Particle Filter Localization for Quadcopters Using Augemented Reality Tags

Edward Francis Kelley V

Princeton University
Department of Computer Science

Advisors:
Professor Szymon Rusinkiewicz
Professor Robert Stengel

May 2013

Abstract

This is my abstract.

Acknowledgements

I want to thank me.

To my parents. Thanks for the whole tuition thing.

Contents

List of Figures

Chapter 1

Introduction

Talk about the increasing use of quadcopters for a variety of uses.

Advantages of using quadcopters, disadvantages/difficulties.

Transition to the difficulties

1.1 Motivation

1.2 Current Acquisition Methods

1.2.1 Laser Scanners

Laser rangefinder technology is the “gold standard” of 3D model acquisition in terms of quality. Modern scanners can produce sub millimeter accuracy, which make them a great choice for detailed digitization of statues. Combined with high-resolution photograph texture-mapping, very few techniques can match the precision and quality of these scans. The Digital Michelangelo Project showed the power and precision of laser scanners by scanning several different statues, including Michelangelo’s David, to 1/4mm accuracy.[13]

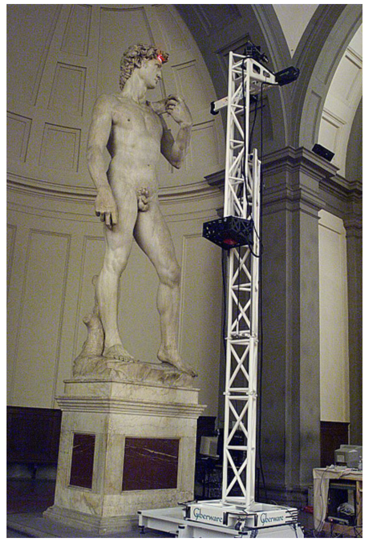

Figure 1.1: An example of a laser scanner setup used by the Digital Michelangelo Project [13].

However, laser scanners do have several drawbacks. The equipment is extremely expensive, bulky, and fragile. The Michelangelo Project had to transport over 4 tons of equipment to Italy in order to produce their scans. Additionally, laser scans involve immense setup and can take many hours. The scan of David took over a thousand man-hours to scan and even more than that in post processing [13].

Multi-View Stereo

Multi-view stereo uses a collection of 2D images to reconstruct a 3D object model. By viewing a single object from hundreds of different camera positions, a 3D model can be generated. Although this technique originally required precisely known camera coordinates, recent algorithms can produce a 3D model from an unordered collection of images with unknown camera positions, assuming that there is sufficient coverage. Existing software packages such as Bundler and Agisoft Photoscan can produce high-quality 3D reconstructions using these unordered image collections. [5][4]

The ability to use a collection of images without precise camera position information means that these 3D objects can be modeled substantially faster than with

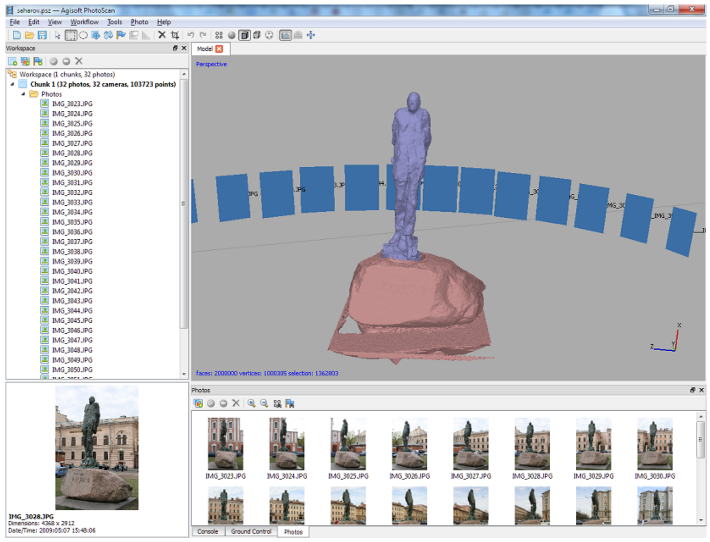

Figure 1.2: A 3D model of a statue generated by Agisoft Photoscan. Notice the derived camera planes encompassing the statue [4].

a laser scanner. With a smaller object, it is a relatively simple process to take pictures of the object from many different angles. However, for a larger object, such as a statue or building, the problem of gathering imagery becomes substantially more difficult.

