system offers also a centralized profile mechanism (based on the LDAP server and the central physical storage) protected with passwords where all personal data (input and re-

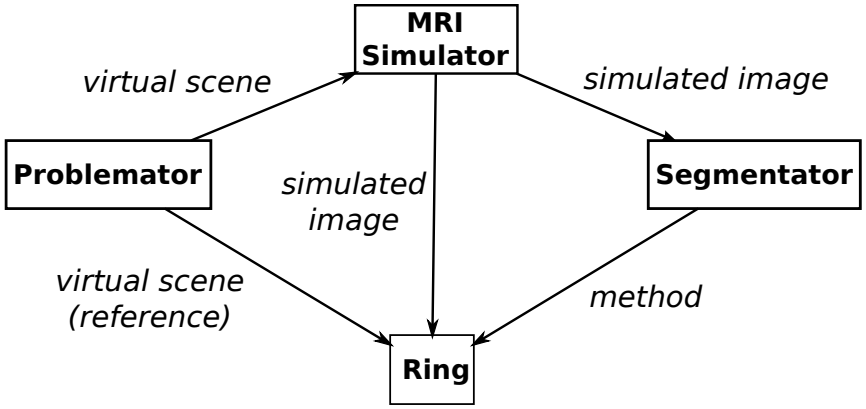


Figure 2. MESA research workflow

sult images, new methods and their parameters, experiments) can be safely stored and later accessed.

The system provides the complete environment to develop and evaluate new segmentation methods. As an element of the evaluation process a virtual phantom of the human body can be constructed. Then it becomes the input to an MRI aquisition simulation, producing artificial images to be segmented. In the same time, thanks to its known geometry, it is used (as a "ground-truth") to assess the precision of an examined segmentation method. The proposed research scenario (workflow - Fig. 2) consists of four steps executed in four modules:

- **Problemator** - for designing a virtual 2D/3D scene from basic geometrical shapes, this scene is then used as a body phantom with a given tissue parametrization (proton density, T1 and T2 times);
- **MRI Simulator** - simulating aquisition of MRI images from the phantom;
- **Segmentator** - core tool for development of the segmentation methods and for making single segmentations (e.g. on the simulated images);
- **Ring** - running series of experiments with the developed methods and given prameter sets (parametric studies), then comparing the results against the "ground-truth" (the phantom with known geometry) and calculating some quality metrics.

# 4. Segmentation methods creation

One of the main goals ot the MESA project was to facilitate the development and running (and finally - evaluation) of the segmentation methods based on the deformable models theory (and more specifically on the active contour - snake). Following the Equations 1 and 2 our computation framework (implemented in the module Segmentator) was designed to reflect the snake formulation and behaviour. Three main component classes/types are available there to create new segmentation metods on different levels of abstraction:

- **energies** calculate the energy in each contour element;
- **extensions** manipulate directly the snake position and form (the most often independently of the energies);
- **models** implement the segmentation algoritms, the most often using all the snake energies.

Each segmentation method is composed of one model and of any number of energies and extensions. Each extension is applied automatically to the contour in every iteration. The energy using is completely dependent on the model. However, in the most common scenario in each iteration the model:

- examines some local neigbourhood (which can vary from model to model) of every snaxel in the contour,
- calculates the local energy (sum of all energies) for each point in this neighbourhood,
- deplaces this snaxel to a new position minimizing the energy.

In every iteration all registred extensions are also run (once for the iteration, they can for instance regularize the contour shape) and some stop condition is verified.

## 4.1. Graphical construction of new segmentation methods using existing components

While working in the MESA environment, the operator can graphically compose a segmentation method from existing components (offered by the system or created by the user - see below). Each such method (the user can have several ones

