# BAB IV
# IMPLEMENTASI

Pada bab ini akan dibahas mengenai implementasi yang dilakukan berdasarkan rancangan yang telah dijabarkan pada bab sebelumnya. Implementasi kode program dilakukan menggunakan bahasa Python.

## 4.1 Lingkungan Implementasi

Spesifikasi perangkat keras dan perangkat lunak yang digunakan dalam implementasi ini ditampilkan pada Tabel 4.1.

Tabel 4.1 Lingkungan Perancangan Perangkat Lunak

| Perangkat | Spesifikasi |
|---|---|
| Perangkat keras | Prosesor: Intel® Core™ i3-2350M CPU @ 2.30GHz 2.30GHz<br>Memori: 4.00 GB |
| Perangkat lunak | Sistem Operasi:<br>Microsoft Windows 8 64-bit Pro<br>Perangkat Pengembang:<br>PyCharm<br>Perangkat Pembantu:<br>Microsoft Excel 2013 |

## 4.2 Implementasi

Sub bab implementasi ini menjelaskan tentang implementasi proses yang sudah dijelaskan pada bab desain perangkat lunak.

### 4.2.1 Implementasi Tahap Reduksi Size Frame

Sub bab ini membahas implementasi tahap reduksi *size frame*. Pada tahap ini data masukan berupa *frame* dan data keluaran yang dihasilkan pada tahap ini adalah *frame* yang telah tereduksi kualitasnya. Implementasi dilakukan dengan

menggunakan fungsi yang sudah disediakan oleh OpenCV yaitu *pyrDown()* dan di tunjukkan oleh Kode Sumber 4.1.

```
1.  currentFrame2 = copy.copy(currentFrame)
2.  currentFrame =
    ImageProcessing.getDownSize(currentFrame)
```

Kode Sumber 4.1 Implementasi Tahap Reduksi *Size Frame*

```
1.  def getDownSize(self,image):
2.          return cv2.pyrDown(image)
```

Kode Sumber 4.2 Penggunaan Fungsi pyrDown()

Kode Sumber 4.2 melakukan reduksi dengan memanggil fungsi *getDownSize()* dam disimpan kedalam variabel *currentFrame*. Variabel *currentFrame2* akan digunakan sebagai masukan dari ekstraksi fitur dan *frame* keluaran yang ditampilkan, sedangkan proses lainnya akan menggunakan variabel *currentFrame*.

### 4.2.2 Implementasi Tahap Deteksi Gerak

Sub bab ini membahas implementasi tahap deteksi gerak piksel. Masukan dari tahap ini adalah *frame* yang sedang diproses. Implementasi dilakukan menggunakan fungsi yang sudah disediakan oleh OpenCV yaitu *BackgroundSubtractorMOG()*. Implementasi ditunjukkan oleh Kode Sumber 4.3.

```
1.  def getMovingForeGround(self, image):
2.     self.BckgrSbsMOG =
    cv2.BackgroundSubtractorMOG()
3.     return
    self.BckgrSbsMOG.apply(image,learningRate =
    0.0005)
```

Kode Sumber 4.3 Implementasi Tahap Deteksi Gerak

Hasil yang dikembalikan oleh fungsi *getMovingForeGround()* adalah gambar dengan nilai tiap pikselnya antara 0 atau 255 dengan. Dimana nilai 255 adalah

nilai piksel yang bergerak terhadap piksel pada *frame* sebelumnya. Hasil dari fungsi *getMovingForeGround()* tidak secara langsung diproses kedalam tahap selanjutnya. Dari hasil *frame* ini akan diambil indeks piksel yang bergerak, implementasi ditunjukkan oleh Kode Sumber 4.4.

```
1.  def getMovingCandidatePiksel(self,
    moving_frame):
2.      listY,listX = np.where( moving_frame ==
    255 )
3.      return np.vstack((listY,listX))
```

Kode Sumber 4.4 Implementasi Penyimpanan Piksel

Fungsi *getMovingCandidatePiksel()* melakukan iterasi untuk mendapatkan piksel-piksel yang bernilai 255. Fungsi ini mengembalikan hasil berupa *list* indeks piksel yang bergerak. Hasil keluaran dari keseluruhan proses deteksi gerak adalah *list* yang berisi indeks piksel y dan x.