1.2.2 Problem Definition

We look to create a system to capture imagery of large 3D objects for use in multi-view stereo software. This system has several requirements.

1. **Low Cost**

   The system should be substantially cheaper than laser scanning.

2. **Easy to Use**

   This system should be able to be deployed by users with minimal training. Additionally, the hardware should be off-the-shelf and easily accessible.

3. **Complete Coverage**

The system must be able to capture images from a wide variety of positions, completely covering every part of the target object.

4. **High Quality Imagery**

   The system must produce sufficiently high resolution, non-blurry images for use in multi-view stereo software.

1.2.3 Proposed Solution

We propose using low-cost autonomous quadcopters to gather imagery needed for use in multi-view stereo software. By flying a quadcopter with a mounted camera around the target object, we can quickly and thoroughly capture images of the target from a wide variety of positions. Using quadcopters has many advantages.

1. Quadcopters can capture images from positions unreachable by ground-based cameras.
2. By methodically flying around the target object at different altitudes, we can guarantee complete coverage of the target object.
3. The imagery can be captured very quickly, on the order of a few minutes.
4. Quadcopters are small, portable, and easily deployable.

Chapter 2

Related Work

2.1 Quadcopters for Model Acquisition

The past decade has seen a huge increase in the use of quadcopters for a variety of applications. With the improvement of stabilization software, quadcopters have seen a rise in popularity as a stable, cheap, and highly maneuverable aerial platform.

Although a relatively new field, several research groups have studied the use of quadcopters in 3D model construction. Irschara et al. created a system to generate 3D models using images taken from UAVs. While a quadcopter was used for gathering imagery, the quadcopter was manually controlled and the main focus of their work was photogrammetry-based model creation [11]. Steffen et al. studied surface reconstruction using aerial photography captured by UAVs [16].

Most relevant to our work, Engel et al. published multiple papers on the camera-based navigation and localization of the AR.Drone. While they were able to acheive very accurate navigation, their work relies on the drone facing a mostly planar surface during the entire flight, a constraint that is not possible in our application.

2.2 GPS-denied Navigation of Quadcopters

Chapter 3

System Design

3.1 Parrot AR.Drone 2.0

3.2 System Architecture

3.2.1 Robot Operating System

3.2.2 ARDrone Autonomy

3.2.3 AR Toolkit

3.2.4 Localization

3.2.5 Controller

3.2.6 Agisoft Photoscan

Chapter 4

Localization

4.1 Problem Description

4.2 Potential Solutions

4.3 Particle Filter

4.3.1 Propagation Step

4.3.2 Correction Step

4.3.3 Correction Using Augmented Reality Tags

Chapter 5

Controller

5.1 Problem Description

5.2 Design

Chapter 6

Results

6.1 Localization

6.2 Controller

6.3 Model Generation

Chapter 7

Conclusion

Appendix A

Implementation

```
#!/usr/bin/env python

#What of this can I get rid of?

import roslib; roslib.load_manifest('quadcopterCode')
import rospy
import tf

# Import the messages we're interested in sending and receiving
from geometry_msgs.msg import Twist     # for sending commands to the drone
from std_msgs.msg import Empty          # for land/takeoff/emergency
from ardrone_autonomy.msg import Navdata # for receiving navdata feedback
from visualization_msgs.msg import *

# An enumeration of Drone Statuses
from drone_status import DroneStatus
from basic_commands import BasicCommands
from keyboard_controller import *
from drone_video_display import DroneVideoDisplay
```

```
from waypoints import waypoints
from localize import *
from particlefilter import *

from math import *
from time import *


LINEAR_ERROR = 200 #mm

ANGULAR_ERROR = 5 #degrees

LINEAR_MAX = .3 #Max tilt amount (unitless)

ANGULAR_MAX = .2 #Max turn amount (unitless)

LINEAR_GAIN = .1 # Pick good values
ANGULAR_GAIN = .1 #




class DroneController(DroneVideoDisplay):
  def __init__(self, cmd):
    # self.cmd = BasicCommands()
    self.localize = localize()
    self.pose = particle(self)
    self.start = True
    self.start_time = time()
    self.last_time = time()
    self.steps = 1
    self.br = tf.TransformBroadcaster()