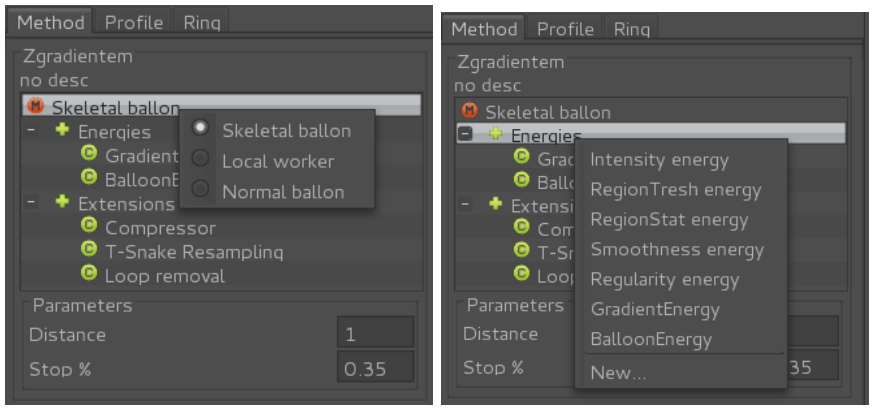


Figure 3. Segmentation method composed from components

stored in and loaded from his profile) is graphically represented in form of a tree (Fig. 3), where its single root specifies the used model. This tree has two main branches: in the first one the snake energies are grouped, in the second one - its extensions. The operator can easily (with the mouse) modify the tree by adding, removing and replacing the components. Each component can have its predefined parameters, which can be also adjusted in the system interface (Fig. 4). In the case of the energies one of these parameters is the **weight** applied to the energy while calculating the total sum energy.

This mode of working can be used to get familiarized with the environment or to teach/learn the active contour techniques. Its utility strongly depends on the pool of available components.

## 4.2. Creation of new energies within the environment

The snake energies are the most often the crucial element in a good segmentation. Experimenting with new energies can result in new snake techniques (e.g. balloon snake [3], electric snake [8]). Having our MESA environment the user can create new energies entirely in the system. Instead of adding an existing energy to the method tree, he can create a new one (Fig. 3). After clicking this added energy a new window opens where the energy definition can be written (Fig. 5). We used JSyntaxPane (code.google.com/p/jsyntaxpane) and BeanShell (www.beanshell.org) components to get the Java language code run-time editing

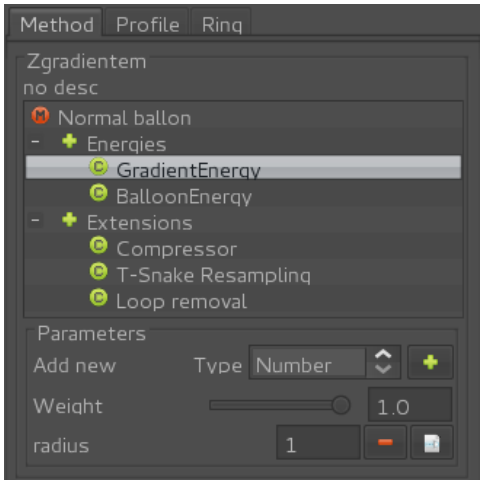


Figure 4. Energy component parameters

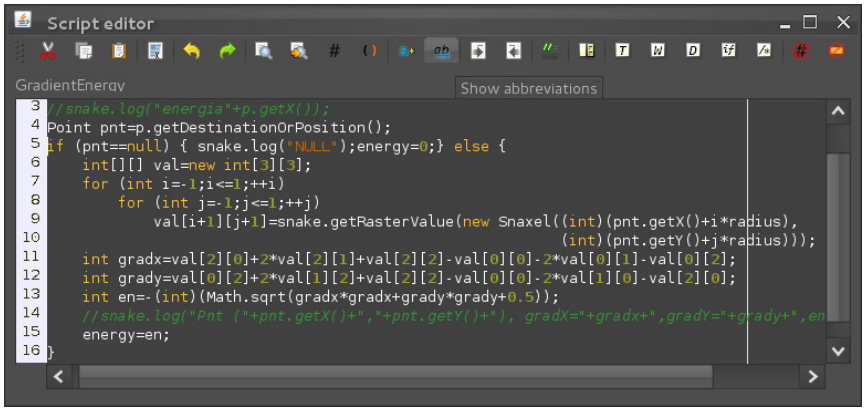


Figure 5. Gradient energy source code

and running functionalities. This definition needs to be the Java source code, where the calculated energy value should be assigned to the variable `energy`. Inside the code the following predefined variables can be accessed (see the class definitions in the next subsection):

- `p` of the class **Snaxel** – the snake snaxel under examination; two locations are embeded in this object: *position* – the original snaxel coordinates, *destination* – a potential new snake position; the method `getDestinationOrPosition():Point` returns the point coordinates where the energy actually is to be calculated;

- `snake` of the class **Snake** – object representing the active contour itself; among the most useful methods one can cite: `getRasterValue` reading the image intensity in the given point/snaxel, `log(String)` printing an arbitrary text into the console during the segmentation, methods accessing the snaxels (`getNextSnaxel(Snaxel)`, `addPoint(Snaxel)`, `iterator():Iterator<Snaxel>`) and calculating some useful values (`getNormalVector(Snaxel)`, `averageDist()`);

- *parameter names* – when the energy has defined some parameters, their currently set values can be accessed from the source code by their given names (`radius` in the gradient energy example – Figures 4 and 5).

