

# Experimental Analyses of Heuristics for Horsefly-type Problems

Gaurish Telang

# Contents

	Page
<i>I Overview</i>	<b>4</b>
<b>1 Descriptions of Problems</b>	<b>5</b>
<b>2 Installation and Use</b>	<b>7</b>
<i>II Programs</i>	<b>9</b>
<b>3 Overview of the Code Base</b>	<b>10</b>
3.1 Source Tree . . . . .	10
3.2 The Main Files . . . . .	11
3.3 Support Files . . . . .	12
<b>4 Some (Boring) Utility Functions</b>	<b>14</b>
4.1 Graphical Utilities . . . . .	14
4.2 Algorithmic Utilities . . . . .	17
<b>5 Classic Horsefly</b>	<b>21</b>
5.1 Module Overview . . . . .	21
5.2 Module Details . . . . .	21
5.3 Local Data Structures . . . . .	26
5.4 Algorithm : Dumb Brute force . . . . .	28
5.5 Algorithm : Greedy—Nearest Neighbor . . . . .	28
5.6 Algorithm : Greedy—Incremental Insertion . . . . .	37
5.7 Insertion Policies . . . . .	46
5.8 Lower Bound: The $\varphi$ -Prim-MST . . . . .	50
5.9 Algorithm : Doubling the $\varphi$ -MST . . . . .	53
5.10 Algorithm : Bottom-Up Split . . . . .	53
5.11 Algorithm : Local Search—Swap . . . . .	53
5.12 Algorithm : K2 Means . . . . .	54
5.13 Algorithm : TSP ordering . . . . .	59
5.14 Local Utility Functions . . . . .	60
5.15 Plotting Routines . . . . .	62
5.16 Animation routines . . . . .	65
5.17 Chapter Index of Fragments . . . . .	69
5.18 Chapter Index of Identifiers . . . . .	70
<b>6 Reverse Horsefly</b>	<b>72</b>
<b>7 One Horse, Multiple Flies</b>	<b>73</b>

---

Appendices	74
A Index of Files	75
B Man-page for <code>main.py</code>	76

# Part I

## Overview

# Chapter 1

## Descriptions of Problems

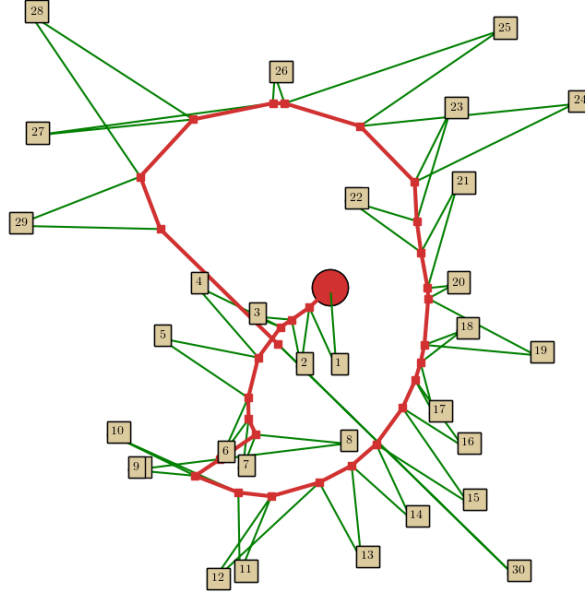


Figure 1.1: An Example of a classic Horsefly tour with  $\varphi = 5$ . The red dot indicates the initial position of the horse and fly, given as part of the input. The ordering of sites shown has been computed with a greedy algorithm which will be described later

The Horsefly problem is a generalization of the well-known Euclidean Traveling Salesman Problem. In the most basic version of the Horsefly problem (which we call “**Classic Horsefly**”), we are given a set of sites, the initial position of a truck(horse) with a drone(fly) mounted on top, and the speed of the drone-speed  $\varphi$ .<sup>1 2</sup>

The goal is to compute a tour for both the truck and the drone to deliver package to sites as quickly as possible. For delivery, a drone must pick up a package from the truck, fly to the site and come back to the truck to pick up the next package for delivery to another site.<sup>3</sup> Both the truck and drone must coordinate their motions to minimize the time it takes for all the sites to get their packages. Figure 1.1 gives an example of such a tour computed using a greedy heuristic for  $\varphi = 5$ .

This suite of programs implement several experimental heuristics, to solve the above NP-hard problem and some of its variations approximately. In this short chapter, we give a description of the problem variations that we will be tackling. Each of the problems, has a corresponding chapter in Part 2, where these heuristics are described and implemented. We also give comparative analyses of their experimental performance on various problem instances.

**Classic Horsefly** This problem has already described in the introduction.

<sup>1</sup>The speed of the truck is always assumed to be 1 in any of the problem variations we will be considering in this report.

<sup>2</sup> $\varphi$  is also called the “speed ratio”.

<sup>3</sup>The drone is assumed to be able to carry at most one package at a time

**Segment Horsefly** In this variation, the path of the truck is restricted to that of a segment, which we can consider without loss of generality to be  $[0, 1]$ . All sites, without loss of generality lie in the upper-half plane  $\mathbb{R}_+^2$ .

**Fixed Route Horsefly** This is the obvious generalization of Segment Horsefly, where the path which the truck is restricted to travel is a piece-wise linear polygonal path.<sup>4</sup> Both the initial position of the truck and the drone are given. The sites to be serviced are allowed to lie anywhere in  $\mathbb{R}^2$ . Two further variations are possible in this setting, one in which the truck is allowed reversals and the other in which it is not.

**One Horse, Two Flies** The truck is now equipped with two drones. Otherwise the setting, is exactly the same as in classic horsefly. Each drone can carry only one package at a time. The drones must fly back and forth between the truck and the sites to deliver the packages. We allow the possibility that both the drones can land at the same time and place on the truck to pick up their next package.<sup>5</sup>

**Reverse Horsefly** In this model, each site (not the truck!) is equipped with a drone, which fly *towards* the truck to pick up their packages. We need to coordinate the motion of the truck and drone so that the time it takes for the last drone to pick up its package (the “makespan”) is minimized.

**Bounded Distance Horsefly** In most real-world scenarios, the drone will not be able to (or allowed to) go more than a certain distance  $R$  from the truck. Thus with the same settings as the classic horsefly, but with the added constraint of the drone and the truck never being more than a distance  $R$  from the truck, how would one compute the truck and drone paths to minimize the makespan of the deliveries?

**Watchman Horsefly** In place of the TSP, we generalize the Watchman route problem here.<sup>6</sup> We are given as input a simple polygon and the initial position of a truck and a drone. The drone has a camera mounted on top which is assumed to have  $360^\circ$  vision. Both the truck and drone can move, but the drone can move at most euclidean distance<sup>7</sup>  $R$  from the truck.

We want every point in the polygon to be seen by the drone at least once. The goal is to minimize the time it takes for the drone to be able to see every point in the simple polygon. In other words, we want to minimize the time it takes for the drone (moving in coordination with the truck) to patrol the entire polygon.

---

<sup>4</sup>More generally, the truck will be restricted to travelling on a road network, which would typically be modelled as a graph embedded in the plane.

<sup>5</sup>In reality, one of the drones will have to wait for a small amount of time while the other is retrieving its package. In a more realising model, we would need to take into account this “waiting time” too.

<sup>6</sup>although abstractly, the Watchman route problem can be viewed as a kind of TSP

<sup>7</sup>The version where instead geodesic distance is considered is also interesting

# Chapter 2

## Installation and Use

To run these programs you will need to install Docker, an open-source containerization program that is easily installable on Windows 10<sup>1</sup>, MacOS, and almost any GNU/Linux distribution. For a quick introduction to containerization, watch the first two minutes of [https://youtu.be/\\_dfL0zuIg2o](https://youtu.be/_dfL0zuIg2o)

The nice thing about Docker is that it makes it easy to run softwares on different OS'es portably and neatly side-steps the dependency hell problem ([https://en.wikipedia.org/wiki/Dependency\\_hell](https://en.wikipedia.org/wiki/Dependency_hell).) The headache of installing different library dependencies correctly on different machines running different OS'es, is replaced **only** by learning how to install Docker and to set up an X-windows connection between the host OS and an instantiated container running GNU/Linux.

A. [ *Get Docker* ] For installation instructions watch

**GNU/Linux** <https://youtu.be/KCckWweNSrM>

**Windows** <https://youtu.be/ym1Wt1MqURY>

To test your installation, run the hello-world container. Note that you might need administrator privileges to run docker. On Windows, you can open the Powershell as an administrator. On GNU/Linux you should use sudo

B. [ *Download customized Ubuntu image* ] `docker pull gtelang/ubuntu_customized`<sup>2</sup>

C. [ *Clone repository* ] `git clone gtelang/horseflies_literate.git`

D. [ *Mount and Launch* ]

**If you are running GNU/Linux** • Open up your favorite terminal emulator, such as xterm, rxvt or konsole

- Copy to clipboard the output of `xauth list`
- `cd horseflies_literate`
- `docker run -it --name horsefly_container --net=host \`  
    `-e DISPLAY -v /tmp/.X11-unix \`  
    `-v `pwd`: /horseflies_mnt gtelang/ubuntu_customized`
- `cd horseflies_mnt`
- `xauth add <paste-from-clipboard>`

The purpose of using “xauth” and “-e DISPLAY -v /tmp/.X11-unix” is to establish an X-windows connection between your operating system and the Ubuntu container that allows you to run GUI apps e.g. the FireFox web-browser.<sup>3</sup>

**If you are running Windows** • Follow every instruction in <https://dev.to/darksmile92/run-gui-app-in-linux-Docker-container-on-windows-host-4kde>.<sup>4</sup> Make sure you can run

---

<sup>1</sup>You might need to turn on virtualization explicitly in your BIOS, after installing Docker as I needed to while setting Docker up on Windows. Here is a snapshot of an image when turning on Intel's virtualization technology through the BIOS: [https://images.techhive.com/images/article/2015/09/virtualbox\\_vt-x\\_amd-v\\_error04\\_phoenix-100612961-large.idge.jpg](https://images.techhive.com/images/article/2015/09/virtualbox_vt-x_amd-v_error04_phoenix-100612961-large.idge.jpg)

<sup>2</sup>The customized Ubuntu image is approximately 7 GB which contains all the libraries (e.g. CGAL, VTK, numpy, and matplotlib) that I typically use to run my research codes portably. On my home internet connection downloading this Ubuntu-image typically takes about 5-10 minutes.

<sup>3</sup>I found the instructions for running GUI apps on containers in <https://www.youtube.com/watch?v=RDg6TRwiPtg>

<sup>4</sup>This step is necessary displaying the Matplotlib canvas as we do in the horseflies project for interactive testing of algorithms.

a gui program like the Firefox web-browser as indicated by the article before going to the next step.

- To mount the horseflies folder, you need to *share* the appropriate drive (e.g. C:\ or D:\) that the horseflies folder is in with Docker. Follow instructions here: <https://rominirani.com/docker-on-windows-mounting-host-directories-d96f3f056a2c> for sharing directories.<sup>5</sup>
- Open up a Windows Powershell (possibly as administrator)
  - `set-variable -name DISPLAY -value <your-ip-address>:0.0`<sup>6</sup>
  - `docker run -ti --rm -e DISPLAY=$DISPLAY -v <location-of-horseflies-folder>:/horseflies_mnt`

**E.** [ *Run experiments* ] If you want to run all the experiments as described in the paper again to reproduce the reported results on your machine, then run<sup>7</sup>,

```
python main.py --run-all-experiments.
```

If you want to run a specific experiment, then run

```
python main.py --run-experiment <experiment-name>.
```

See Index for a list of all the experiments.

**F.** [ *Test algorithms interactively* ] If you want to test the algorithms in interactive mode (where you get to select the problem-type, mouse-in the sites on a canvas, set the initial position of the truck and drone and set  $\varphi$ ), run `python main.py --<problem-name>`. The list of problems are the same as that given in the previous chapter. The problem name consists of all lower-case letters with spaces replaced by hyphens.

Thus for instance “Watchman Horsefly” becomes `watchman-horsefly` and “One Horse Two Flies” becomes `one-horse-two-flies`.

To interactively experiment with different algorithms for, say, the Watchman Horsefly problem , type at the terminal `python main.py --watchman-horsefly`

If you want to delete the Ubuntu image and any associated containers run the command<sup>8</sup>

```
docker rm -f horsefly_container; docker rmi -f ubuntu_customized
```

That’s it! Happy horseflying!

<sup>5</sup>you might need administrator privileges to perform this step, as pointed out by the article.

<sup>6</sup>You can find your ip-address by the output of the `ipconfig` command in the Powershell

<sup>7</sup>Allowing, of course, for differences between your machine’s CPU and mine when it comes to reporting absolute running time

<sup>8</sup>the ubuntu image is 7GB afterall!



# Part II

## Programs

# Chapter 3

## Overview of the Code Base

All of the code has been written in Python 2.7 and tested using the standard CPython implementation of the language. In some cases, calls will be made to external C++ libraries (mostly CGAL and VTK) using SWIG (<http://www.swig.org/>) for speeding up a slow routine or to use a function that is not available in any existing Python package.