```

```python
        self.cmd = cmd

        self.waypoints = waypoints("/home/ekelley/ros_workspace/sandbox/QuadcopterMapping/quadcopterCode/data/waypoints.txt")

        self.current_waypoint = self.waypoints.get_waypoint()

        flag = raw_input("Start?")


    def get_distance(self):
        return sqrt((self.current_waypoint.x - self.pose.x)**2 + (self.current_waypoint.y - self.pose.y)**2 + (self.current_waypoint.y - self.pose.y)**2)

    def get_angle_diff(self):
        return abs(self.clamp_angle(self.current_waypoint.theta - self.pose.theta))

    def clamp_angle(self, angle):
        if (angle > 180):
            return angle - 360
        elif (angle < -180):
            return angle + 360
        else:
            return angle

    def listener(self):
        rospy.Subscriber("/ardrone/navdata", Navdata, self.update_command)
        rospy.Subscriber("/visualization_marker", Marker, self.got_marker)
        # spin() simply keeps python from exiting until this node is stopped
        rospy.spin()

```

```
def got_marker(self, data):
  self.localize.ar_correct(data)

def update_command(self, data):
  self.last_time = time()
  self.localize.update(data)
  self.pose = self.localize.estimate()
  self.br.sendTransform((0, 0, 0), tf.transformations.
      quaternion_from_euler(0, 0, 0), rospy.Time(0), "/ardrone/
      ardrone_base_link", "/world")

  distance = self.get_distance()
  angle = self.get_angle_diff()

  #If it hit the target, move on
  if ((distance < LINEAR_ERROR) and (angle < ANGULAR_ERROR)):
    self.current_waypoint = self.waypoints.get_waypoint()

  x_diff = (self.current_waypoint.x - self.pose.x)
  y_diff = (self.current_waypoint.y - self.pose.y)
  z_diff = (self.current_waypoint.z - self.pose.z)

  # print("Elapsed time: %f" % (time() - self.last_time))
  avg = (time() - self.start_time)/self.steps
  # print("Average time: %f" % (avg))
  self.steps += 1
  #Define tilt as being within - and + MAX values

  if ((time() - self.start_time > 5) and (time() - self.start_time <
      10)):
    print("TURNING")
    self.cmd.SetCommand(roll=0,pitch=0,yaw_velocity=.1,z_velocity=0)
  else:
```

```
            self.cmd.SetCommand(roll=0,pitch=0,yaw_velocity=.1,z_velocity=0)
        #SetCommand
        # if (self.start):
        #     self.cmd.SendTakeoff()
        #     self.start = false
        # elif (self.status == DroneStatus.Flying or self.status == DroneStatus.GotoHover or self.status == DroneStatus.Hovering):
        #     self.cmd.SetCommand(roll=0,pitch=0,yaw_velocity=0,z_velocity=0)

def main():
    rospy.init_node("controller")
    cmd = BasicCommands()
    controller = DroneController(cmd)
    run = raw_input("Press any key to run:")

    print "Running"
    controller.listener()

if __name__ == '__main__':
    main()
```

```
#!/usr/bin/env python

import sys
import time
import math
#ROS related initializations
import roslib
import rospy
import os
from std_msgs.msg import String, Float32
from particlefilter import *
from visualization_msgs.msg import *
```

```python

#SHOULD BE PUBLISHING A TRANSFORM
class localize:
  def __init__(self):
    self.time = time.time()
    self.pf = particlefilter()

  #NEED TO ADD HEADING INFORMATION
  def update(self, data):
    #Get delta t and update tm
    delta_t = time.time() - self.time #Time in seconds
    self.time = time.time()
    self.pf.propogate(delta_t, data.ax, data.ay, data.az, data.rotX, data.rotY, data.rotZ)
    self.pf.correct(delta_t, data.vx, data.vy, data.altd, data.magX, data.magY, data.magZ)

  def ar_correct(self, data):
    self.pf.ar_correct(data)

  def estimate(self):
    return self.pf.est