The standard Java library classes can be also used (`java.lang.Math` for instance).

The most often this mode of working is sufficient to design and evaluate new segmetnation methods. It requires some skills in the Java language programming, but on a really basic level (arithmetic operations, variables declaring, methods calling).

### 4.3. External creation of new components

When the two described above modes are not sufficient to implement more sophisticated segmentation scenarios, one can develop new extensions and even models. Especially the latter ones allow to define custom segmentation algorithms going beyond the standard searching in the snaxel neighbourhood for a new location with a lower energy. To achieve this goal it is necessary to write new complete Java classes implementing interfaces or inheriting classes from the MESA Application Programming Inteface (API) and to pack them in a special way. This should be done outside the MESA system – for instance in a programming environment (like Eclipse or NetBeans) or using the bare Java SDK (Software Development Toolkit). In this mode all three component classes (energies, extensions and models) can be developed (however new energies can also be written in the embeded mode).

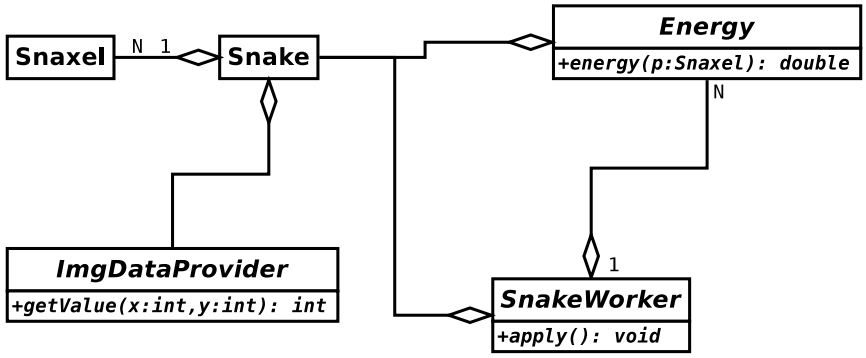


Figure 6. MESA API class diagram

### 4.3.1. MESA Application Programming Interface

In our computation framework five main classes have the following responsabilities (Fig. 6):

- **Snaxel** – represents a point building the snake;
- ***ImgDataProvider*** – abstract base type implemented by concrete image classes, it represents an image - environment where the snake evolves, each concrete class should override the `getValue` methods returning the image intensity in a given point;
- **Snake** – represents the active contour as a collection of snaxels, it has a reference to an image (*ImgDataProvider*) where it is situated;
- ***Energy*** – abstract base class for all classes representing implemented energies; each energy should override the method `energy(Snaxel):double` calculating the given snaxel energy; each energy has a reference to its snake, so it can read for example the neigbour snaxels positions;
- ***SnakeWorker*** – abstract base class for both contour models and extensions; for the models its `apply` method realizes the single iteration (so the main skeleton of the segmentation), for the extensions this method is called once in each iteration (for instance for the shape modification); this class stores the reference to the snake object and to all energies attributed to it.

The above list is only a general description. In reality some classes are subclassed to represent separately the 2D and 3D cases.

### 4.3.2. New components packaging

After writing new classes realizing new components it is necessary to properly pack them to use in the MESA environment. Finally, they will be put in a plugin - a JAR (Java ARchive) file with a special structure.

First of all, each new component class needs to be annotated (with the Java annotation framework). In this annotation the component class/type and name (displayed in the MESA environment) should be given (Listing 1). The component class/type specification is important since different types can have a common base class (e.g. extensions and models inherit `SnakeWorker`). If the component has parameters (see below), their names and default values also have to be specified here.

Listing 1. Example of component classes annotation

```
1  @Definition(
2      label="Skeletal model",
3      starter=true,
4      type=DefinitionType.MODEL
5  )
6  public class SkeletonModel extends SnakeWorker{ ... }
```

Then all components defined in the plugin need to be referenced by methods of a meta class (implementing `ComponentService` – Listing 7).