### 4.2.3 Implementasi Tahap Deteksi Warna Piksel

Deteksi warna piksel dilakukan dengan pengecekan warna setiap piksel pada *list* piksel yang dihasilkan melalui proses deteksi gerak. Masukan dari tahap ini adalah *list* piksel yang telah didapatkan pada tahap deteksi gerak dan *frame* dengan *channel* R,G,B. Sebelum menghitung probabilitas warna piksel, terlebih dahulu dilakukan perhitungan mencari nilai rata-rata dan standar deviasi untuk nilai piksel R,G,B. Sebelas gambar api disimpan dan dilakukan proses perhitungan nilai rata-rata dan nilai standar deviasi untuk setiap *channel*. Pada implementasi sistem, proses menghitung nilai rata-rata dan standar deviasi setiap *channel* dapat dilihat pada Kode Sumber 4.5.

```
1.  def getStdDevAndMean(self,path):
2.      list_file = File.readFolder(self,path)
3.      R = []
4.      G = []
5.      B = []
```

```
6.      for x in list_file:
7.          image =
    ImageProcessing.readImage(self,path+'/'+x)
8.          r = np.array(image[:,:,2]).ravel()
9.          g = np.array(image[:,:,1]).ravel()
10.         b = np.array(image[:,:,0]).ravel()
11.         R += (r.tolist())
12.         G += (g.tolist())
13.         B += (b.tolist())
14.     mean = []
15.     mean.append(np.average(B))
16.     mean.append(np.average(G))
17.     mean.append(np.average(R))
18.     standard_deviasi = []
19.     standard_deviasi.append(np.std(B))
20.     standard_deviasi.append(np.std(G))
21.     standard_deviasi.append(np.std(R))
22.     return standard_deviasi, mean
```

Kode Sumber 4.5 Implementasi Menghitung Nilai Standar Deviasi dan Rata-Rata Setiap *Channel*

Pada Kode Sumber 4.5 setiap gambar *dataset* dibaca dan dilakukan pemisahan setiap *channel*. Selanjutnya, dilakukan perhitungan rata-rata dan perhitungan standar deviasi setiap *channel*. Setelah didapatkan nilai rata-rata dan standar deviasi setiap *channel*, proses penentuan kandidat api menggunakan metode probabilitas warna piksel dilakukan. Setiap piksel dihitung nilai probabilitas *channel* R,G,B. Selanjutnya, menghitung nilai probabilitas piksel dengan mengalikan nilai probabilitas R,G,B. Proses perhitungan probabilitas piksel memakan waktu yang cukup lama jika menghitung setiap kemungkinan kandidat piksel api yang lolos pada tahap deteksi gerak. Pada implementasi, nilai probabilitas tiap piksel yang masuk kedalam probabilitas warna api dimasukan kedalam file. Nilai yang disimpan adalah nilai R,G,B piksel yang termasuk kedalam warna piksel api. Hal ini dilakukan karena proses perhitungan yang lama jika menghitungan probabilitas setiap piksel. Isi dari file adalah *list* kemungkinan piksel-piksel yang masuk dalam probabilitas

warna api. Proses *generate list* piksel api dapat dilihat pada Kode Sumber 4.6.

```
1.  def getListColorPiksel
    (self,list_standard_deviasi, list_mean):
2.      res = []
3.      threshold = self.getThreshold()
4.      for B in range(0,256):
5.          for G in range(B,256):
6.              for R in xrange(G,256):
7.                  if
8.  Data.getGaussianProbability(self,B,
    list_standard_deviasi[0], list_mean[0])*
    Data.getGaussianProbability(self,G,
    list_standard_deviasi[1], list_mean[1])*
    Data.getGaussianProbability(self,R,
    list_standard_deviasi[2], list_mean[2]) >
    threshold:
9.                      res.append([B,G,R])
10. return res
```