if __name__ == "__main__":
  #start the class
  print("Running localize")
  localize_1 = localize()
  localize_1.listener()
```

```python
#!/usr/bin/env python

import sys
```

```
import time
import random
from math import *
import numpy
#ROS related initializations
import os
import roslib
import rospy
from walkerrandom import *
from std_msgs.msg import *
from visualization_msgs.msg import *
from geometry_msgs.msg import *
from tf import *
from tf.transformations import *
from tf import TransformerROS

#Coordinate frame info
# -linear.x: move backward
# +linear.x: move forward
# -linear.y: move right
# +linear.y: move left
# -linear.z: move down
# +linear.z: move up

# -angular.z: turn left
# +angular.z: turn right

class particlefilter:
  def __init__(self, num_particles=100, vis_noise=10, ultra_noise=100, mag_noise=37, linear_noise=.002, angular_noise=2.3):
    print("Starting a particle filer with %d particles" % num_particles)
    self.filename = "/home/ekelley/Dropbox/thesis_data/" + time.strftime('%Y_%m_%d_%H_%M_%S') #in the format YYYYMMDDHHMMSS
```

```
        self.fp = open(self.filename + ".txt", "w")
        self.fp_part = open(self.filename+"_part.txt", "w")
        self.fp_ar = open(self.filename+"_ar.txt", "w")
        self.num_particles = num_particles


        self.vis_noise = vis_noise
        self.ultra_noise = ultra_noise
        self.mag_noise = mag_noise
        self.linear_noise = linear_noise
        self.angular_noise = angular_noise


        self.start_mag_heading = 0
        self.start_gyr_heading = 0
        self.gyr_theta = 0
        self.particle_list = []
        self.weight_dict = dict()
        self.first_propogate = True
        self.first_correct = True
        self.step = 0
        self.est = particle(self)
        self.acc_est = particle(self) #Estimation using acc and gyr
        self.vis_est = particle(self) #Estimation using vis odometry and ultrasound
        for i in range(num_particles):
            self.particle_list.append(particle(self))

        self.fp.write("self.step,delta_t,x_acc,y_acc,z_acc,gyr_theta_est,rotX,rotY,delta_theta,self.acc_est.x,self.acc_est.y,self.acc_est.z,self.acc_est.theta,x_vel,y_vel,z_est,magX,magY,magZ,mag_theta_est,new_x,new_y,self.vis_est.x,self.vis_est.y,self.est.x,self.est.y,self.est.theta\n")
```

```
        self.est_pose = Point()
        self.line = Marker()
        self.est_pub = rospy.Publisher('pf_pose', Point)
        self.listener = TransformListener()
        self.transformer = TransformerROS()
        self.update_marker()

    #Propogate particles based on accelerometer data and gyroscope-based theta
    def propogate(self, delta_t, x_acc, y_acc, z_acc, rotX, rotY, rotZ):
        if (self.first_propogate):
            self.start_gyr_heading = rotZ

        #PROPOGATE USING THE AVERAGE OF EST.THETA AND THETA_EST-PREV_THETA

        delta_theta = self.clamp_angle((rotZ- self.start_gyr_heading) - self.acc_est.theta) #Should I be using self.est.theta instead of prev_theta?

        for particle in self.particle_list:
            particle.propogate(delta_t, self.convert_g(x_acc), self.convert_g(y_acc), self.convert_g(z_acc), rotX, rotY, delta_theta, True)

        #Propogate estimate based on magnetometer. for testing
        self.acc_est.propogate(delta_t, self.convert_g(x_acc), self.convert_g(y_acc), self.convert_g(z_acc), rotX, rotY, delta_theta, False)

        self.fp.write("%d,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f," % (self.step, delta_t, x_acc, y_acc, z_acc, rotZ, rotX, rotY, delta_theta, self.acc_est.x, self.acc_est.y, self.acc_est.z, self.acc_est.theta))