Finally, the meta class full package name (in the described example `tewi.segmed.snake.plugins.SkeletonService`) has to be listed in the file called `tewi.segmed.snake.defs.ComponentService` placed in the `META-INF/services` folder of the plugin JAR file.

After building a JAR file (e.g. with the Eclipse "Export as a JAR file" option) with the compiled classes and described above information, the plugin JAR file, once placed in the MESA `CLASSPATH`, will be automatically recognized and all its components made available in the main environment. This mechanism is based on the API ServiceLoader (http://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html).

### 4.3.3. Components parameters

All the component types can have parameters used in their methods. To define them in the source code and to use them in the MESA GUI during the snake evolution, they should be specified in two places (Listing 2):

- in the component class annotation (parameter names and default values);
- in the component constructor – their current values (fixed in the GUI – Fig. 3) are passed to the component object every time it is created (before each launching of the snake evolution iteration).

Listing 2. Example of component parameters

```
1  @Definition(
2     label="Skeletal ballon",
3     paramNames={"Distance", "Stop %"},
4     paramValues={"15","0.35"},
5     type=DefinitionType.MODEL )
6  public class SkeletonWorker extends SnakeWorker{
7    private double maxDistance;
8    private double changeRatio;
9
10   public SkeletonWorker(Snake2D snake,double maxD,double chgR){
11         super(snake);
12         this.maxDistance = maxD;
13         this.changeRatio = chgR;
14   }
15   public void apply() {
16     ...
17     snake.checkIsChanged(changeRatio);
18   }
19   ...
20 }
```

For the energies, the first parameter (the second constructor parameter, after the `snake` object) is always the energy `weight`, used internally by the MESA framework to multiply the obtained energy value. The component constuctor should save the parameters for a future use by the component methods (`apply` for the models and extensions, `energy` for the energies).

# 5. Existing model implementation

In the section 4 we gave the example of the gradient energy (Fig. 5) implemented in the embeded mode using the Java language scripting. Here we present a complete active contour model implemented in the external mode, with all the component classes (models, energies and extensions). This model is impossible to construct in the former one, because it uses a non standard contour representation and hence the optimization scheme.

## 5.1. Dual active contour

Introducing external forces to overcome the local minima (like the balloon force [3]) can cause the snake to not stop on the right border. The energy drop here can be smaller than the benefit of following this too strong external force. The well adjusted balance between all the forces/energies (their weights) is the key to a good segmentation and in the active contour applications this task usually needs a lot of work. Moreover, once fixed it can fail in even a similar class of images and objects to segment.

The dual active contour [5] is a trial to at least alleviate this problem. Instead of using a single contour, two contours are initialized and they evolve at the begining independently. The first one is placed inside the segmented object while the second one operates outside it. When both of them stop in (local) minima, an additional spring force (translated into energy) is applied to the snake with higher energy (so worse placed) attracing it to the better placed one (with lower energy). This force/energy is increased gradually until the affected snake moves. After exiting the local minimum and decreasing the energy the spring force/energy is removed and the configuration evolves again independently. The segmentation is finished when two contours converge in the global minimum (hopefully on the segmented object).

## 5.2. Two contours snake implementation in MESA

The model consisting of two contours can not be treated by the standard MESA models assuming only one closed sequence of snaxels. But we demonstrate here that it can be achieved by externally creating new components. Three component groups are needed: **model** built of two contours, **energies** adapted to this specific model and **extension** removing a potential reciprocal rotation of two contours.

### 5.2.1. Energies

The standard MESA energies could have been used here (applied separately to each contour) with one exception. The balloon energy should work differently for the inner and outer contours. For the former this energy should push it outward, for the latter – inward. To allow the model to identify this component (in order to apply it differently to two contours), a separate class was created with the code presented in Listing 3. To put all the components in a single JAR library, we implemented in this manner also other energies – gradient (source code like in Fig. 5) and rigidity (Listing 4) ones – however they are not specific to the dual active contour.