## Source Tree

```
..
|-- src
|   |-- expts
|   |-- lib
|   |   |-- problem_classic_horsefly.py
|   |   |-- utils_algo.py
|   |   `-- utils_graphics.py
|   |-- tests
|   `-- Makefile
|-- tex
|   |-- directory-tree.tex
|   |-- horseflies.pdf
|   |-- horseflies.tdo
|   |-- horseflies.tex
|   `-- standard_settings.tex
|-- webs
|   |-- problem-classic-horsefly
|   |   |-- algo-bottom-up-split.web
|   |   |-- algo-doubling-phi-mst.web
|   |   |-- algo-dumb.web
|   |   |-- algo-greedy-incremental-insertion.web
|   |   |-- algo-greedy-nn.web
|   |   |-- algo-k2-means.web
|   |   |-- algo-local-search-swap.web
|   |   |-- algo-tsp-ordering.web
|   |   |-- lower-bound-phi-mst.web
|   |   `-- problem-classic-horsefly.web
|   |-- problem-fixed-route-horsefly
|   |   `-- problem-fixed-route-horsefly.web
|   |-- problem-one-horse-multiple-flies
|   |   `-- problem-one-horse-multiple-flies.web
|   |-- problem-reverse-horsefly
|   |   `-- problem-reverse-horsefly.web
|   |-- problem-segment-horsefly
|   |   `-- problem-segment-horsefly.web
|   |-- problem-watchman-horsefly
|   |   `-- problem-watchman-horsefly.web
|   `-- '
```

```
| |-- descriptions-of-problems.web
| |-- horseflies.web
| |-- installation-and-use.web
| |-- overview-of-code-base.web
| |-- utility-functions.web
|-- main.py
|-- todo
`-- weave-tangle.sh
```

12 directories, 33 files

There are three principal directories

**webs/** This contains the source code for the entire project written in the nuweb format along with documents (mostly images) needed during the compilation of the  $\text{\LaTeX}$  files which will be extracted from the `.web` files.

**src/** This contains the source code for the entire project “tangled” (i.e. extracted) from the `.web` files.

**tex/** This contains the monolithic `horseflies.tex` extracted from the `.web` files and a bunch of other supporting  $\text{\LaTeX}$  files. It also contains the final compiled `horseflies.pdf` (the current document) which contains the documentation of the project, interwoven with code-chunks and cross-references between them along with the experimental results.

The files in `src` and `tex` should not be touched. Any editing required should be done directly to the `.web` files which should then be weaved and tangled using `weave-tangle.sh`.

# The Main Files

## 3.2.1

- A. [ `main.py` ] The file `main.py` in the top-level folder is the *entry-point* for running code. Its only job is to parse the command-line arguments and pass relevant information to the handler functions for each problem and experiment.

- B. [ *Algorithmic Code* ] All such files are in the directory `src/lib/`. Each of the files with prefix “`problem_*`” contain implementations of algorithms for one specific problem. For instance `problem_watchman_horsefly.py` contains algorithms for approximately solving the Watchman Horsefly problem.

Since Horsefly-type problems are typically NP-hard, an important factor in the subsequent experimental analysis will require, comparing an algorithm’s output against good lower bounds. Each such file, will also have routines for efficiently computing or approximating various lower-bounds for the corresponding problem’s *OPT*.

- C. [ *Experiments* ] All such files are in the directory `src/expt/`. Each of the files with prefix “`expt_*`” contain code for testing hypotheses regarding a problem, generating counter-examples or comparing the experimental performance of the algorithm implementations for each of the problems. Thus `expt_watchman_horsefly.py` contains code for performing experiments related to the Watchman Horsefly problem.

If you need to edit the source-code for algorithms or experiment you should do so to the `.web` files in the `web` directory. Every problem has a dedicated *folder* containing source-code for algorithms and experiments pertaining to that problem. Every algorithm and experiment has a dedicated `.web` file in these problem directories. Such files are all “tied” together using the file with prefix `problem-<problem-name>` in that same directory (i.e. the file acts as a kind of handler for each problem, that includes the algorithms and experiment web files with the `@i` macro.)

Add an item containing the interface files. Do this for the Haskell files that you will ultimately add in later.

### 3.2.2 Let's define the main.py file now.

Each problem or experiment has a handler routine that effectively acts as a kind of “main” function for that module that does house-keeping duties by parsing the command-line arguments passed by main, setting up the canvas by calling the appropriate graphics routines and calling the algorithms on the input specified through the canvas.

"../main.py" 12a≡

```

< Turn off Matplotlibs irritating DEBUG messages 12b >
< Import problem module files 12c >

if __name__=="__main__":
    # Select algorithm or experiment
    if (len(sys.argv)==1):
        print "Specify the problem or experiment you want to run"
        sys.exit()

    elif sys.argv[1] == "--problem-classic-horsefly":
        chf.run_handler()

    elif sys.argv[1] == "--problem-one-horse-multiple-flies":
        ohmf.run_handler()

    else:
        print "Option not recognized"
        sys.exit()
◇

```

**3.2.3** On my customized Ubuntu container, Matplotlib produces tons of DEBUG log messages because it recently switched to the logging library for...well...logging. The lines in this chunk were suggested by the link <http://matplotlib.1069221.n5.nabble.com/How-to-turn-off-matplotlib-DEBUG-msgs-td48822.html> for quietening down Matplotlib.

```

< Turn off Matplotlibs irritating DEBUG messages 12b > ≡
import logging
mpl_logger = logging.getLogger('matplotlib')
mpl_logger.setLevel(logging.WARNING)
◇

```

Fragment referenced in 12a.

< Import problem module files 12c > ≡

```

import sys
sys.path.append('src/lib')
import problem_classic_horsefly as chf
import problem_one_horse_multiple_flies as ohmf
◇

```

Fragment referenced in 12a.

## Support Files

- A. [ *Utility Files* ] All such utility files are in the directory `src/lib/`. These files contain common utility functions for manipulating data-structures, plotting and graphics routines common to all

---

horsefly-type problems. All such files have the prefix `utils_*`. These Python files are generated from the single `.web` file `utils.web` in the `web` subdirectory.

- B.** [ *Tests* ] All such files are in the directory `src/test/`. To automate testing of code during implementations, tests for various routines across the entire code-base have been written in files with prefix `test_*`.

Every problem, utility, and experimental files in `src/lib` and `src/expts` has a corresponding test-file in this folder.

# Chapter 4

## Some (Boring) Utility Functions

We will be needing some utility functions, for drawing and manipulating data-structures which will be implemented in files separate from `problem_classic_horsefly.py`. All such files will be prefixed with the work `utils_`. Many of the important common utility functions are defined here; others will be defined on the fly throughout the rest of the report. This chapter just collects the most important of the functions for the sake of clarity of exposition in the later chapters.

## Graphical Utilities

Here we will develop routines to interactively insert points onto a Matplotlib canvas and clear the canvas. Almost all variants of the horsefly problem will involve mousing in sites and the initial position of the horse and fly. These points will typically be represented by small circular patches. The type of the point will be indicated by its color and size e.g. initial position of truck and drone will typically be represented by a large red dot while and the sites by smaller blue dots.

Matplotlib has extensive support for inserting such circular patches onto its canvas with mouse-clicks. Each such graphical canvas corresponds (roughly) to Matplotlib figure object instance. Each figure consists of several Axes objects which contains most of the figure elements i.e. the Axes objects correspond to the “drawing area” of the canvas.

**4.1.1** First we set up the axes limits, dimensions and other configuration quantities which will correspond to the “without loss of generality” assumptions made in the statements of the horsefly problems. We also need to set up the axes limits, dimensions, and other fluff. The following fragment defines a function which “normalizes” a drawing area by setting up the x and y limits and making the aspect ratio of the axes object the same i.e. 1.0. Since Matplotlib is principally a plotting software, this is not the default behavior, since scales on the x and y axes are adjusted according to the data to be plotted.

```
"../src/lib/utils_graphics.py" 14≡
```

```
from matplotlib import rc
from colorama import Fore
from colorama import Style
from scipy.optimize import minimize
from sklearn.cluster import KMeans
import argparse
import itertools
import math
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pprint as pp
import randomcolor
import sys
import time

xlim, ylim = [0,1], [0,1]
```

```
def applyAxCorrection(ax):
    ax.set_xlim([xlim[0], xlim[1]])
    ax.set_ylim([ylim[0], ylim[1]])
    ax.set_aspect(1.0)
    ◇
```

File defined by 14, 15abc, 17a.

**4.1.2** Next, given an axes object (i.e. a drawing area on a figure object) we need a function to delete and remove all the graphical objects drawn on it.

"../src/lib/utils\_graphics.py" 15a≡

```
def clearPatches(ax):
    # Get indices cooresponding to the polygon patches
    for index , patch in zip(range(len(ax.patches)), ax.patches):
        if isinstance(patch, mpl.patches.Polygon) == True:
            patch.remove()
    ax.lines[:] = []
    applyAxCorrection(ax)
    ◇
```

File defined by 14, 15abc, 17a.

**4.1.3** Now remove the patches which were rendered for each cluster Unfortunately, this step has to be done manually, the canvas patch of a cluster and the corresponding object in memory are not reactively connected. I presume, this behaviour can be achieved by sub-classing.

"../src/lib/utils\_graphics.py" 15b≡

```
def clearAxPolygonPatches(ax):

    # Get indices cooresponding to the polygon patches
    for index , patch in zip(range(len(ax.patches)), ax.patches):
        if isinstance(patch, mpl.patches.Polygon) == True:
            patch.remove()
    ax.lines[:] = []
    applyAxCorrection(ax)
    ◇
```

File defined by 14, 15abc, 17a.

**4.1.4** Now for one of the most important routines for drawing on the canvas! To insert the sites, we double-click the left mouse button and to insert the initial position of the horse and fly we double-click the right mouse-button.

The following chunk defines a function that creates a closure for a mouseclick even on the matplotlib canvas.

Note that the left mouse-button corresponds to button 1 and right mouse button to button 3 in the code-fragment below.

"../src/lib/utils\_graphics.py" 15c≡

```
def wrapperEnterRunPoints(fig, ax, run):
    def _enterPoints(event):
        if event.name == 'button_press_event' and \
            (event.button == 1 or event.button == 3) and \
```

Remove the previous red patches, which contain the old position of the horse and fly. Doing this is slightly painful, hence keeping it for later.

```

event.dblclick == True and event.xdata != None and event.ydata != None:

    if event.button == 1:
        < Insert blue circle representing a site 16a >

    elif event.button == 3:
        < Insert big red circle representing initial position of horse and fly 16b >

        < Clear polygon patches and set up last minute ax tweaks 16c >

    return _enterPoints

```

File defined by [14](#), [15abc](#), [17a](#).

### 4.1.5

*< Insert blue circle representing a site 16a > ≡*

```

newPoint = (event.xdata, event.ydata)
run.sites.append( newPoint )
patchSize = (xlim[1]-xlim[0])/140.0

ax.add_patch( mpl.patches.Circle( newPoint, radius = patchSize,
                                   facecolor='blue', edgecolor='black' ))
ax.set_title('Points Inserted: ' + str(len(run.sites)), \
             fontdict={'fontsize':40})

```

Fragment referenced in [15c](#).

### 4.1.6

*< Insert big red circle representing initial position of horse and fly 16b > ≡*

```

inithorseposn = (event.xdata, event.ydata)
run.inithorseposn = inithorseposn
patchSize = (xlim[1]-xlim[0])/70.0

ax.add_patch( mpl.patches.Circle( inithorseposn, radius = patchSize,
                                   facecolor= '#D13131', edgecolor='black' ))

```

Fragment referenced in [15c](#).

**4.1.7** It is inefficient to clear the polygon patches *inside* the `enterRunpoints` event loop as done here. However, this has just been done for simplicity: the intended behaviour at any rate, is to clear all the polygon patches from the axes object, once the user starts entering in more points to the cloud for which the clustering was just computed and rendered. The moment the user starts entering new points, the previous polygon patches are garbage collected.

*< Clear polygon patches and set up last minute ax tweaks 16c > ≡*

```

clearAxPolygonPatches(ax)
applyAxCorrection(ax)
fig.canvas.draw()

```

Fragment referenced in [15c](#).

**4.1.8** We also need a function to be able to generate visually distinct colors. The HSV color model is particularly suitable for this. The following function has been adapted from <https://martin.ankerl>.



[com/2009/12/09/how-to-create-random-colors-programmatically/](http://com/2009/12/09/how-to-create-random-colors-programmatically/).