Kode Sumber 4.6 *Generate list* piksel api

Kode Sumber 4.6 melakukan iterasi pengecekan piksel. Iterasi tidak dilakukan sebanyak kombinasi nilai R,G,B. hal ini dikarenakan nilai *channel* R dari piksel api lebih besar dari nilai *channel* G dan nilai *channel* G lebih besar dari nilai *channel* B [10]. Pada Kode Sumber 4.6, dilakukan pemanggilan fungsi *getGaussianProbability()*. Fungsi *getGaussianProbability()* digunakan untuk menghitung nilai probabilitas tiap *channel* piksel. Fungsi tersebut mengimplementasikan rumus dari probabilitas distribusi gaussian. Implementasi fungsi *getGaussianProbability()* dapat dilihat pada Kode Sumber 4.7.

```
1.  def getGaussianProbability(self, data,
    standard_deviasi, mean):
2.      data = float(data)
3.      standard_deviasi =
    float(standard_deviasi)
4.      mean = float(mean)
5.      result = pow((data-mean),2)
```

```
6.      div = 2*pow(standard_deviasi,2)
7.      exp = np.exp(-result/div)
8.      result =
    standard_deviasi*np.sqrt(2*np.pi)
9.      result = 1/result
10      return result* exp
```

Kode Sumber 4.7 Fungsi Menghitung Nilai Probabilitas Distribusi Gaussian

```
1.  def getThreshold(self):
2.      return 5*pow(10,-9)
```

Kode Sumber 4.8 Mendapatkan Threshold

Kode Sumber 4.7 melakukan perhitungan probabilitas piksel api setiap *channel*. Nilai *threshold* pada Kode Sumber 4.8 didapatkan dari hasil analisa yang telah dilakukan. Ketika program dijalankan, program akan membaca file yang berisi *list* piksel yang sudah di *generate* sebelumnya dan menyimpan nilai piksel tersebut kedalam *array*. Proses pembacaan file bisa dilihat pada Kode Sumber 4.9.

```
1.  def getFireArray(self,path):
2.      lists = [[[False for k in xrange(256)]
    for j in xrange(256)]for i in xrange(256)]
3.      data = open(path,'r')
4.      for x in data:
5.          color = (x.split('\n')[0]).split(' ')
6.          lists[int(color[0])]
    [int(color[1])][int(color[2])] = True
7.      return lists
```

Kode Sumber 4.9 Membaca *List* Piksel Api

Pada Kode Sumber 4.9 dilakukan peroses pembuatan *array* tiga dimensi, dengan *default* *False* pada nilai indeks. Pada *line* enam, dilakukan perubahan nilai sesuai indeks yang ada pada *list* dengan mengubah nilai *array* tiga dimensi tersebut menjadi *True*. Nilai indeks dimensi *array* tersebut mewakilkan

indeks R,G,B. Setelah didapatkan *array* dengan indeks yang mewakili nilai R,G,B dan hasilnya, dilakukan pengecekan terhadap kandidat piksel yang didapatkan pada proses deteksi gerak. Implementasi proses pengecekan kandidat piksel dapat dilihat pada Kode Sumber 4.10.

```
1.  def getColorCandidatePiksel(self,
2.  list_standard_deviasi, list_mean):
3.      true_piksel = []
4.      false_piksel = []
5.      for x in range(0,len(list_candidate[0])):
6.          data = image[list_candidate[0][x]]
    [list_candidate[1][x]]
7.          B,G,R = data[0],data[1],data[2]
8.          if color_dataset[B][G][R] == True:
9.  true_piksel.append([list_candidate[0][x],
    list_candidate[1][x]])
10.         else :
11. false_piksel.append([list_candidate[0][x],
    list_candidate[1][x]])
12.     return true_piksel,false_piksel
```

Kode Sumber 4.10 Implementasi Tahap Deteksi Warna Piksel

Hasil yang dikeluarkan pada tahap ini adalah *list* nilai indeks piksel yang masuk kedalam piksel api.