```

```
  def convert_g(self, acc):
    g_to_mmss = 9806.65
    return acc*g_to_mmss


  #Correct particles based on visual odometry and magnetometer readings
  def correct(self, delta_t, x_vel, y_vel, z_est, magX, magY, magZ):
    if (self.first_correct):
      self.start_mag_heading = self.get_heading(magX, magY, magZ)

    mag_theta_est = self.clamp_angle(self.get_heading(magX, magY, magY)-self.start_mag_heading)
    x_delta = (x_vel*cos(radians(mag_theta_est)) - y_vel*sin(radians(mag_theta_est)))*delta_t
    y_delta = (x_vel*sin(radians(mag_theta_est)) + y_vel*cos(radians(mag_theta_est)))*delta_t

    new_x = x_delta + self.est.x
    new_y = y_delta + self.est.y

    #Weight particles
    self.weight_particles(delta_t, new_x, new_y, z_est, mag_theta_est)

    #Create new set of particles
    self.particle_list = []

    #Initialize the random selector. Can select items in O(1) time
    wrand = walkerrandom(self.weight_dict.values(), self.weight_dict.keys())

    for i in range(self.num_particles):
      particle = wrand.random()
      self.particle_list.append(particle)
```

```
    self.vis_est.x += x_delta;
    self.vis_est.y += y_delta;
    self.vis_est.z = z_est;
    self.vis_est.theta = mag_theta_est

    self.fp.write("%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f\n" % (x_vel, y_vel, z_est, magX, magY, magZ, mag_theta_est, new_x, new_y, self.vis_est.x, self.vis_est.y, self.est.x, self.est.y, self.est.theta))
    self.step += 1
    self.estimate()
    self.update_marker()

  def ar_correct(self, marker):
    marker_id = marker.id
    pose = marker.pose

    pose_trans = ()
    pose_rot = ()
    #Is this necessary
    pose_trans = (pose.position.x, pose.position.y, pose.position.z)
    pose_rot = (pose.orientation.x, pose.orientation.y, pose.orientation.z, pose.orientation.w)

    # pose_mat = fromTranslationRotation(pose_trans, pos_rot)

    marker_name = "/marker_%d" % marker_id

    # print("Looking for marker %s" % marker_name)

    #Get the offset of the marker from the origin
    try:
```

```
    (marker_trans, marker_rot) = self.listener.lookupTransform('world',
        marker_name, rospy.Time(0))
  except (LookupException, ConnectivityException,
      ExtrapolationException):
    print("Unable to find marker transform")
    return

  try:
    (base_trans, base_rot) = self.listener.lookupTransform('/ardrone/
        ardrone_base_link', '/ardrone/ardrone_base_bottomcam', rospy.
        Time(0))
  except (LookupException, ConnectivityException,
      ExtrapolationException):
    print("Unable to find ardrone transform")
    return

  #Everything is in m not mm
  pose_mat = numpy.matrix(self.transformer.fromTranslationRotation(
      pose_trans, pose_rot))
  pose_mat_inv = pose_mat.getI()
  marker_mat = numpy.matrix(self.transformer.fromTranslationRotation(
      marker_trans, marker_rot))
  base_mat = numpy.matrix(self.transformer.fromTranslationRotation(
      base_trans, base_rot))
  base_mat_inv = base_mat.getI()
  # print marker_mat

  origin = numpy.matrix([[0], [0], [0], [1]])

  #Not quite the right transformation
  global_mat = marker_mat*pose_mat_inv*base_mat_inv

  global_trans = translation_from_matrix(global_mat)
```

```
        global_rot = rotation_from_matrix(global_mat)

        print global_rot

        # print pose_mat_inv*base_mat_inv*marker_mat*origin
        # print pose_trans
        # print marker_trans
        # print base_trans
        # marker_mat = fromTranslationRotation(marker_trans, marker_rot)

        #What is the correct order for the transformation matrices?
        #TURN INTO ARDRONE_BASE_LINK
        # estimate = pose_inv_mat*marker_mat

        #Take inverse of estimate, and apply it to origin to get in global space

        #Upper left 3x3 matrix should be the rotation. First column is the normalized vector of heading (use atan2)

        #Should I create new particles or just strongly resample the old ones? Maybe a mixture?
        #INSERT ADJUSTMENT HERE


        # self.fp.write("%d,%f,%f,%f,%f,%f,%f,%f," % (rospy.Time(0), self.step, marker_id, estimate[0], estimate[1], estimate[2]))
        # self.fp.write("%f,%f,%f,%f,%f,%f,%f," % (pose_trans[0], pose_trans[1], pose_trans[2], pose_rot[0], pose_rot[1], pose_rot[2], pose_rot[3]))
        # self.fp.write("%f,%f,%f,%f,%f,%f,%f\n" % (marker_trans[0], marker_trans[1], marker_trans[2], marker_rot[0], marker_rot[1], marker_rot[2], marker_rot[3]))
```

```python
def update_marker(self):

  self.est_pose.x = self.est.x
  self.est_pose.y = self.est.y
  self.est_pose.z = self.est.z

  self.est_pub.publish(self.est_pose)


#Calculate the weight for particles
def weight_particles(self, delta_t, x_est, y_est, z_est, theta_est):
  self.weight_dict = dict()
  weight_sum = 0
  for particle in self.particle_list:
    #THIS WEIGHTING IS JUST A PLACEHOLDER
    #REPLACE WITH SENSOR MODEL DATA

    #Calculate distances
    lat_dist = ((x_est - particle.x)**2 + (y_est - particle.y)**2)**.5
    vert_dist = abs(z_est - particle.z)
    theta_dist = abs(theta_est - particle.theta)

    lat_weight = self.normpdf(lat_dist, 0, self.vis_noise) #
        Potentially do the z distance separately?
    vert_weight = self.normpdf(vert_dist, 0, self.ultra_noise)
    theta_weight = self.normpdf(theta_dist, 0, self.mag_noise)

    weight = lat_weight + vert_weight + theta_weight

    self.weight_dict[particle] = weight
    weight_sum += weight
```

```python

    #Normalize the weights
    for particle, weight in self.weight_dict.iteritems():
      if (weight_sum != 0):
        weight = weight/weight_sum

  #http://stackoverflow.com/questions/12412895/calculate-probability-in-normal-distribution-given-mean-std-in-python
  def normpdf(self, x, mean, sd):
      var = float(sd)**2
      denom = (2*pi*var)**.5
      num = exp(-(float(x)-float(mean))**2/(2*var))
      return num/denom

  # http://www51.honeywell.com/aero/common/documents/myaerospacecatalog-documents/Defense_Brochures-documents/Magnetic__Literature_Application_notes-documents/AN203_Compass_Heading_Using_Magnetometers.pdf
  def get_heading(self, magX, magY, magZ):

    heading = (atan2(magX, magY)/pi)*180
    return self.clamp_angle(heading - self.start_mag_heading)



  def clamp_angle(self, angle):
    if (angle > 180):
      return angle - 360
    elif (angle < -180):
      return angle + 360
    else:
      return angle