To generate a random color the function uses a random hue but fixed values of the saturation and value (both lying in the interval  $[0, 1)$ ).

```

"../src/lib/utils_graphics.py" 17a≡

def get_random_color(sat=0.7, val=0.7):
    def hsv_to_rgb(h, s, v):
        h_i = int((h*6))
        f = h*6 - h_i
        p = v * (1 - s)
        q = v * (1 - f*s)
        t = v * (1 - (1 - f) * s)

        if h_i==0:
            r, g, b = v, t, p
        elif h_i==1:
            r, g, b = q, v, p
        elif h_i==2:
            r, g, b = p, v, t
        elif h_i==3:
            r, g, b = p, q, v
        elif h_i==4:
            r, g, b = t, p, v
        elif h_i==5:
            r, g, b = v, p, q

        return [int(r*256), int(g*256), int(b*256)]
    return hsv_to_rgb(np.random.rand(), sat, val)
◇

```

File defined by [14](#), [15abc](#), [17a](#).

## Algorithmic Utilities

**4.2.1** Given a list of points  $[p_0, p_1, p_2, \dots, p_{n-1}]$ . the following function returns,  $[p_1 - p_0, p_2 - p_1, \dots, p_{n-1} - p_{n-2}]$  i.e. it converts the list of points into a consecutive list of numpy vectors. Points should be lists or tuples of length 2

```

"../src/lib/utils_algo.py" 17b≡

import numpy as np
import random
from colorama import Fore
from colorama import Style

def vector_chain_from_point_list(pts):
    vec_chain = []
    for pair in zip(pts, pts[1:]):
        tail= np.array (pair[0])
        head= np.array (pair[1])
        vec_chain.append(head-tail)

    return vec_chain
◇

```

File defined by [17b](#), [18abcd](#), [19abc](#).

**4.2.2** Given a polygonal chain in the form of successive points  $[p_0, p_1, p_2, \dots, p_{n-1}]$ , an important computation is to calculate its length. Points should be lists or tuples of length 2. If no points or just one point is given in the list of points, then 0 is returned.

Typically used for computing the length of the horse's and fly's tours.

"../src/lib/utils\_algo.py" 18a≡

```
def length_polygonal_chain(pts):
    vec_chain = vector_chain_from_point_list(pts)

    acc = 0
    for vec in vec_chain:
        acc = acc + np.linalg.norm(vec)
    return acc
◇
```

File defined by [17b](#), [18abcd](#), [19abc](#).

**4.2.3** The following routine is useful on long lists returned from external solvers. Often point-data is given to and returned from these external routines in flattened form. The following routines are needed to convert such a “flattened” list into a list of points and vice versa.

Convert a vector of even length into a vector of points. i.e.  $[x_0, x_1, x_2, \dots, x_{2n}] \rightarrow [[x_0, x_1], [x_2, x_3], \dots, [x_{2n-1}, x_{2n}]]$

"../src/lib/utils\_algo.py" 18b≡

```
def pointify_vector(x):
    if len(x) % 2 == 0:
        pts = []
        for i in range(len(x))[:2]:
            pts.append([x[i], x[i+1]])
        return pts
    else:
        sys.exit('List of items does not have an even length to be able to be pointified')
◇
```

File defined by [17b](#), [18abcd](#), [19abc](#).

The next chunk performs the opposite process i.e. it flattens the vector e.g.  $[[0, 1], [2, 3], [4, 5]] \rightarrow [0, 1, 2, 3, 4, 5]$

"../src/lib/utils\_algo.py" 18c≡

```
def flatten_list_of_lists(l):
    return [item for sublist in l for item in sublist]
◇
```

File defined by [17b](#), [18abcd](#), [19abc](#).

**4.2.4** Python's default print function prints each list on a single line. For debugging purposes, it helps to print a list with one item per line.

"../src/lib/utils\_algo.py" 18d≡

```
def print_list(xs):
    for x in xs:
        print x
◇
```

File defined by [17b](#), [18abcd](#), [19abc](#).

**4.2.5** This chunk just calculates the list of partial sums e.g.  $[4, 2, 3] \rightarrow [4, 6, 9]$

"../src/lib/utils\_algo.py" 19a≡

```
def partial_sums( xs ):
    psum = 0
    acc = []
    for x in xs:
        psum = psum+x
        acc.append( psum )
    return acc
```

◇

File defined by 17b, 18abcd, 19abc.

**4.2.6** For two given lists of points test if they are equal or not. We do this by checking the  $L^\infty$  norm.

"../src/lib/utils\_algo.py" 19b≡

```
def are_site_orderings_equal(sites1, sites2):

    for (x1,y1), (x2,y2) in zip(sites1, sites2):
        if (x1-x2)**2 + (y1-y2)**2 > 1e-8:
            return False
    return True
```

◇

File defined by 17b, 18abcd, 19abc.

**4.2.7** This function just generates a bunch of non-uniformly distributed random points inside the unit-square. According to this scheme, you will often notice clusters clumped near the border of the unit-square.

"../src/lib/utils\_algo.py" 19c≡

```
def bunch_of_non_uniform_random_points(numpts):
    cluster_size = int(np.sqrt(numpts))
    numcenters = cluster_size

    import scipy
    import random
    centers = scipy.rand(numcenters,2).tolist()

    scale, points = 4.0, []
    for c in centers:
        cx, cy = c[0], c[1]
        < For current center c of this loop, generate cluster_size points uniformly in a square centered at it 20a >

        < Whatever number of points are left to be generated, generate them uniformly inside the unit-square 20b >

    return points
```

◇

File defined by 17b, 18abcd, 19abc.

Defines: cluster\_size 20ab, scale, 20a.

**4.2.8** Note that the smaller square around a center, inside which the points are generated is made to lie in the unit-square. This is reflected in the assignment to `sq_size` below.

*⟨ For current center  $c$  of this loop, generate `cluster_size` points uniformly in a square centered at it 20a ⟩*  $\equiv$

```
sq_size      = min(cx,1-cx,cy, 1-cy)
loc_pts_x    = np.random.uniform(low=cx-sq_size/scale, high=cx+sq_size/scale, size=(cluster_size,))
loc_pts_y    = np.random.uniform(low=cy-sq_size/scale, high=cy+sq_size/scale, size=(cluster_size,))
points.extend(zip(loc_pts_x, loc_pts_y))
◇
```

Fragment referenced in 19c.

Uses: `cluster_size` 19c, `scale`, 19c.

## 4.2.9

*⟨ Whatever number of points are left to be generated, generate them uniformly inside the unit-square 20b ⟩*  $\equiv$

```
num_remaining_pts = numpts - cluster_size * numcenters
remaining_pts = scipy.rand(num_remaining_pts, 2).tolist()
points.extend(remaining_pts)
◇
```

Fragment referenced in 19c.

Uses: `cluster_size` 19c.

# Chapter 5

## Classic Horsefly

### Module Overview

**5.1.1** All algorithms to solve the classic horsefly problems have been implemented in `problem_classic_horsefly.py`. The `run_handler` function acts as a kind of main function for this module. It is called from `main.py` to process the command-line arguments and run the experimental or interactive sections of the code.

```
"../src/lib/problem_classic_horsefly.py" 21a≡  
  
    < Relevant imports for classic horsefly 21b >  
    < Set up logging information relevant to this module 22a >  
    def run_handler():  
        < Define key-press handler 22b >  
        < Set up interactive canvas 25b >  
  
    < Local data-structures for classic horsefly 26a >  
    < Local utility functions for classic horsefly 60a, ... >  
    < Algorithms for classic horsefly 28, ... >  
    < Lower bounds for classic horsefly 51 >  
    < Plotting routines for classic horsefly 62a, ... >  
    < Animation routines for classic horsefly 65 >  
    ◇
```

### Module Details

#### 5.2.1

< Relevant imports for classic horsefly 21b > ≡

```
from colorama import Fore, Style  
from matplotlib import rc  
from scipy.optimize import minimize  
from sklearn.cluster import KMeans  
import argparse  
import inspect  
import itertools  
import logging  
import math  
import matplotlib as mpl  
import matplotlib.pyplot as plt  
# plt.style.use('seaborn-poster')  
import numpy as np  
import os
```

```

import pprint as pp
import randomcolor
import sys
import time
import utils_algo
import utils_graphics
◇

```

Fragment referenced in 21a.

**5.2.2** The logger variable becomes global in scope to this module. This allows me to write customized debug and info functions that let's me format the log messages according to the frame level. I learned this trick from the following Stack Overflow post <https://stackoverflow.com/a/5500099/505306>.

*< Set up logging information relevant to this module 22a > ≡*

```

logger=logging.getLogger(__name__)
logging.basicConfig(level=logging.DEBUG)

def debug(msg):
    frame,filename,line_number,function_name,lines,index=inspect.getouterframes(
        inspect.currentframe())[1]
    line=lines[0]
    indentation_level=line.find(line.lstrip())
    logger.debug('{i} [{m}]'.format(
        i='.'*indentation_level, m=msg))

def info(msg):
    frame,filename,line_number,function_name,lines,index=inspect.getouterframes(
        inspect.currentframe())[1]
    line=lines[0]
    indentation_level=line.find(line.lstrip())
    logger.info('{i} [{m}]'.format(
        i='.'*indentation_level, m=msg))
◇

```

Fragment referenced in 21a.

Uses: logger 39b.

**5.2.3** The key-press handler function detects the keys pressed by the user when the canvas is in active focus. This function allows you to set some of the input parameters like speed ratio  $\varphi$ , or selecting an algorithm interactively at the command-line, generating a bunch of uniform or non-uniformly distributed points on the canvas, or just plain clearing the canvas for inserting a fresh input set of points.

*< Define key-press handler 22b > ≡*

```

# The key-stack argument is mutable! I am using this hack to my advantage.
def wrapperkeyPressHandler(fig,ax, run):
    def _keyPressHandler(event):
        if event.key in ['i', 'I']:
            < Start entering input from the command-line 23 >
        elif event.key in ['n', 'N', 'u', 'U']:
            < Generate a bunch of uniform or non-uniform random points on the canvas 24 >
        elif event.key in ['c', 'C']:
            < Clear canvas and states of all objects 25a >
    return _keyPressHandler
◇

```

Fragment referenced in 21a.

Defines: wrapperkeyPressHandler 25b.

**5.2.4** Before running an algorithm, the user needs to select through a menu displayed at the terminal, which one to run. Each algorithm itself, may be run under different conditions, so depending on the key-pressed(and thus algorithm chosen) further sub-menus will be generated at the command-line.

After running the appropriate algorithm, we render the structure computed to a matplotlib canvas/window along with possibly some meta data about the run at the terminal.

This code-chunk is long, but just has brain-dead code. Nothing really needs to be explained about it any further, nor does it need to be broken down.

*(Start entering input from the command-line 23)  $\equiv$*

```

phi_str = raw_input(Fore.YELLOW + "Enter speed of fly (should be >1): " + Style.RESET_ALL)
phi = float(phi_str)

input_str = raw_input(Fore.YELLOW                                +\
    "Enter algorithm to be used to compute the tour:\n Options are:\n" +\
    " (e)   Exact \n"                                           +\
    " (t)   TSP  \n"                                           +\
    " (tl)  TSP  (using approximate L1 ordering)\n"             +\
    " (k)   k2-center \n"                                       +\
    " (kl)  k2-center (using approximate L1 ordering)\n"       +\
    " (g)   Greedy\n"                                           +\
    " (gl)  Greedy (using approximate L1 ordering)\n"           +\
    " (ginc) Greedy Incremental\n"                               +\
    " (phi-mst) Compute the phi-prim-mst "                      +\
    Style.RESET_ALL)

input_str = input_str.lstrip()

# Incase there are patches present from the previous clustering, just clear them
utils_graphics.clearAxPolygonPatches(ax)

if input_str == 'e':
    horseflytour = \
        run.getTour( algo_dumb,
                     phi )
elif input_str == 'k':
    horseflytour = \
        run.getTour( algo_kmeans,
                     phi,
                     k=2,
                     post_optimizer=algo_exact_given_specific_ordering)

    print " "
    print Fore.GREEN, horseflytour['tour_points'], Style.RESET_ALL
elif input_str == 'kl':
    horseflytour = \
        run.getTour( algo_kmeans,
                     phi,
                     k=2,
                     post_optimizer=algo_approximate_L1_given_specific_ordering)
elif input_str == 't':
    horseflytour = \
        run.getTour( algo_tsp_ordering,
                     phi,
                     post_optimizer=algo_exact_given_specific_ordering)
elif input_str == 'tl':
    horseflytour = \
        run.getTour( algo_tsp_ordering,
                     phi,
                     post_optimizer= algo_approximate_L1_given_specific_ordering)
elif input_str == 'g':
    horseflytour = \
        run.getTour( algo_greedy,
                     phi,
                     post_optimizer= algo_exact_given_specific_ordering)
elif input_str == 'gl':

```

```

horseflytour = \
    run.getTour( algo_greedy,
                phi,
                post_optimizer= algo_approximate_L1_given_specific_ordering)

elif input_str == 'ginc':
    horseflytour = \
        run.getTour( algo_greedy_incremental_insertion,
                    phi, post_optimizer= algo_exact_given_specific_ordering)

elif input_str == 'phi-mst':
    phi_mst = \
        run.computeStructure(compute_phi_prim_mst ,phi)
else:
    print "Unknown option. No horsefly for you! ;-D "
    sys.exit()

#print horseflytour['tour_points']

if input_str not in ['phi-mst']:
    plotTour(ax,horseflytour, run.inithorseposn, phi, input_str)
elif input_str == 'phi-mst':
    draw_phi_mst(ax, phi_mst, run.inithorseposn, phi)

utils_graphics.applyAxCorrection(ax)
fig.canvas.draw()
◇

```

Fragment referenced in [22b](#).