### 4.2.4 Implementasi Tahap Region Growing

Sub bab ini membahas implementasi tahap *region growing* yang menggunakan kandidat piksel api pada tahap deteksi warna api sebagai masukan. *Region growing* dilakukan menggunakan *list* kandidat piksel api sebagai masukan awal dari piksel yang dicari *region*nya. Titik piksel kandidat api dilakukan pengecekan probabilitas warna api terhadap delapan tetangga piksel tersebut. Jika piksel tetangga termasuk piksel api, maka piksel tersebut akan ditandai sebagai region dari piksel awalan tersebut. *Region* akan diberi nomor sesuai dengan titik piksel awal dari *region* tersebut. Implementasi *region growing* dapat dilihat pada Kode Sumber 4.11.

```
1.  def getRegionGrowing(self, list_candidate,
    images, color_dataset,counter):
2.     gray_image =
    ImageProcessing.getRGBtoGray(self,images)
3.     is_visit = gray_image*0
4.     result_image = copy.copy(gray_image)
5.     region_number = 0
6.     for x in list_candidate:
7.        coor_y = x[0]
8.        coor_x = x[1]
9.        if is_visit[coor_y][coor_x] == 0:
10.          stack = []
11.          region_number+=1
12.          stack.append([coor_y,coor_x])
13.          is_visit[coor_y][coor_x] =
    region_number
             result_image, is_visit =
14. self.doFloodFill( gray_image, result_image,
    is_visit, stack, region_number,
    color_dataset, images)
16.    return is_visit
```

Kode Sumber 4.11 Implementasi Tahap *Region Growing*

Kode Sumber 4.11 melakukan inisialisasi *region* dengan nilai 0 pada variabel *is_visit*. Selanjutnya melakukan iterasi sebanyak kandidat piksel api yang masuk kedalam tahap ini. Setiap kandidat piksel api dilakukan pengecekan, apakah piksel tersebut sudah dilakukan *region growing* atau belum. Jika belum (nilai variabel indeks yang sedang dicek bernilai 0) , maka koordinat dari titik tersebut akan dijadikan sebagai *seed* untuk dimasukan kedalam *stack* dan dilakukan *growing*. Implementasi *growing* dapat dilihat pada Kode Sumber 4.12.

Kode Sumber 4.12 melakukan pegecekan terhadap *stack* piksel yang akan dicek. Jika stack masih memiliki nilai, maka nilai tersebut akan digunakan sebagai *seed* untuk melakukan cek terhadap tetangga piksel. Jika tetangga piksel belum mempunyai *region* dan masuk kedalam *region* tersebut, maka nilai dari variabel *is_visit* diganti sesuai dengan nilai *region* dan koordinat piksel tersebut dimasukan kedalam *stack* untuk melakukan pengecekan tetangga selanjutnya. Pada fungsi

*growing()* melakukan pemenggilan fungsi *getClockWise()*. Implementasi *getClockWise()* dapat dilihat pada Kode Sumber 4.12.

```
1.  def growing(self,
    gray_image ,result_image ,is_visit, stack,
    region_number, color_dataset,
    original_image):
2.      clocks = Data.getClockwise(self)
3.      while len(stack) != 0:
4.          coory,coorx = stack[0]
5.          stack.pop(0)
6.          result_image[coory][coorx] = 255
7.          data = original_image[coory][coorx]
8.          B,G,R = data[0],data[1],data[2]
9.          for x in clocks:
10.             try :
11.                 if
12.  is_visit[coory+x[0]][coorx+x[1]] == 0 and
     color_dataset[B][G][R] == True :
                        is_visit[coory+x[0]]
13.  [coorx+x[1]] = region_number
14.                     stack.append([coory+x[0],
15.  coorx+x[1]]
                except :
16.                 pass
17.     return result_image,is_visit
```

Kode Sumber 4.12 Implementasi *Growing*

```
1.  def getClockwise(self):
2.      clocks = []
3.      clocks.append([-1,-1])
4.      clocks.append([-1,0])
5.      clocks.append([-1,1])
6.      clocks.append([0,-1])
7.      clocks.append([0,1])
8.      clocks.append([1,-1])
9.      clocks.append([1,0])
10.     clocks.append([1,1])
11.     return clocks
```

Kode Sumber 4.13 Implementasi Tahap *Clock Wise*

Iterasi yang dilakukan pada *getClockWise()* akan melakukan pengecekan delapan tetangga searah jarum jam. Hasil keluaran dair ptoses ini adalah *region* yang mempunyai nomor *region* yang berbeda antar *region*.