Listing 3. Balloon energy class

```
1  @Definition(label = "Balloon energy",
2      type = DefinitionType.ENERGY)
3  public class BalloonEnergy extends Energy {
4    public BalloonEnergy(Snake2D snake, double weight) {
5      super(snake, weight); }
6    public double energy(Snaxel p) {
7      Point pnt=p.getDestination();
8      if (p==null) return 0;
9      int xc,yc,x0=p.getX(),y0=p.getY(),
10       x=(int)(pnt.getX()), y=(int)(pnt.getY());
11     int[] pp=snake.getPointsArray();
12     for (int i=0;i<pp.length;i+=2) xc+=pp[i];yc+=pp[i+1];
13     xc/=pp.length/2; yc/=pp.length/2;
14     int vradx=x0-xc,vrady=y0-yc;
15     int vnewx=x-x0,vnewy=y-y0;
16     return -weight*(vradx*vnewx+vrady*vnewy)/
17         Math.sqrt(vradx*vradx+vrady*vrady);     } }
```

Listing 4. Rigidity energy class

```
1  @Definition(label = "Rigidity energy",
2      type = DefinitionType.ENERGY)
3  public class RigidityEnergy extends Energy {
4    public RigidityEnergy(Snake2D snake, double weight) {
5      super(snake, weight); }
6    public double energy(Snaxel p) {
7      Snaxel p=snake.getPreviousSnaxel(p),
8             n=snake.getNextSnaxel(p);
9      Point pnt=p.getDestination();
```

```
10      if (pnt==null) pnt=p.getPosition();
11      double px=p.getX()-pnt.getX(),py=p.getY()-pnt.getY(),
12             nx=n.getX()-pnt.getX(),ny=n.getY()-pnt.getY(),
13             len=Math.sqrt(px*px+py*py)* Math.sqrt(nx*nx+ny*ny);
14      if (Math.abs(len)<0.001) return 0;
15      return weight*(px*nx+py*ny)/len;   } }
```

### 5.2.2. Two contours snake model

The model class (Listing 5) starts from only one contour drawn graphically by the operator. On the first run (a simple geometrical test – line 10) the snaxels number is doubled – the second half of the `snake` holds the outer snake (the grown inner one). Every next iteration starts with temporarily splitting the `snake` into the inner and outer ones (line 14). Then each contour is minimized independently (the moved snaxels number is returned as well as the total energy for each one) – lines 15-16. Finally the single contour representation is restored by joining two these collections (line 17). If no snaxels moved (line 18), the additional spring energy is applied to the snake with higher energy (line 23-24). If this energy has been already applied, its effect is increased (lines 21-22). The snake is then re-run with this additional energy (line 25). In the opposite case (any snaxel moved) and if the spring energy has been already applied, it is kept until the affected contour decreases its energy (lines 26-28).

The single contour minimization (private method `run`) consists in searching the local neighbourhood of every its snaxel for a position minimizing its sum of energies. While cumulating the energies (line 37), the balloon energy is identified and its sign is negated for the outer snake (what causes it to shrink instead of growing – line 39). The spring energy is also added to the total energy – only for the affected contour (lines 41-44).

Listing 5. Simplified class representing the dual active contour model

```
1  @Definition(label = "Double snake",
2  type = DefinitionType.MODEL)
3  public class DoubleSnakeModel extends SnakeWorker {
4    ...
5    private boolean first=true;
6    private int springIn=0, springOut=0;
7    private double worseInitEnergy;
8    @Override
```

```
 9   public void apply() {
10     if (! isDoubleContour(snake)) {
11       //produce the outer contour and
12       // join their points to the 'snake' variable
13       return;       }
14     Snake2D in, out; //first and second half of 'snake'
15     iMove=run(in,true,out);iEnergy=sEnergy(in,true);
16     oMove=run(out,false,in);oEnergy=sEnergy(out,false);
17     snake=in+out;
18     if (iMove+oMove==0) {
19       if (first) { first=false;
20         worseInitEnergy=Math.max(iEnergy,oEnergy); }
21       if (springIn>0) ++springIn;
22       else if (springOut>0) ++springOut;
23       else if (iEnergy>oEnergy) ++springIn;
24       else ++springOut;
25       if (springIn+springOut<100) apply(); }
26     else if (springIn+springOut>0) {
27       curEnergy=(springIn>0)?iEnergy:oEnergy;
28       if (curEnergy>=worseInitEnergy) apply();  } }
29   private int run(Snake2D snake, boolean inner){
30     moveCounter=0;
31     for (Snaxel s:snake) {
32       moved=false;
33       for (int x=-searchRadius;x<=searchRadius;++x)
34         for (int y=-searchRadius;y<=searchRadius;++y) {
35           s.setDestination(s.getX()+x,s.getY()+y);
36           energy=0;
37           for (Energy e:energies) {
38             if (e instanceof BalloonEnergy && !inner)
39               energy-=e.energy(s);
40             else  energy+=e.energy(s);  }
41           if (inner && springIn>0)
42             energy+=springIn*springEnergy(s);
43           if (!inner && springOut>0)
44             energy+=springOut*springEnergy(s);
45           if (energy<minEnergy) {
46             moved=true; minEnergy=energy;
47             minPosition=s.getDestination(); } }
48       if (moved){ ++moveCounter;
49         s.setPosition(minPosition);  } }
50     return moveCounter; } }
```