Uses: [algo\\_exact\\_given\\_specific\\_ordering 32a](#), [algo\\_greedy\\_incremental\\_insertion, 39a](#), [computeStructure 26d](#), [draw\\_phi\\_mst 64b](#), [getTour 26c](#), [plotTour 62a](#).

**5.2.5** This chunk generates points uniformly or non-uniformly distributed in the unit square  $[0, 1]^2$  in the Matplotlib canvas. I will document the schemes used for generating the non-uniformly distributed points later. These schemes are important to test the effectiveness of the horsefly algorithms. Uniform point clouds do not highlight the weaknesses of sequencing algorithms as David Johnson implies in his article on how to write experimental algorithm papers when he talks about algorithms for the TSP.

Note that the option keys 'n' or 'N' for entering in non-uniform random-points is just in case the caps-lock key has been pressed on by the user accidentally. Similarly for the 'u' and 'U' keys.

*⟨ Generate a bunch of uniform or non-uniform random points on the canvas 24 ⟩ ≡*

```

numpts = int(raw_input("\n" + Fore.YELLOW + \
                        "How many points should I generate?: " + \
                        Style.RESET_ALL))

run.clearAllStates()
ax.cla()

utils_graphics.applyAxCorrection(ax)
ax.set_xticks([])
ax.set_yticks([])

fig.texts = []

import scipy
if event.key in ['n', 'N']:
    run.sites = utils_algo.bunch_of_non_uniform_random_points(numpts)
else :
    run.sites = scipy.rand(numpts,2).tolist()

patchSize = (utils_graphics.xlim[1]-utils_graphics.xlim[0])/140.0

```



```

for site in run.sites:
    ax.add_patch(mpl.patches.Circle(site, radius = patchSize, \
                                     facecolor='blue',edgecolor='black' ))

ax.set_title('Points : ' + str(len(run.sites)), fontdict={'fontsize':40})
fig.canvas.draw()
◇

```

Fragment referenced in [22b](#).

Uses: `clearAllStates` [26b](#).

**5.2.6** Clearing the canvas and states of all objects is essential when we want to test out the algorithm on a fresh new point-set; the program need not be shut-down and rerun.

*⟨ Clear canvas and states of all objects 25a ⟩ ≡*

```

run.clearAllStates()
ax.cla()

utils_graphics.applyAxCorrection(ax)
ax.set_xticks([])
ax.set_yticks([])

fig.texts = []
fig.canvas.draw()
◇

```

Fragment referenced in [22b](#).

Uses: `clearAllStates` [26b](#).

## 5.2.7

*⟨ Set up interactive canvas 25b ⟩ ≡*

```

fig, ax = plt.subplots()
run = HorseFlyInput()
#print run

ax.set_xlim([utils_graphics.xlim[0], utils_graphics.xlim[1]])
ax.set_ylim([utils_graphics.ylim[0], utils_graphics.ylim[1]])
ax.set_aspect(1.0)
ax.set_xticks([])
ax.set_yticks([])

mouseClick = utils_graphics.wrapperEnterRunPoints (fig,ax, run)
fig.canvas.mpl_connect('button_press_event' , mouseClick )

keyPress    = wrapperkeyPressHandler(fig,ax, run)
fig.canvas.mpl_connect('key_press_event', keyPress  )
plt.show()
◇

```

Fragment referenced in [21a](#).

Uses: `HorseFlyInput` [26a](#), `wrapperkeyPressHandler` [22b](#).

# Local Data Structures

**5.3.1** This class manages the input and the output of the result of calling various horsefly algorithms.

```

< Local data-structures for classic horsefly 26a > ≡
class HorseFlyInput:
    def __init__(self, sites=[], inithorseposn=()):
        self.sites = sites
        self.inithorseposn = inithorseposn

    < Methods for HorseFlyInput 26b, ... >

```

Fragment referenced in [21a](#).

Defines: `HorseFlyInput` [25b](#).

**5.3.2** Set the sites to an empty list and initial horse position to the empty tuple.

```

< Methods for HorseFlyInput 26b > ≡

def clearAllStates (self):
    self.sites = []
    self.inithorseposn = ()


```

Fragment defined by [26bcd](#), [27](#).

Fragment referenced in [26a](#).

Defines: `clearAllStates` [24](#), [25a](#).

**5.3.3** This method sets an algorithm for calculating a horsefly tour. The name of the algorithm is passed as a command-line argument. The list of possible algorithms are typically prefixed with `algo_`.

The output is a dictionary of size 2, containing two lists:

1. Contains the vertices of the polygonal path taken by the horse
2. The list of sites in the order in which they are serviced by the tour, i.e. the order in which the sites are serviced by the fly.

```

< Methods for HorseFlyInput 26c > ≡

def getTour(self, algo, speedratio, k=None, post_optimizer=None):

    if k==None and post_optimizer==None:
        return algo(self.sites, self.inithorseposn, speedratio)
    elif k == None:
        return algo(self.sites, self.inithorseposn, speedratio, post_optimizer=post_optimizer)
    else:
        return algo(self.sites, self.inithorseposn, speedratio, k, post_optimizer=post_optimizer)


```

Fragment defined by [26bcd](#), [27](#).

Fragment referenced in [26a](#).

Defines: `getTour` [23](#).

Uses: `self.inithorseposn`, [47a](#), `self.sites`, [47a](#).

## 5.3.4

```

< Methods for HorseFlyInput 26d > ≡

def computeStructure(self, structure_func, phi):
    print Fore.RED, "Computing the phi-mst", Style.RESET_ALL
    return structure_func(self.sites, self.inithorseposn, phi)


```

Fragment defined by [26bcd](#), [27](#).

Fragment referenced in [26a](#).

Defines: `computeStructure` [23](#).

Uses: `self.inithorseposn`, [47a](#), `self.sites`, [47a](#).

### 5.3.5 This chunk prints a customized representation of the `HorseFlyInput` class

*( Methods for HorseFlyInput 27 )*  $\equiv$

```
def __repr__(self):

    if self.sites != []:
        tmp = ''
        for site in self.sites:
            tmp = tmp + '\n' + str(site)
        sites = "The list of sites to be serviced are " + tmp
    else:
        sites = "The list of sites is empty"

    if self.inithorseposn != ():
        inithorseposn = "\nThe initial position of the horse is " + str(self.inithorseposn)
    else:
        inithorseposn = "\nThe initial position of the horse has not been specified"

    return sites + inithorseposn
◇
```

Fragment defined by [26bcd](#), [27](#).

Fragment referenced in [26a](#).

Now that all the boring boiler-plate and handler codes have been written, its finally time for algorithmic ideas and implementations! Every algorithm is given an algorithmic overview followed by the detailed steps woven together with the source code.

Any local utility functions, needed for algorithmic or graphing purposes are collected at the end of this chapter.

# Algorithm: Dumb Brute force

**5.4.1 Algorithmic Overview** For each of the  $n!$  ordering of sites find the ordering which gives the smallest horsefly tour length. Note that given a particular order of visitation, the optimal tour for the horse can be computed optimally using convex optimization methods or by using the SLSQP solver as I do here.

This method is practical only for a very small number of sites, like say 6 or 7. However, it is useful in generating small counter-examples for various conjectures and as a benchmark for the quality of other algorithms for a small number of sites.

## 5.4.2 Algorithmic Details

*(Algorithms for classic horsefly 28)  $\equiv$*

```
def algo_dumb(sites, horseflyinit, phi):

    tour_length_fn = tour_length(horseflyinit)
    best_tour      = algo_exact_given_specific_ordering(sites, horseflyinit, phi)
    i              = 0

    for sites_perm in list(itertools.permutations(sites)):

        print "Testing a new permutation ", i, " of the sites"; i = i + 1
        tour_for_current_perm = algo_exact_given_specific_ordering (sites_perm, horseflyinit, phi)

        if tour_length_fn(utils_algo.flatten_list_of_lists(tour_for_current_perm ['tour_points']) ) \
            < tour_length_fn(utils_algo.flatten_list_of_lists(          best_tour ['tour_points']) ):

            best_tour = tour_for_current_perm
            print Fore.RED + "Found better tour!" + Style.RESET_ALL

        #print Fore.RED + "\nHorse Waiting times are ", best_tour['horse_waiting_times'] , Style.RESET_ALL
    return best_tour
```

◇

Fragment defined by [28](#), [29](#), [32a](#), [34](#), [39a](#), [54](#), [57](#), [59](#).

Fragment referenced in [21a](#).

Uses: [algo\\_exact\\_given\\_specific\\_ordering 32a](#), [tour\\_length 60a](#).

# Algorithm: Greedy—Nearest Neighbor

**5.5.1 Algorithmic Overview** Before proceeding we give a special case of the classical horseflies problem, which we term “collinear-horsefly”. Here the objective function is again to minimize the tour-length of the drone with the additional restriction that the truck must always be moving in a straight line towards the site on the line-segment joining itself and the site, while the drone is also restricted to travelling along the same line segment.

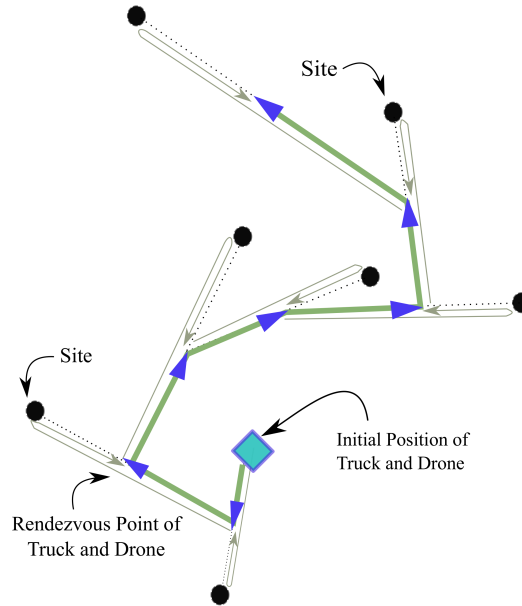


Figure 5.1: The Collinear Horsefly Problem

We can show that an optimal (unrestricted) horsfly solution can be converted to a collinear-horsfly solution at a constant factor increase in the makespan.

## 5.5.2 Algorithmic Details

**5.5.3** This implements the greedy algorithm for the canonical greedy algorithm for collinear horsfly, and then uses the ordering obtained to get the exact tour for that given ordering. Many variations on this are possible. However, this algorithm is simple and may be more amenable to theoretical analysis. We will need an inequality for collapsing chains however.

After extracting the ordering. we use exact/approximate solver for getting a horse-tour that is optimal/approximately optimal for the computed ordering of sites by greedy.

*(Algorithms for classic horsfly 29)  $\equiv$*

```
def algo_greedy(sites, inithorseposn, phi,
                write_algo_states_to_disk_p = True,
                animate_schedule_p = True,
                post_optimizer = None):

    (Set log, algo-state and input-output files config for algo_greedy 30a)
    (Define function next_rendezvous_point_for_horse_and_fly 30b)
    (Define function greedy 31a)

    sites1 = sites[:]
    sites_ordered_by_greedy = greedy(inithorseposn, remaining_sites=sites1)
    answer = post_optimizer(sites_ordered_by_greedy, inithorseposn, phi)

    (Write input and output of algo_greedy to file 31b)
    (Make an animation of the schedule computed by algo_greedy, if animate_schedule_p == True 31c)
    return answer
```

◇

Fragment defined by 28, 29, 32a, 34, 39a, 54, 57, 59.  
 Fragment referenced in 21a.  
 Uses: greedy 31a, write\_algo\_states\_to\_disk\_p 39a.

*⟨ Set log, algo-state and input-output files config for algo\_greedy 30a ⟩ ≡*

```
import sys, logging, datetime, os, errno

algo_name      = 'algo-greedy-nearest-neighbor'
time_stamp     = datetime.datetime.now().strftime('Day-%Y-%m-%d_ClockTime-%H:%M:%S')
dir_name       = algo_name + '---' + time_stamp
log_file_name  = dir_name + '/' + 'run.log'
io_file_name   = 'input_and_output.yml'

# Create directory for writing data-files and logs to for
# current run of this algorithm
try:
    os.makedirs(dir_name)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

logging.basicConfig( filename = log_file_name,
                    level     = logging.DEBUG,
                    format    = '%(asctime)s: %(levelname)s: %(message)s',
                    filemode  = 'w' )
#logger = logging.getLogger()
info("Started running greedy_nearest_neighbor for classic horsefly")

algo_state_counter = 0
◇
```

Fragment referenced in 29.  
 Uses: greedy 31a, logger 39b.