### 4.2.5 Implementasi Tahap Perhitungan Luasan Region

Sub bab ini membahas implementasi tahap perhitungan luasan *region*. Masukan dari tahap ini adalah kandidat piksel api yang telah didapatkan dari tahap probabilitas warna api, dan *region*. Pada tahap perhitungan luasan *region* dihitung banyaknya piksel yang ada pada *region* tersebut. Jika luasan piksel *region* tersebut melebihi batas, maka kandidat piksel yang ada pada *region* tersebut masuk sebagai kandidat piksel api . Implementasi dapat dilihat pada Kode Sumber 4.14.

```
1.  def getFilterSizeRegion
    (self,list_candidate,region):
2.      true_piksel = []
3.      false_piksel = []
4.      list_region = np.unique(region)
5.      threshold = dict()
6.      for x in range(1,len(list_region)):
7.          lists = np.where(region == x)
8.          if len(lists[0]) > 1*len(region)*
    len(region[0])/100:
9.              threshold[x] = True
10.         else :
11.             threshold[x] = False
12.     for x in list_candidate:
13.         coor_y = x[0]
14.         coor_x = x[1]
15.     if threshold[region[coor_y][coor_x]] ==
    True:
16.         true_piksel.append([coor_y,coor_x])
17.     else :
18.         false_piksel.append([coor_y,coor_x])
19.     return true_piksel,false_piksel
```

Kode Sumber 4.14 Implementasi Tahap Variasi Warna *Region*

Setiap *region* akan dilakukan iterasi dan dilakukan pengecekan. Jika luasan dari *region* tersebut memenuhi syarat, maka seluruh piksel kandidat api masuk kedalam proses berikutnya. Hasil keluaran dari fungsi ini adalah *list* piksel yang yang masuk kedalam kandidat piksel api.

### 4.2.6 Implementasi Tahap Ekstraksi Fitur dengan Wavelet

Sub bab ini membahas implementasi tahap ekstraksi fitur. Ekstraksi fitur dilakukan dengan mengubah *frame* kedalam domain *wavelet*. Digunakan sepuluh buah *frame* yang berurutan untuk diubah kedalam domain *wavelet*. Pada implementasi setiap *frame* diubah kedalam domain *wavelet* dan disimpan kedalam *list*. Implementasi mengubah dan manyimpan *frame* domain *wavelet* dapat dilihat pada Kode Sumber 4.15.

```
1.  LL,(HL,LH,HH) =
    wv.toWavelet(copy.copy(grayImage))
2.  list_wavelet.append([HL,LH,HH])
```

Kode Sumber 4.15 Implementasi Memasukan Nilai *Wavelet* Kedalam *List*

Pada Kode Sumber 4.16, fungsi *toWavelet()* digunakan untuk mengubah *frame* kedalam domain *wavelet*. Implementasi mengubah *frame* kedalam domain *wavelet* dapat dilihat pada Kode Sumber 4.16.

```
1.  def toWavelet(image):
2.      return pywt.dwt2(image,'db2')
```

Kode Sumber 4.16 Pemanggilan Fungsi *Wavelet*

### 4.2.7 Implementasi Tahap Klasifikasi

Sub bab ini membahas implementasi tahap klasifikasi. Pada tahap ini sistem diberi masukan kandidat piksel api dan sepuluh buah *frame* domain *wavelet*, dimana masing-masing

*frame* berisi tiga buah sub *frame* seperti penjelasan sub bab 3.4.1 . Sebelum melakukan klasifikasi, data training terlebih dahulu diproses sebagai data *training* untuk klasifikasi. Implementasi *training* klasifikasi dapat dilihat pada Kode Sumber 4.17.

```
1.  def getClassifier(datatraining):
2.      x,y = readDataSet(datatraining)
3.      clf = svm.SVC(kernel = 'rbf',C = 3.5)
        clf.fit(x,y)
        return clf
```