### 5.2.3. Extension for removing rotation

When the additional spring force is applied, it attracts the worse contour points toward the corresponding points in the second contour. During the evolution these two contours can rotate reciprocally. The extension termed a rotation remover (Listing 6) removes this rotation by minimizing the sum of distances between corresponding points in two contours.

Listing 6. Extension class for removing reciprocal rotation of two contours

```
@Definition(label = "Rotation Remover",
type = DefinitionType.EXTENSION)
public class RemoveRotation extends SnakeWorker{
  public RemoveRotation(Snake2D snake) {
    super(snake); }
  private double rotDist(List<Snaxel> s1,List<Snaxel> s2,int o){
    int N=s1.size();  double distance=0;
    for (int i=0;i<N;++i) S
      distance+= s1.get(i).distEuc(s2.get((i+o)%N));
    return distance; }
  @Override
  public void apply() {
    List<Snaxel> snaxels = snake.getPoints();
    int N=snaxels.size()/2, minOff=0;
    List<Snaxel> in=new ArrayList<Snaxel>(),
          out=new ArrayList<Snaxel>();
    for (int i=0;i<N;++i) {
      in.add(snaxels.get(i));out.add(snaxels.get(i+N)); }
    double minDist=rotDistance(in, out, 0), dist;
    for (int off=1;off<N;++off) {
      dist=rotDistance(in, out, off);
      if (dist<minDist) {dist=minDist;minOff=off;} }
    for (int i=0;i<N;++i) in.add(out.get((i+minOff)%N));
    snake.setPoints(in);  } }
```

Finally, all these classes were packaged in the meta class describing the plugin (Listing 7).

Listing 7. Meta class referencing all the components of the dual contour plugin

```
public class DoubleSnakePluginService
      implements ComponentService {
  @Override
  public String get2DEnergies() {
    return DefsMaker.getDefinitions(BalloonEnergy.class,
        GradientEnergy.class,RigidityEnergy.class); }
```

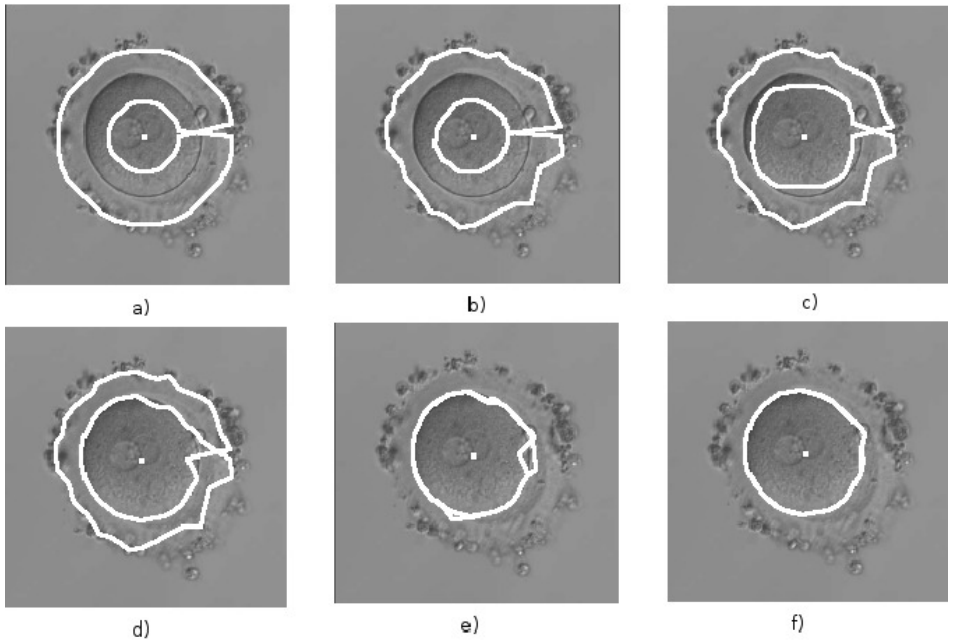


Figure 7. An example of dual active contour segmenting the human cell image

```
 7     @Override
 8     public String get2DExtensions() {
 9       return DefsMaker.getDefinitions(RemoveRotation.class);   }
10     @Override
11     public String get2DModels() {
12       return DefsMaker.getDefinitions(DoubleSnakeModel.class);   }
13   }
```