**5.5.4** When there is a single site, the meeting point of horse and fly can be computed exactly (A simple formula is trivial to derive too, which I do so later)/

Here I just use the exact solver for computing the horse tour when the ordering is given for a single site.

*⟨ Define function next\_rendezvous\_point\_for\_horse\_and\_fly 30b ⟩ ≡*

```
def next_rendezvous_point_for_horse_and_fly(horseposn, site):

    horseflytour = algo_exact_given_specific_ordering([site], horseposn, phi)
    return horseflytour['tour_points'][-1]
◇
```

Fragment referenced in 29.  
 Uses: algo\_exact\_given\_specific\_ordering 32a.

**5.5.5** Begin the recursion process where for a given initial position of horse and fly and a given collection of sites you find the nearest neighbor proceed according to segment horsefly formula for just and one site, and for the new position repeat the process for the remaining list of sites. The greedy approach can be extended to by finding the k nearest neighbors, constructing the exact horsefly tour there, at the exit point, you repeat by taking k nearest neighbors and so on.

For reference see this link on how nn queries are performed. <https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.query.html> Warning this is inefficient!!! I am rebuilding the kd-tree at each step. Right now, I am only doing this for convenience.

The next site to get serviced by the drone and horse after they meet-up is the one which is closest to the current position of the horse.

*⟨ Define function greedy 31a ⟩ ≡*

```
def greedy(current_horse_posn, remaining_sites):

    if len(remaining_sites) == 1:
        return remaining_sites
    else:
        from scipy import spatial
        tree = spatial.KDTree(remaining_sites)
        pts = np.array([current_horse_posn])
        query_result = tree.query(pts)
        next_site_idx = query_result[1][0]
        next_site = remaining_sites[next_site_idx]

        next_horse_posn = next_rendezvous_point_for_horse_and_fly(current_horse_posn, next_site)
        remaining_sites.pop(next_site_idx) # the pop method modifies the list in place.

    return [next_site] + greedy(current_horse_posn = next_horse_posn, remaining_sites = remaining_sites)
```

◇

Fragment referenced in 29.

Defines: greedy 29, 30a, 39b.

**5.5.6** The final answer is written to disk in the form of a YAML file. It lists the input sites in the order of visitation computed by the algorithm and gives the tour of the horse. Note that the number of points on the horse's tour is 1 more than the number of given sites.

*⟨ Write input and output of algo\_greedy to file 31b ⟩ ≡*

```
data = {'visited_sites' : answer['site_ordering'] ,
        'horse_tour'    : [inithorseposn] + answer['tour_points'] ,
        'phi'           : phi ,
        'inithorseposn' : inithorseposn}

import yaml
with open(dir_name + '/' + io_file_name, 'w') as outfile:    yaml.dump( data, \
                                outfile, \
                                default_flow_style=False)
debug("Dumped input and output to " + io_file_name)
```

◇

Fragment referenced in 29.

Uses: io\_file\_name, 39b.

## 5.5.7

*⟨ Make an animation of the schedule computed by algo\_greedy, if animate\_schedule\_p == True 31c ⟩ ≡*

```
if animate_schedule_p :
    animateSchedule(dir_name + '/' + io_file_name)
```

◇

Fragment referenced in 29.

**5.5.8** Many of the heuristics, such as the two above that we just implemented, we compute an ordering of sites to visit and then compute the tour-points for the horse. For a given order of visitation calculating the horse-tour can be done by convex optimization. We give one such routine below, that uses the SLSQP non-linear solver from scipy for computing this horse-tour. I will implement the convex optimization routine from John's paper in a later section. Having two such independent routines for doing the same computation can help in benchmarking.

Later, we will also study approximation algorithms for methods to compute horse-tours for a given order of visitation. For these I will need to benchmark the speed of solving SOCP's versus LP's to see what interesting questions can be studied in this regard.

Since the horsely tour lies inside the square, the bounds for each coordinate for the initial guess is between 0 and 1. Many options are possible, Below I try two possibilities

*⟨ Algorithms for classic horsefly 32a ⟩ ≡*

```
def algo_exact_given_specific_ordering (sites, horseflyinit, phi):

    ⟨ Useful functions for algo_exact_given_specific_ordering 32b, ... ⟩

    cons = generate_constraints(horseflyinit, phi, sites)

    # Initial guess for the non-linear solver.
    #x0 = np.empty(2*len(sites)); x0.fill(0.5) # choice of filling vector with 0.5 is arbitrary
    x0 = utils_algo.flatten_list_of_lists(sites) # the initial choice is just the sites

    assert(len(x0) == 2*len(sites))

    x0          = np.array(x0)
    sol         = minimize(tour_length(horseflyinit), x0, method= 'SLSQP', \
                          constraints=cons, options={'maxiter':500})

    tour_points = utils_algo.pointify_vector(sol.x)
    numsites    = len(sites)
    alpha       = horseflyinit[0]
    beta        = horseflyinit[1]
    s           = utils_algo.flatten_list_of_lists(sites)
    horse_waiting_times = np.zeros(numsites)
    ps          = sol.x

    for i in range(numsites):

        if i == 0 :
            horse_time      = np.sqrt((ps[0]-alpha)**2 + (ps[1]-beta)**2)
            fly_time_to_site = 1.0/phi * np.sqrt((s[0]-alpha)**2 + (s[1]-beta)**2 )
            fly_time_from_site = 1.0/phi * np.sqrt((s[0]-ps[1])**2 + (s[1]-ps[1])**2)
        else:
            horse_time      = np.sqrt((ps[2*i]-ps[2*i-2])**2 + (ps[2*i+1]-ps[2*i-1])**2)
            fly_time_to_site = 1.0/phi * np.sqrt((s[2*i]-ps[2*i-2])**2 + (s[2*i+1]-ps[2*i-1])**2 )
            fly_time_from_site = 1.0/phi * np.sqrt((s[2*i]-ps[2*i])**2 + (s[2*i+1]-ps[2*i+1])**2 )

        horse_waiting_times[i] = horse_time - (fly_time_to_site + fly_time_from_site)

    return {'tour_points'          : tour_points,
            'horse_waiting_times'  : horse_waiting_times,
            'site_ordering'        : sites,
            'tour_length_with_waiting_time_included': \
                tour_length_with_waiting_time_included(\
                    tour_points, \
                    horse_waiting_times, \
                    horseflyinit)}
```

◇

Fragment defined by 28, 29, 32a, 34, 39a, 54, 57, 59.

Fragment referenced in 21a.

Defines: algo\_exact\_given\_specific\_ordering 23, 28, 30b.

Uses: generate\_constraints 33, tour\_length 60a, tour\_length\_with\_waiting\_time\_included 60b.

**5.5.9** For the  $i$ th segment of the horsefly tour this function returns a constraint function which models the fact that the time taken by the fly is equal to the time taken by the horse along that particular segment.

*⟨ Useful functions for algo\_exact\_given\_specific\_ordering 32b ⟩ ≡*



```

def ith_leg_constraint(i, horseflyinit, phi, sites):
    if i == 0 :
        def _constraint_function(x):

            #print "Constraint ", i
            start = np.array (horseflyinit)
            site  = np.array (sites[0])
            stop  = np.array ([x[0],x[1]])

            horsetime = np.linalg.norm( stop - start )

            flytime_to_site  = 1/phi * np.linalg.norm( site - start )
            flytime_from_site = 1/phi * np.linalg.norm( stop - site )
            flytime          = flytime_to_site + flytime_from_site
            return horsetime-flytime

        return _constraint_function
    else :

        def _constraint_function(x):

            #print "Constraint ", i
            start = np.array ( [x[2*i-2], x[2*i-1]] )
            site  = np.array ( sites[i])
            stop  = np.array ( [x[2*i] , x[2*i+1]] )

            horsetime = np.linalg.norm( stop - start )

            flytime_to_site  = 1/phi * np.linalg.norm( site - start )
            flytime_from_site = 1/phi * np.linalg.norm( stop - site )
            flytime          = flytime_to_site + flytime_from_site
            return horsetime-flytime

        return _constraint_function

```

◇

Fragment defined by [32b](#), [33](#).

Fragment referenced in [32a](#).

Defines: `ith_leg_constraint` [33](#).

**5.5.10** Given input data, of the problem generate the constraint list for each leg of the tour. The number of legs is equal to the number of sites for the case of single horse, single drone

*< Useful functions for algo\_exact\_given\_specific\_ordering 33 >*  $\equiv$

```

def generate_constraints(horseflyinit, phi, sites):
    cons = []
    for i in range(len(sites)):
        cons.append({'type':'eq', 'fun': ith_leg_constraint(i,horseflyinit,phi,sites)})
    return cons

```

◇

Fragment defined by [32b](#), [33](#).

Fragment referenced in [32a](#).

Defines: `generate_constraints` [32a](#), [57](#).

Uses: `ith_leg_constraint` [32b](#).

**5.5.11** Another useful post-optimizer is one using the  $L1$  metric and linear programming. This solves a Linear program using MOSEK and tries to solve the  $L1$  version of the equations, with some modifications as outlined in the notebook.

The hope is that solving this is more scalable even if approximate than using the SLSQP solver which chokes on  $\geq 70$ -80 sites.

I followed the MOSEK tutorial given here to set up the linear system <https://docs.mosek.com/8.1/pythonapi/tutorial-lo-shared.html>

Note that MOSEK has been optimized to solve large sparse systems of LPs. The LP that I set up here is extremely sparse! And hence a perfect fit for MOSEK.

*( Algorithms for classic horsefly 34 )*  $\equiv$

```
def algo_approximate_L1_given_specific_ordering(sites, horseflyinit, phi):
    import mosek
    numsites = len(sites)

    def p(idx):
        return idx + 0*numsites

    def b(idx):
        return idx + 2*numsites

    def f(idx):
        return idx + 4*numsites

    def h(idx):
        return idx + 6*numsites

    # Define a stream printer to grab output from MOSEK
    def streamprinter(text):
        sys.stdout.write(text)
        sys.stdout.flush()

    numcon = 9 + 13*(numsites-1) # the first site has 9 constraints while the remaining n-1 sites have 13 constraints
    numvar = 8 * numsites # Each 'L1 triangle' has 8 variables associated with it

    alpha = horseflyinit[0]
    beta = horseflyinit[1]

    s = utils_algo.flatten_list_of_lists(sites)

    # Make mosek environment
    with mosek.Env() as env:
        # Create a task object
        with env.Task(0, 0) as task:
            # Attach a log stream printer to the task
            task.set_Stream(mosek.streamtype.log, streamprinter)
            # Append 'numcon' empty constraints.
            # The constraints will initially have no bounds.
            task.appendcons(numcon)
            # Append 'numvar' variables.
            # The variables will initially be fixed at zero (x=0).
            task.appendvars(numvar)

            for idx in range(numvar):
                if (0 <= idx) and (idx < 2*numsites): # free variables (p section of the vector)
                    task.putvarbound(idx, mosek.boundkey.fr, -np.inf, np.inf)

                elif idx == 2*numsites : # b_0 is a known variable
                    val = abs(s[0]-alpha)
                    task.putvarbound(idx, mosek.boundkey.fx, val, val)

                elif idx == 2*numsites +1 : # b_1 is a known variable
```

```

        val = abs(s[1]-beta)
        task.putvarbound(idx, mosek.boundkey.fx, val, val)

    else : # b_2, onwards and the f and h sections of the vector
        task.putvarbound(idx, mosek.boundkey.lo, 0.0, np.inf)

# All the coefficients corresponding to the h's are 1's
# and for the others the coefficients are 0.
for i in range(numvar):
    if i >= 6*numsites: # the h-section
        task.putcj(i,1)
    else: # the p,b,f sections of x
        task.putcj(i,0)

# Constraints for the zeroth triangle corresponding to the zeroth site
row = -1
row += 1; task.putconbound(row, mosek.boundkey.up, -np.inf, alpha ) ; task.putarow(row, [p(0), h(0)], [1.0,
row += 1; task.putconbound(row, mosek.boundkey.lo, alpha , np.inf) ; task.putarow(row, [p(0), h(0)], [1.0,

row += 1; task.putconbound(row, mosek.boundkey.up, -np.inf, beta ) ; task.putarow(row, [p(1), h(1)], [1.0,
row += 1; task.putconbound(row, mosek.boundkey.lo, beta , np.inf) ; task.putarow(row, [p(1), h(1)], [1.0,

row += 1; task.putconbound(row, mosek.boundkey.up, -np.inf, s[0] ) ; task.putarow(row, [p(0), f(0)], [1.0,
row += 1; task.putconbound(row, mosek.boundkey.lo, s[0] , np.inf) ; task.putarow(row, [p(0), f(0)], [1.0,

row += 1; task.putconbound(row, mosek.boundkey.up, -np.inf, s[1] ) ; task.putarow(row, [p(1), f(1)], [1.0,
row += 1; task.putconbound(row, mosek.boundkey.lo, s[1] , np.inf) ; task.putarow(row, [p(1), f(1)], [1.0,

# The most important constraint of all! On the ``L1 triangle''
# time for drone to start from the truck reach site and get back to truck
# = time for truck between the two successive rendezvous points
# The way I have modelled the following constraint it is not exactly
# the same as the previous statement of equality of times of truck
# and drone, but for initial experiments it looks like this gives
# waiting times to be automatically close to 0 (1e-9 close to machine-epsilon)
# Theorem in the making??
row += 1; task.putconbound(row, mosek.boundkey.fx, 0.0 , 0.0 ) ;
task.putarow(row, [b(0), b(1), f(0), f(1), h(0), h(1)], [1.0,1.0,1.0,1.0,-phi, -phi])

# Constraints beginning from the 1st triangle
for i in range(1,numsites):
    row+=1 ; task.putconbound(row, mosek.boundkey.lo, -s[2*i] , np.inf) ; task.putarow(row, [b(2*i), p(
    row+=1 ; task.putconbound(row, mosek.boundkey.lo, s[2*i] , np.inf) ; task.putarow(row, [b(2*i), p(2
    row+=1 ; task.putconbound(row, mosek.boundkey.lo, -s[2*i+1], np.inf) ; task.putarow(row, [b(2*i+1), p(
    row+=1 ; task.putconbound(row, mosek.boundkey.lo, s[2*i+1], np.inf) ; task.putarow(row, [b(2*i+1), p(

    row+=1 ; task.putconbound(row, mosek.boundkey.lo, -s[2*i] , np.inf) ; task.putarow(row, [f(2*i), p(
    row+=1 ; task.putconbound(row, mosek.boundkey.lo, s[2*i] , np.inf) ; task.putarow(row, [f(2*i), p(
    row+=1 ; task.putconbound(row, mosek.boundkey.lo, -s[2*i+1], np.inf) ; task.putarow(row, [f(2*i+1), p(
    row+=1 ; task.putconbound(row, mosek.boundkey.lo, s[2*i+1], np.inf) ; task.putarow(row, [f(2*i+1), p(

    row+=1 ; task.putconbound(row, mosek.boundkey.lo, 0.0 , np.inf); task.putarow(row, [p(2*i) , p(2*i
    row+=1 ; task.putconbound(row, mosek.boundkey.up, -np.inf , 0.0 ) ; task.putarow(row, [p(2*i) , p(2*i
    row+=1 ; task.putconbound(row, mosek.boundkey.lo, 0.0 , np.inf); task.putarow(row, [p(2*i+1), p(2*i
    row+=1 ; task.putconbound(row, mosek.boundkey.up, -np.inf , 0.0 ) ; task.putarow(row, [p(2*i+1), p(2*i

    # The most important constraint of all! On the ``L1 triangle''
    # time for drone to start from the truck reach site and get back to truck
    # = time for truck between the two successive rendezvous points
    row+=1; task.putconbound(row, mosek.boundkey.fx, 0.0 , 0.0 ) ;
    task.putarow(row, [b(2*i), b(2*i+1), f(2*i), f(2*i+1), h(2*i), h(2*i+1)], [1.0,1.0,1.0,1.0,-phi, -phi])

# Input the objective sense (minimize/maximize)