Kode Sumber 4.17 *Training* Klasifikasi

Setiap piksel pada kandidat piksel api dilakukan perhitungan nilai fitur seperti yang sudah dijelaskan pada sub bab 3.4.1. Implementasi klasifikasi dapat dilihat pada Kode Sumber 4.18.

```
1.  def doClassification(classifier, list,
2.  wavelet):
3.      truePiksel = []
4.      falsePiksel = []
5.      listMax = []
6.      listMin = []
7.      cpyWavelet = np.int_(copy.copy(wavelet))
8.      cpyWavelet = np.power(cpyWavelet,2)
9.      for x in cpyWavelet:
10.         lists = np.add(np.add(x[0],x[1]) ,
    x[2])
11.         listMax.append(np.max(lists))
12.         listMin.append(np.min(lists))
13.     for x in list:
14.         data = []
15.         cnt= 0
16.         for y in wavelet:
                res = pow(y[0][x[0]][x[1]],2) +
17. pow(y[1][x[0]][x[1]],2) +
    pow(y[2][x[0]][x[1]],2)
            res = (float(res)-
18. float(listMin[cnt]))/(float(listMax[cnt])-
    float(listMin[cnt]))
```

```
            res = float('%.2f' % res)
19.         cnt+=1
20.         data.append(res)
21.       data = np.sort(data)
22.       classes = classifier.predict(data)
23.       if classes == 'Api':
24.           truePiksel.append([x[0],x[1]])
25.       else :
26.           falsePiksel.append([x[0],x[1]])
27.   return truePiksel,falsePiksel
28.
```

Kode Sumber 4.18 Implementasi Tahap Klasifikasi

Pada Kode Sumber 4.18 dilakukan iterasi sebanyak kandidat piksel api yang lolos ketahap verifikasi. Setiap piksel akan dihitung nilai fitur *wavelet*. Dilakukan normalisasi nilai fitur yang dilakukan pada *line* 18. Jika hasil klasifikasi suatu fitur adalah api, maka nilai indeks dari piksel tersebut akan dimasukan kedalam *list*. Hasil yang dikeluarkan dari proses ini adalah *list* piksel api yang lolos tahap verifikasi.

### 4.2.8 Implementasi Tahap Menandai Region Api

Sub bab ini membahas implementasi menandai *region* api. Piksel api yang sudah lolos tahap verifikasi selenjutnya diproses sebagai data keluaran yang ditampilkan. Masukan dari tahap ini adalah kandidat piksel api dan *frame* dari variabel *currentFrame2*. Karena perbedaan ukuran antara indeks piksel api, dilakukan normalisasi indeks. Dilakukan penyesuaian indeks-indeks piksel dengan *frame* keluaran. Implementasi *training* klasifikasi dapat dilihat pada Kode Sumber 4.19.

```
1.  def markingFire(self, list_fire, image,
    constanta):
2.      if len(list_fire) == 0:
3.          return image
4.      list = np.array(list_fire)
5.      min_y,max_y =
    min(list[:,0]),max(list[:,0])
```

```
6.      min_x,max_x =
    min(list[:,1]),max(list[:,1])
7.      distance_y = int((max_y-
    min_y)/2)*constanta
8.      distance_x = int((max_x-
    min_x)/2)*constanta
9.      center_point =
    [int((max_y+min_y)*constanta/2) ,
    int((max_x+min_x)*constanta/2)]
10.     min_y,min_x,max_y,max_x =
    center_point[0]-distance_y, center_point[1]
    - distance_x, center_point[0] + distance_y,
    center_point[1] +distance_x
11.     for y in range(min_y,max_y+1):
12.         image[y][min_x] = [255,191,0]
13.         image[y][max_x] = [255,191,0]
14.     for x in range(min_x,max_x+1):
15.         image[min_y][x] = [255,191,0]
16.         image[max_y][x] = [255,191,0]
17.     return image
```

Kode Sumber 4.19 Implementasi Tahap Menandai Region Api

Hasil keluaran dari Kode Sumber 4.19 adalah *frame* dengan tanda persegi jika terdapat piksel api pada *frame* yang diproses.