## 5.3. Dual active contour in action

The implemented dual active contour plugin was tested on the human cell images. In general it performed well, passing a local noise, even when its intensity was comparable to the segmented object.

One example (Fig. 7) shows a case more difficult to segment. On the image *a)* two contours are initialized. After independent evolution they reach their local minima – *b)*. Then the inner contour, as worse placed, is given the additional spring

energy and grows considerably – *c)*. After decreasing its energy the additional energy is removed and they evolve again independently until a next equilibrium – *d)*. In this configuration the outer contour has higher energy and it is given the spring energy. With it it comes very closely to the inner one – *e)*. Finally, both contours converge giving the correct segmentation – *f)*.

## 6. Conclusions and future work

In this paper we demonstrated the MESA environment capabilities to construct new segmention methods based on the deformable models (active contour – snake). Apart from assembling them from existing components, the operator can develop new ones. If modification of the method is moderate (in respect to the standard segmentation schema) it is sufficient to write few lines of the Java code in the embeded mode defining a new energy. It should be achievable even for people without high programming skills. When the modification goes beyond this standard schema, it is still possible to implement it with our system but it requires more programming and packaging work. It was shown that thanks to the system API flexibility even very deep modifications (e.g. two contours instead of one) are possible to achieve.

The MESA environment can be still developped to enhance possibilities of constructing new segmentations methods. One of the directions can be introducing new component classes/types, for example a one implementing the stop condition (now it has to be hardcoded in the model component). Another useful feature could be a cache memory storing values between the consecutive iterations. In the current state all the objects are discarded after each iteration and they are recreated for the next one – all the calculated information is lost. For example in the dual active contour model the first iteration is identified by analyzing the contour shape. One simple boolean variable persisting between the iterations could solve this problem easier. Some group of active contour techniques can be accelerated by precomputing necessary values. The gradient vector field [4] or the electric potential field [8] are good examples. These values could be stored in such a cache memory to be accessed from every iteration without recomputing.

## Acknowledgment

This work was supported with the European funds in the frame of the project "Information Platform TEWI" (Innovative Economy Programme).

## References

[1] Tsechpenakis, G., *Deformable Model-based Medical Image Segmentation*, In: Multi Modality State-of-the-Art Medical Image Segmentation and Registration Methodologies, Springer Publishing, 2004.

[2] Kass, M., Witkin, A., and Terzopoulos, D., *Snakes: Active Contour Models*, International Journal of Computer Vision, Vol. 1(4), 1988, pp. 321–331.

[3] Cohen, L., *On active contour models and balloons*, CVGIP: Image Underst., Vol. 53, 1991, pp. 211–218.

[4] Xu, C. and Prince, J. L., *Snakes, Shapes, and Gradient Vector Flow*, IEEE Transactions on Image Processing, Vol. 7, No. 3, 1998, pp. 359–369.

[5] Gunn, S. and Nixon, M., *A robust snake implementation; a dual active contour*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 19, 1997, pp. 63–68.

[6] Mcinerney, T. and Terzopoulos, D., *T-Snakes: Topology Adaptive Snakes*, Medical Image Analysis, 1999, pp. 840–845.

[7] Sethian, J., *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*, Cambridge University Press, 1999.

[8] Jalba, A., Wilkinson, M., and Roerdink, J., *CPM: A Deformable Model for Shape Recovery and Segmentation Based on Charged Particles*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 26, No. 10, 2004, pp. 1320–1335.

[9] Reska, D., Jurczuk, K., Boldak, C., and Kretowski, M., *MESA: MEdical Segmentation Arena environment*, http://mesa.wi.pb.edu.pl/.

[10] Reska, D., Boldak, C., and Kretowski, M., *A distributed approach for development of deformable model-based segmentation methods (accepted)*, Image Processing and Communications, 2013.