```

```

task.putobjsense(mosek.objsense.minimize)
task.optimize()
# Print a summary containing information
# about the solution for debugging purposes
#task.solutionsummary(mosek.streamtype.msg)

# Get status information about the solution
solsta = task.getsolsta(mosek.soltype.bas)

if (solsta == mosek.solsta.optimal or
    solsta == mosek.solsta.near_optimal):
    xx = [0.] * numvar
    # Request the basic solution.
    task.getxx(mosek.soltype.bas, xx)
    #print("Optimal solution: ")
    #for i in range(numvar):
    #    print("x[" + str(i) + "]=" + str(xx[i]))
elif (solsta == mosek.solsta.dual_infeas_cer or
      solsta == mosek.solsta.prim_infeas_cer or
      solsta == mosek.solsta.near_dual_infeas_cer or
      solsta == mosek.solsta.near_prim_infeas_cer):
    print("Primal or dual infeasibility certificate found.\n")
elif solsta == mosek.solsta.unknown:
    print("Unknown solution status")
else:
    print("Other solution status")

# Now that we have solved the LP
# We need to extract the ``p`` section of the vector
ps = xx[:2*numsites]
bs = xx[2*numsites:4*numsites]
fs = xx[4*numsites:6*numsites]
hs = xx[6*numsites:]

#####
# This commented out section is important to check how close to zero the waiting times
# are as calculated by the LP. To understand this, comment in this section and comment
# out the part using tghe L2 metric below it
#####
# horse_waiting_times = np.zeros(numsites)
# for i in range(numsites):
#     if i == 0 :
#         horse_time = abs(ps[0]-alpha) + abs(ps[1]-beta)
#         fly_time_to_site = 1.0/phi * (abs(s[0]-alpha) + abs(s[1]-beta))
#         fly_time_from_site = 1.0/phi * (abs(s[0]-ps[1]) + abs(s[1]-ps[1]))
#     else:
#         horse_time = abs(ps[2*i]-ps[2*i-2]) + abs(ps[2*i+1]-ps[2*i-1])
#     fly_time_to_site = 1.0/phi * ( abs(s[2*i]-ps[2*i-2]) + abs(s[2*i+1]-ps[2*i-1]) )
#     fly_time_from_site = 1.0/phi * ( abs(s[2*i]-ps[2*i]) + abs(s[2*i+1]-ps[2*i+1]) )
#     horse_waiting_times[i] = horse_time - (fly_time_to_site + fly_time_from_site)

horse_waiting_times = np.zeros(numsites)
for i in range(numsites):
    if i == 0 :
        horse_time = np.sqrt((ps[0]-alpha)**2 + (ps[1]-beta)**2)
        fly_time_to_site = 1.0/phi * np.sqrt((s[0]-alpha)**2 + (s[1]-beta)**2)
        fly_time_from_site = 1.0/phi * np.sqrt((s[0]-ps[1])**2 + (s[1]-ps[1])**2)
    else:
        horse_time = np.sqrt((ps[2*i]-ps[2*i-2])**2 + (ps[2*i+1]-ps[2*i-1])**2)
        fly_time_to_site = 1.0/phi * np.sqrt( (s[2*i]-ps[2*i-2])**2 + (s[2*i+1]-ps[2*i-1])**2 )
        fly_time_from_site = 1.0/phi * np.sqrt( (s[2*i]-ps[2*i])**2 + (s[2*i+1]-ps[2*i+1])**2 )

```

---

```

horse_waiting_times[i] = horse_time - (fly_time_to_site + fly_time_from_site)

tour_points = utils_algo.pointify_vector(ps)
return {'tour_points'      : tour_points,
        'horse_waiting_times': horse_waiting_times,
        'site_ordering'    : sites,
        'tour_length_with_waiting_time_included': tour_length_with_waiting_time_included(tour_points, hors

```

◇

Fragment defined by [28](#), [29](#), [32a](#), [34](#), [39a](#), [54](#), [57](#), [59](#).

Fragment referenced in [21a](#).

Uses: [tour\\_length\\_with\\_waiting\\_time\\_included](#) [60b](#).

## Algorithm: Greedy—Incremental Insertion

### Algorithmic Overview

**5.6.1** The greedy nearest neighbor heuristic described in [section 5.5](#) gives an  $O(\log n)$  approximation for  $n$  sites in the plane. However, there exists an alternative greedy incremental insertion algorithm for the TSP that yields a 2-approximation. Similar to the greedy-nn algorithm we can generalize the greedy-incremental approach to the collinear-horseflies setting (cf: [Figure 5.1](#)).

**5.6.2** In this approach, we maintain a list of visited sites  $V$  (along with the order of visitation  $\mathcal{O}$ ) and the unvisited sites  $U$ . For the given collinear-horsefly tour serving  $V$  pick a site  $s$  from  $U$  along with a position in  $\mathcal{O}$  (calling the resulting ordering  $\mathcal{O}'$ ) that minimizes the cost of the horsefly tour serving the sites  $V \cup \{s\}$  in the order  $\mathcal{O}'$ .

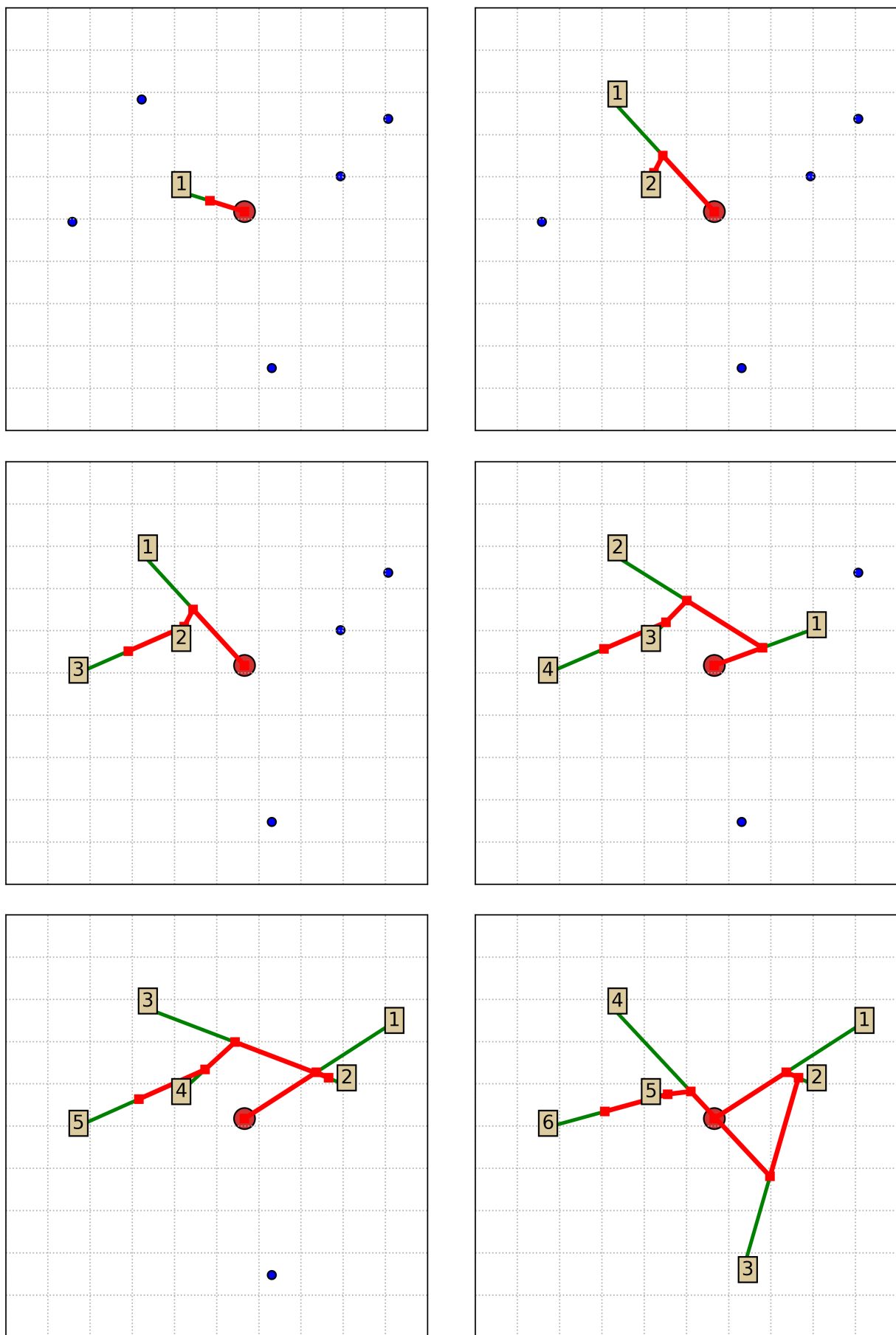


Figure 5.2: Greedy incremental insertion for collinear horseflies.  $\varphi = 3.0$ . Notice that the ordering of the visited sites keep changing based on where we decide to insert an unvisited site.

Figure 5.2 depicts the incremental insertion process for the case of 4 sites and  $\varphi = 3$ . Notice that the ordering of the visited sites keep changing based on where we decide to insert an unvisited site.

The implementation of this algorithm for collinear-horseflies raises several interesting non-trivial data-structural questions in their own right: how to quickly find the site from  $U$  to insert into  $V$ , and keep track the changing length of the horsefly tour. Note that inserting a site causes the length of the tour of the truck to change, for all the sites after  $s$ .

## Algorithmic Details

**5.6.3** The implementation of the algorithm is “parametrized” over various strategies for insertion. i.e. we treat each insertion policy as a black-box argument to the function.

Efficient policies for detecting the exact or approximate point for cheapest insertion will be described in section 5.7. We also implement a “naive” policy as a way benchmark the quality and speed of implementation of future insertion policies.