```

```
  #Return an estimate of the pose
  #For now just use linear combination. Should we cluster instead?
  def estimate(self):
    self.est = particle(self)
    for part, weight in self.weight_dict.iteritems():
      self.est.x += part.x*weight
      self.est.y += part.y*weight
      self.est.z += part.z*weight
      self.est.theta += part.z*weight


  def print_particles(self):
    for particle in self.particle_list:
      print particle.to_string()


class particle:
  def __init__(self, particlefilter, x=0, y=0, z=0, theta=0):
    self.x = x #Global
    self.y = y
    self.z = z
    self.x_vel = 0; #Local
    self.y_vel = 0;
    self.z_vel = 0;
    self.theta = theta
    self.parent = particlefilter

  #Update the values of the particle based
  def propogate(self, delta_t, x_acc, y_acc, z_acc, rotX, rotY,
      theta_delta, noise):
    if (self.parent.step%100 == 0):
```

```
      self.parent.fp_part.write("%d, %f, %f, %f, %f, %f, %f, %f, %f\n" % (self.parent.step, delta_t, self.x, self.y, self.z, self.x_vel, self.y_vel, self.z_vel, self.theta))

    x_acc_noise = x_acc
    y_acc_noise = y_acc
    z_acc_noise = z_acc
    theta_delta_noise = theta_delta
    rotX_noise = rotX
    rotY_noise = rotY

    if (noise):
      x_acc_noise = random.normalvariate(x_acc, self.parent.linear_noise)
      y_acc_noise = random.normalvariate(y_acc, self.parent.linear_noise)
      z_acc_noise = random.normalvariate(z_acc, self.parent.linear_noise)
      rotX_noise = random.normalvariate(theta_delta, self.parent.angular_noise)
      rotY_noise = random.normalvariate(theta_delta, self.parent.angular_noise)
      theta_delta_noise = random.normalvariate(theta_delta, self.parent.angular_noise)

    self.theta = self.parent.clamp_angle(self.theta + theta_delta_noise)

    acc_m = numpy.matrix([[x_acc_noise], [y_acc_noise], [z_acc_noise], [1]])

    acc_global_m = rotate(acc_m, rotX_noise, rotY_noise, self.theta)

    x_acc_global = acc_global_m.item(0)
```

```
    y_acc_global = acc_global_m.item(1)
    z_acc_global = acc_global_m.item(2) - 0.942871 #From sensor data

    self.x_vel = x_acc_global*delta_t + self.x_vel
    self.y_vel = y_acc_global*delta_t + self.y_vel
    self.z_vel = z_acc_global*delta_t + self.z_vel


    x_delta = self.x_vel*delta_t
    y_delta = self.y_vel*delta_t
    z_delta = self.z_vel*delta_t

    self.x += x_delta
    self.y += y_delta
    self.z += z_delta

  def to_string(self):
    return "(%.2f, %.2f, %.2f, %.4f)" % (self.x, self.y, self.z, self.
        theta)


def rotate(m, rotX, rotY, rotZ):
  #RIGHT HAND VS LEFT HAND?
  rotX_m = numpy.matrix([[ 1, 0, 0, 0],
              [ 0, cos(radians(rotX)), -sin(radians(rotX)), 0],
              [ 0, sin(radians(rotX)), cos(radians(rotX)), 0],
              [ 0, 0, 0, 1]])
  rotY_m = numpy.matrix([[cos(radians(rotY)), 0, -sin(radians(rotY)),
      0],
              [ 0, 1, 0, 0],
              [ sin(radians(rotY)), 0, cos(radians(rotY)), 0],
              [ 0, 0, 0, 1]])
```

```
    rotZ_m = numpy.matrix([[cos(radians(rotZ)), -sin(radians(rotZ)), 0,
        0],
                    [sin(radians(rotZ)),cos(radians(rotZ)),0,0],
                    [0,0,1,0],
                    [0,0,0,1]])
    return rotX_m*rotY_m*rotZ_m*m


def main():
    pf = particlefilter(num_particles=10)

    print "-----------INIT-----------"

    pf.print_particles()
    # propogate(self, delta_t, x_acc, y_acc, z_acc, rotX, rotY,
        theta_delta, noise)
    pf.propogate(.1, 10, 10, 0, 0, 0, 32, True)

    print "-----------PROP-----------"

    pf.print_particles()
    pf.correct(.1, 20, 20, 0, 24, 53, 10)

    print "-----------CORRECT-----------"

    pf.print_particles()

if __name__ == "__main__":
    main()
```

Bibliography

[1]

[2]

[3]

[4] Agisoft photoscan.

[5] Bundler: Structure from motion (sfm) for unordered image collections.

[6] Pierre-Jean Bristeau, Franois Callou, David Vissire, and Nicolas Petit. The navigation and control technology inside the ar.drone micro uav, 2011.

[7] Nick Dijkshoorn. Simultaneous localization and mapping with the ar.drone, 2012.

[8] J. Engel, J. Sturm, and D. Cremers. Accurate figure flying with a quadrocopter using onboard visual and inertial sensing. *IMU*, 320:240.

[9] J. Engel, J. Sturm, and D. Cremers. Camera-based navigation of a low-cost quadrocopter. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 2815 –2821, oct. 2012.

[10] Dieter Fox, Sebastian Thrun, Wolfram Burgard, and Frank Dellaert. Particle filters for mobile robot localization, 2001.

[11] A. Irschara, V. Kaufmann, M. Klopschitz, H. Bischof, and F. Leberl. Towards fully automatic photogrammetric reconstruction using digital images taken from uavs. In *Proceedings of the ISPRS TC VII Symposium100 Years ISPRS*, 2010.

[12] K.Y.K. Leung, C.M. Clark, and J.P. Huissoon. Localization in urban environments by matching ground level video images with an aerial image. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 551 –556, may 2008.

[13] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 131–144, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[14] Joao Pedro Baptista Mendes. Assisted teleoperation of quadcopters using obstacle avoidance. 2012.

[15] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3d model acquisition, 2002.

[16] R. Steffen and W. Förstner. On visual real time mapping for unmanned aerial vehicles. In *21st Congress of the International Society for Photogrammetry and Remote Sensing (ISPRS)*, pages 57–62, 2008.

[17] Teddy Yap, Mingyang Li, Anastasios I. Mourikis, and Christian R. Shelton. A particle filter for monocular vision-aided odometry.