*⟨ Algorithms for classic horsefly 39a ⟩ ≡*

```

⟨ Define auxiliary helper functions 45c, ... ⟩
⟨ Define various insertion policy classes 47a ⟩
def algo_greedy_incremental_insertion(sites, inithorseposn, phi,
                                     insertion_policy_name      = "naive",
                                     write_algo_states_to_disk_p = True ,
                                     animate_schedule_p          = True  ,
                                     post_optimizer              = None):
    ⟨ Set log, algo-state and input-output files config 39b ⟩
    ⟨ Set insertion policy class for current run 40a ⟩

    while insertion_policy.unvisited_sites_idx:
        ⟨ Use insertion policy to find the cheapest site to insert into current tour 40b ⟩
        ⟨ Write algorithms current state to file 41a ⟩

        ⟨ Write input and output to file 44a ⟩
        ⟨ Make an animation of the schedule, if animate_schedule_p == True 45a ⟩
        #sys.exit()
        ⟨ Make an animation of algorithm states, if write_algo_states_to_disk_p == True 44b ⟩
        ⟨ Return horsefly tour, along with additional information 45b ⟩
    ◇

```

Fragment defined by 28, 29, 32a, 34, 39a, 54, 57, 59.

Fragment referenced in 21a.

Defines: `algo_greedy_incremental_insertion`, 23, `write_algo_states_to_disk_p` 29, 41ab, 44b.

**5.6.4** Note that for each run of the algorithm, we create a dedicated directory and use a corresponding log file in that directory. It will typically contain detailed information on the progress of the algorithm and the steps executed.

For algorithm analysis, and verification of correctness, on the other hand, we will typically be interested in the states of the data-structures at the end of the while loop; each such state will be written out as a YAML file. Such files can be useful for animating the progress of the algorithm.

Finally, just before returning the answer, we write the input and output to a separate YAML file. All in all, there are three “types” of output files within each directory that corresponds to an algorithm’s run: a log file, algorithm states files, and finally an input-output file.

*⟨ Set log, algo-state and input-output files config 39b ⟩ ≡*

```

import sys, logging, datetime, os, errno

algo_name      = 'algo-greedy-incremental-insertion'
time_stamp     = datetime.datetime.now().strftime('Day-%Y-%m-%d_ClockTime-%H:%M:%S')

```

```

dir_name      = algo_name + '---' + time_stamp
log_file_name = dir_name + '/' + 'run.log'
io_file_name  = 'input_and_output.yml'

# Create directory for writing data-files and logs to for
# current run of this algorithm
try:
    os.makedirs(dir_name)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise

logging.basicConfig( filename = log_file_name,
                    level    = logging.DEBUG,
                    format   = '%(asctime)s: %(levelname)s: %(message)s',
                    filemode = 'w' )

#logger = logging.getLogger()
info("Started running greedy_incremental_insertion for classic horsefly")

algo_state_counter = 0
◇

```

Fragment referenced in 39a.

Defines: `io_file_name`, 31b, 44a, `logger` 22a, 30a.

Uses: `greedy` 31a.

**5.6.5** This fragment merely sets the variable `insertion_policy` to the appropriate function. This will later help us in studying the speed of the algorithm and quality of the solution for various insertion policies during the experimental analysis.

*⟨ Set insertion policy class for current run 40a ⟩ ≡*

```

if insertion_policy_name == "naive":
    insertion_policy = PolicyBestInsertionNaive(sites, inithorseposn, phi)
else:
    print insertion_policy_name
    sys.exit("Unknown insertion policy: ")
debug("Finished setting insertion policy: " + insertion_policy_name)
◇

```

Fragment referenced in 39a.

**5.6.6** Note that while defining the body of the algorithm, we treat the insertion policy (whose name has already been passed as a string argument) as a kind of black-box, since all policy classes have the same interface. The detailed implementation for the various insertion policies are given later.

*⟨ Use insertion policy to find the cheapest site to insert into current tour 40b ⟩ ≡*

```

insertion_policy.insert_another_unvisited_site()
debug(Fore.GREEN + "Inserted another unvisited site" + Style.RESET_ALL)
◇

```

Fragment referenced in 39a.

**5.6.7** When using Python 2.7 (as I am doing with this suite of programs), you should have the `pyyaml` module version 3.12 installed. Version 4.1 breaks for some weird reason; it can't seem to serialize Numpy objects. See <https://github.com/kevin1024/vcrpy/issues/366> for a brief discussion on this topic.

The version of `pyyaml` on your machine can be checked by printing the value of `yaml.__version__`. To install the correct version of `pyyaml` (if you get errors) use

```
sudo pip uninstall pyyaml && sudo pip install pyyaml=3.12
```



**5.6.8** We use the `write_algo_states_to_disk_p` boolean argument to explicitly specify whether to write the current algorithm state along with its image to disk or not. This is because Matplotlib and PyYaml is very slow when writing image files to disk. Later on, I will probably switch to Asymptote for all my plotting, but for the moment I will stick to Matplotlib because I don't want to have to switch languages right now.

And much of my plots will be of a reasonably high-quality for the purpose of presentations. This will naturally affect timing/benchmarking results.

*⟨ Write algorithms current state to file 41a ⟩ ≡*

```
if write_algo_states_to_disk_p:
    import yaml
    algo_state_file_name = 'algo_state_' + \
        str(algo_state_counter).zfill(5) + \
        '.yaml'

    data = {'insertion_policy_name' : insertion_policy_name,
            'unvisited_sites'      : [insertion_policy.sites[u] \
                                     for u in insertion_policy.unvisited_sites_idx],
            'visited_sites'        : insertion_policy.visited_sites,
            'horse_tour'           : insertion_policy.horse_tour }

    with open(dir_name + '/' + algo_state_file_name, 'w') as outfile:
        yaml.dump( data, \
                    outfile, \
                    default_flow_style = False)
    ⟨ Render current algorithm state to image file 41b ⟩

    algo_state_counter = algo_state_counter + 1
    debug("Dumped algorithm state to " + algo_state_file_name)
◇
```

Fragment referenced in [39a](#).

Uses: `write_algo_states_to_disk_p` [39a](#).

*⟨ Render current algorithm state to image file 41b ⟩ ≡*

```
import utils_algo
if write_algo_states_to_disk_p:
    ⟨ Set up plotting area and canvas, fig, ax, and other configs 41c ⟩
    ⟨ Extract x and y coordinates of the points on the horse, fly tours, visited and unvisited sites 42a ⟩
    ⟨ Mark initial position of horse and fly boldly on canvas 42b ⟩
    ⟨ Place numbered markers on visited sites to mark the order of visitation explicitly 43b ⟩
    ⟨ Draw horse and fly-tours 43a ⟩
    ⟨ Draw unvisited sites as filled blue circles 43c ⟩
    ⟨ Give metainformation about current picture as headers and footers 43d ⟩
    ⟨ Write image file 43e ⟩
◇
```

Fragment referenced in [41a](#).

Uses: `write_algo_states_to_disk_p` [39a](#).

## 5.6.9

*⟨ Set up plotting area and canvas, fig, ax, and other configs 41c ⟩ ≡*

```
from matplotlib import rc
rc('font', **{'family': 'serif', \
              'serif': ['Computer Modern']})
rc('text', usetex=True)
fig, ax = plt.subplots()
ax.set_xlim([0,1])
```

```

ax.set_ylim([0,1])
ax.set_aspect(1.0)
ax = fig.gca()
ax.set_xticks(np.arange(0, 1, 0.1))
ax.set_yticks(np.arange(0, 1., 0.1))
plt.grid(linestyle='dotted')
ax.set_xticklabels([]) # to remove those numbers at the bottom
ax.set_yticklabels([])

ax.tick_params(
    bottom=False,      # ticks along the bottom edge are off
    left=False,        # ticks along the top edge are off
    labelbottom=False) # labels along the bottom edge are off

```

Fragment referenced in [41b](#).

### 5.6.10 Matplotlib typically plots points using x and y coordinates of the points in separate points.

*( Extract x and y coordinates of the points on the horse, fly tours, visited and unvisited sites 42a )*  $\equiv$

```

# Route for the horse
xhs = [ data['horse_tour'][i][0] \
        for i in range(len(data['horse_tour'])) ]
yhs = [ data['horse_tour'][i][1] \
        for i in range(len(data['horse_tour'])) ]

# Route for the fly. The fly keeps alternating between the site and the horse
xfs , yfs = [xhs[0]], [yhs[0]]
for site, pt in zip (data['visited_sites'],
                    data['horse_tour'][1:]):
    xfs.extend([site[0], pt[0]])
    yfs.extend([site[1], pt[1]])

xvisited = [ data['visited_sites'][i][0] \
              for i in range(len(data['visited_sites'])) ]
yvisited = [ data['visited_sites'][i][1] \
              for i in range(len(data['visited_sites'])) ]

xunvisited = [ data['unvisited_sites'][i][0] \
               for i in range(len(data['unvisited_sites'])) ]
yunvisited = [ data['unvisited_sites'][i][1]
               for i in range(len(data['unvisited_sites'])) ]
debug("Extracted x and y coordinates for route of horse, fly, visited and unvisited sites")

```

Fragment referenced in [41b](#).

### 5.6.11

*( Mark initial position of horse and fly boldly on canvas 42b )*  $\equiv$

```

ax.add_patch( mpl.patches.Circle( inithorseposn, \
                                  radius = 1/55.0,\
                                  facecolor= '#D13131', #'red',\
                                  edgecolor='black') )
debug("Marked the initial position of horse and fly on canvas")

```

Fragment referenced in [41b](#).

---

```

< Draw horse and fly-tours 43a > ≡
    ax.plot(xfs,yfs,'g-',linewidth=1.1)
    ax.plot(xhs, yhs, color='r', \
            marker='s', markersize=3, \
            linewidth=1.6)
    debug("Plotted the horse and fly tours")
    ◇

```

Fragment referenced in [41b](#).

```

< Place numbered markers on visited sites to mark the order of visitation explicitly 43b > ≡
    for x,y,i in zip(xvisited, yvisited, range(len(xvisited))):
        ax.text(x, y, str(i+1), fontsize=8, \
                bbox=dict(facecolor='#ddcba0', alpha=1.0, pad=2.0))
    debug("Placed numbered markers on visited sites")
    ◇

```

Fragment referenced in [41b](#).

```

< Draw unvisited sites as filled blue circles 43c > ≡
    for x, y in zip(xunvisited, yunvisited):
        ax.add_patch( mpl.patches.Circle( (x,y),\
                                           radius    = 1/100.0,\
                                           facecolor = 'blue',\
                                           edgecolor = 'black') )

    debug("Drew univisted sites")
    ◇

```

Fragment referenced in [41b](#).

## 5.6.12

```

< Give metainformation about current picture as headers and footers 43d > ≡
    fontsize = 15
    ax.set_title( r'Number of sites visited so far: ' +\
                  str(len(data['visited_sites'])) +\
                  '/' + str(len(sites))          , \
                  fontdict={'fontsize':fontsize})
    ax.set_xlabel(r'$\varphi$'+str(phi), fontdict={'fontsize':fontsize})
    debug("Setting title, headers, footers, etc...")
    ◇

```

Fragment referenced in [41b](#).

Note that after writing image files, you should close the current figure. Otherwise the collection of all the open figures starts hogging the RAM. Matplotlib throws a warning to this effect (if you don't close to the figures) after writing about 20 figures:

```

/usr/local/lib/python2.7/dist-packages/matplotlib/pyplot.py:528: RuntimeWarning:
More than 20 figures have been opened. Figures created through the pyplot interface
('matplotlib.pyplot.figure') are retained until explicitly closed and may consume
too much memory. (To control this warning, see the rcParam 'figure.max_open_warning').
max_open_warning, RuntimeWarning)

```

There is a Stack Overflow answer (<https://stackoverflow.com/a/21884375/505306>) which advises to call `plt.close()` after writing out a file that closes the *current* figure to avoid the above warning.

```

< Write image file 43e > ≡

```

```

image_file_name = 'algo_state_' + \
    str(algo_state_counter).zfill(5) + \
    '.png'
plt.savefig(dir_name + '/' + image_file_name, \
    bbox_inches='tight', dpi=250)
print "Wrote " + image_file_name + " to disk"
plt.close()
debug(Fore.BLUE+"Rendered algorithm state to image file"+Style.RESET_ALL)
◇

```

Fragment referenced in 41b.

**5.6.13** The final answer is written to disk in the form of a YAML file. It lists the input sites in the order of visitation computed by the algorithm and gives the tour of the horse. Note that the number of points on the horse's tour is 1 more than the number of given sites.

*⟨ Write input and output to file 44a ⟩ ≡*

```

# ASSERT: `inithorseposn` is included as first point of the tour
assert(len(insertion_policy.horse_tour) == len(insertion_policy.visited_sites) + 1)

# ASSERT: All sites have been visited. Simple sanity check
assert(len(insertion_policy.sites) == len(insertion_policy.visited_sites))

data = {'insertion_policy_name' : insertion_policy_name ,
        'visited_sites'       : insertion_policy.visited_sites ,
        'horse_tour'          : insertion_policy.horse_tour ,
        'phi'                  : insertion_policy.phi ,
        'inithorseposn'       : insertion_policy.inithorseposn}

import yaml
with open(dir_name + '/' + io_file_name, 'w') as outfile:    yaml.dump( data, \
    outfile, \
    default_flow_style=False)
debug("Dumped input and output to " + io_file_name)
◇

```

Fragment referenced in 39a.

Uses: io\_file\_name, 39b.

**5.6.14** If algorithm states have been rendered to files in the run-folder, we stitch them together using ffmpeg and make an .mp4 animation of the changing states of the algorithms. The .mp4 file will be in the algorithm's run folder. I used the tutorial given on [https://en.wikibooks.org/wiki/FFMPEG\\_An\\_Intermediate\\_Guide/image\\_sequence](https://en.wikibooks.org/wiki/FFMPEG_An_Intermediate_Guide/image_sequence) for choosing the particular command-line options to ffmpeg below. The options -hide\_banner -loglevel panic to quieten ffmpeg's output were suggested by <https://superuser.com/a/1045060/102371>

*⟨ Make an animation of algorithm states, if write\_algo\_states\_to\_disk\_p == True 44b ⟩ ≡*

```

if write_algo_states_to_disk_p:
    import subprocess, os
    os.chdir(dir_name)
    subprocess.call( ['ffmpeg', '-hide_banner', '-loglevel', 'verbose', \
        '-r', '1', '-i', 'algo_state_%05d.png', \
        '-vcodec', 'mpeg4', '-r', '10', \
        'algo_state_animation.avi'] )
    os.chdir('../')
◇

```

Fragment referenced in 39a.

Uses: write\_algo\_states\_to\_disk\_p 39a.

**5.6.15** This chunk reads the information in the input-output file just written out as a YAML file in the run-folder and then renders the process of the horse and fly moving around the plane delivering packages to sites.

*⟨ Make an animation of the schedule, if animate\_schedule\_p == True 45a ⟩ ≡*

```

    if animate_schedule_p :
        animateSchedule(dir_name + '/' + io_file_name)
    ◇

```

Fragment referenced in [39a](#).

## 5.6.16

*⟨ Return horsefly tour, along with additional information 45b ⟩ ≡*

```

debug("Returning answer")
horse_waiting_times = np.zeros(len(sites)) # TODO write this to file later
return {'tour_points'          : insertion_policy.horse_tour[1:],
        'horse_waiting_times'   : horse_waiting_times,
        'site_ordering'        : insertion_policy.visited_sites,
        'tour_length_with_waiting_time_included': \
                                tour_length_with_waiting_time_included(\
                                    insertion_policy.horse_tour[1:], \
                                    horse_waiting_times, \
                                    inithorseposn)}
    ◇

```

Fragment referenced in [39a](#).

Uses: `tour_length_with_waiting_time_included` [60b](#).

**5.6.17** We now define some of the functions that were referred to in the above chunks. Given the initial position of the truck and drone, and a list of sites, we need to compute the collinear horsefly tour length for the given ordering. This is the function that is used in every policy class while deciding which is the cheapest unvisited site to insert into the current ordering of visited sites.

Note that the order in which sites are passed to this function matters. It assumes that you want to compute the collinear horseflies tour length for the sites *in the given order*.

For this, we use the formula for computing the rendezvous point when there is only a single site, given by the code-chunk below.

*⟨ Define auxiliary helper functions 45c ⟩ ≡*

```

def single_site_solution(site, horseposn, phi):

    h = np.asarray(horseposn)
    s = np.asarray(site)

    hs_mag = 1.0/np.linalg.norm(s-h)
    hs_unit = 1.0/hs_mag * (s-h)

    r      = h + 2*hs_mag/(1+phi) * hs_unit # Rendezvous point
    hr_mag = np.linalg.norm(r-h)

    return (tuple(r), hr_mag)
    ◇

```

Fragment defined by [45c](#), [46ab](#).

Fragment referenced in [39a](#).

Defines: `single_site_solution` [46ab](#), [53](#).

With that the tour length functions for collinear horseflies can be implemented as an elementary instance of the fold pattern of functional programming. <sup>1</sup>

*⟨ Define auxiliary helper functions 46a ⟩ ≡*

```
def compute_collinear_horseflies_tour_length(sites, horseposn, phi):

    if not sites: # No more sites, left to visit!
        return 0
    else:         # Some sites are still left on the itinerary

        (rendezvous_pt, horse_travel_length) = single_site_solution(sites[0], horseposn, phi )
        return horse_travel_length + \
            compute_collinear_horseflies_tour_length( sites[1:], rendezvous_pt, phi )

    ◇
```

Fragment defined by [45c](#), [46ab](#).

Fragment referenced in [39a](#).

Defines: `compute_collinear_horseflies_tour_length` [48ac](#).

Uses: `single_site_solution` [45c](#).

*⟨ Define auxiliary helper functions 46b ⟩ ≡*

```
def compute_collinear_horseflies_tour(sites, inithorseposn, phi):

    horseposn          = inithorseposn
    horse_tour_points = [inithorseposn]

    for site in sites:
        (rendezvous_pt, _) = single_site_solution(site, horseposn, phi )

        horse_tour_points.append(rendezvous_pt)
        horseposn = rendezvous_pt

    return horse_tour_points

    ◇
```

Fragment defined by [45c](#), [46ab](#).

Fragment referenced in [39a](#).

Defines: `compute_collinear_horseflies_tour` [49](#).

Uses: `single_site_solution` [45c](#).

## Insertion Policies

We have finished implemented the entire algorithm, except for the implementation of the various insertion policy classes.

The main job of an insertion policy class is to keep track of the unvisited sites, the order of the visited sites and the horsefly tour itself. Every time, the method `.get_next_site(...)` is called, it chooses an appropriate (i.e. cheapest) unvisited site to insert into the current ordering, and update the set of visited and unvisited sites and details of the horsefly tour.

To do this quickly it will typically need auxiliary data-structures whose specifics will depend on the details of the policy chosen.

**5.7.1 Naive Insertion** First, a naive implementation of the cheapest insertion heuristic, that will be useful in future benchmarking of running times and solution quality for implementations that are quicker but make more sophisticated uses of data-structures.

---

<sup>1</sup>Python has folds tucked away in some corner of its standard library. But I am not using it during the first hacky portion of this draft. Also Shane mentioned it has performance issues? Double-check this later!

In this policy for each unvisited site we first find the position in the current tour, which after insertion into that position amongst the visited sites yields the smallest increase in the collinear-horseflies tour-length.

Then we pick the unvisited site which yields the overall smallest increase in tour-length and insert it into its computed position from its previous paragraph.

Clearly this implementation and has at least quadratic running time. Later on, we will be investigating algorithms and data-structures for speeding up this operation.

The hope is to be able to find a dynamic data-structure to perform this insertion in logarithmic time. Variations on tools such as the well-separated pair decomposition might help achieve this goal. Jon Bentley used kd-trees to perform the insertion in his experimental TSP paper, but he wasn't dealing with the shifting tour structure as we have in horseflies. Also he did not deal with the question of finding an approximate point for insertion. These

**5.7.2** Since the interface for all policy classes will be the same, it is best, if have a base class for such classes. Since the details of the interface may change, I'll probably do this later. For now, I'll just keep all the policy classes completely separate while keeping the interface of the constructors and methods the same. I'll refactor things later.

The plan in that case should be to make an abstract class that has an abstract method called `insert_unvisited_site` and three data-fields made from the base-constructor named `sites`, `inithorseposn` and `phi`. Classes which inherit this abstract base class, will add their own local data-members and methods for keeping track of data for insertion.

*⟨ Define various insertion policy classes 47a ⟩*  $\equiv$

```
class PolicyBestInsertionNaive:

    def __init__(self, sites, inithorseposn, phi):

        self.sites          = sites
        self.inithorseposn  = inithorseposn
        self.phi            = phi

        self.visited_sites  = []                # The actual list of visited sites (not indices)
        self.unvisited_sites_idx = range(len(sites)) # This indexes into self.sites
        self.horse_tour      = [self.inithorseposn]
```

*⟨ Methods for PolicyBestInsertionNaive 47b ⟩*

◇

Fragment referenced in 39a.

Defines: `self.horse_tour` 49, `self.inithorseposn`, 26cd, 48ac, 49, `self.sites`, 26cd, `self.visited_sites`, 48a, 49.

### 5.7.3

*⟨ Methods for PolicyBestInsertionNaive 47b ⟩*  $\equiv$

```
def insert_another_unvisited_site(self):
    ⟨ Compute the length of the tour that currently services the visited sites 48a ⟩
    delta_increase_least_table = [] # tracking variable updated in for loop below

    for u in self.unvisited_sites_idx:
        ⟨ Set up tracking variables local to this iteration 48b ⟩
        ⟨ If self.sites[u] is chosen for insertion, find best insertion position and update delta_increase_least_table 48c ⟩

        ⟨ Find the unvisited site which on insertion increases tour-length by the least amount 48d ⟩
        ⟨ Update states for PolicyBestInsertionNaive 49 ⟩
```

◇

Fragment referenced in 47a.

Defines: `delta_increase_least_table` 48cd.

## 5.7.4

*⟨ Compute the length of the tour that currently services the visited sites 48a ⟩ ≡*

```
current_tour_length = \
    compute_collinear_horseflies_tour_length(\
        self.visited_sites,\
        self.inithorseposn,\
        self.phi)
```

◇

Fragment referenced in 47b.

Defines: `current_tour_length` 48c.

Uses: `compute_collinear_horseflies_tour_length` 46a, `self.inithorseposn`, 47a, `self.visited_sites`, 47a.

## 5.7.5

*⟨ Set up tracking variables local to this iteration 48b ⟩ ≡*

```
ibest = 0
delta_increase_least = float("inf")
```

◇

Fragment referenced in 47b.

Defines: `delta_increase_least` 48c, `ibest`, 48c.

## 5.7.6

*⟨ If self.sites[u] is chosen for insertion, find best insertion position and update delta\_increase\_least\_table 48c ⟩ ≡*

```
for i in range(len(self.sites)):

    visited_sites_test = self.visited_sites[:i] + \
        [ self.sites[u] ] + \
        self.visited_sites[i:]

    tour_length_on_insertion = \
        compute_collinear_horseflies_tour_length(\
            visited_sites_test,\
            self.inithorseposn,\
            self.phi)

    delta_increase = tour_length_on_insertion - current_tour_length
    assert(delta_increase >= 0)

    if delta_increase < delta_increase_least:
        delta_increase_least = delta_increase
        ibest = i

    delta_increase_least_table.append({'unvisited_site_idx' : u, \
        'best_insertion_position' : ibest, \
        'delta_increase' : delta_increase_least})
```

◇

Fragment referenced in 47b.

Uses: `compute_collinear_horseflies_tour_length` 46a, `current_tour_length` 48a, `delta_increase_least` 48b, `delta_increase_least_table` 47b, `ibest`, 48b, `self.inithorseposn`, 47a.

## 5.7.7

*⟨ Find the unvisited site which on insertion increases tour-length by the least amount 48d ⟩ ≡*



```

best_table_entry = min(delta_increase_least_table, \
                        key = lambda x: x['delta_increase'])

unvisited_site_idx_for_insertion = best_table_entry['unvisited_site_idx']
insertion_position                = best_table_entry['best_insertion_position']
delta_increase                    = best_table_entry['delta_increase']
◇

```

Fragment referenced in [47b](#).

Uses: `delta_increase_least_table` [47b](#).

## 5.7.8

⟨ *Update states for PolicyBestInsertionNaive* 49 ⟩ ≡

```

# Update visited and unvisited sites info
self.visited_sites = self.visited_sites[:insertion_position] + \
                    [self.sites[unvisited_site_idx_for_insertion]] + \
                    self.visited_sites[insertion_position:]

self.unvisited_sites_idx = filter( lambda elt: elt != unvisited_site_idx_for_insertion, \
                                   self.unvisited_sites_idx )

# Update the tour of the horse
self.horse_tour = compute_collinear_horseflies_tour(\
                self.visited_sites, \
                self.inithorseposn, \
                self.phi)
◇

```

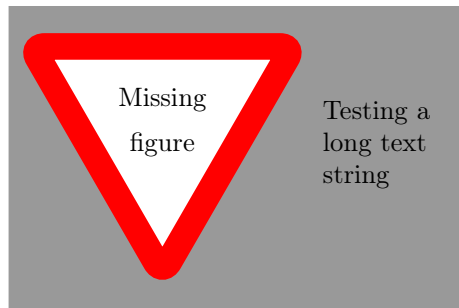
Fragment referenced in [47b](#).

Uses: `compute_collinear_horseflies_tour` [46b](#), `self.horse_tour` [47a](#), `self.inithorseposn`, [47a](#), `self.visited_sites`, [47a](#).

# Lower Bound: The $\varphi$ -Prim-MST

**Overview** To compare the experimental performance of algorithms for NP-hard optimization problems wrt solution quality, it helps to have a cheaply computable lower bound that acts as a proxy for OPT. In the case of the TSP, a lower bound is the weight of the minimum spanning tree on the set of input sites.

To compute the MST on a set of points, one typically uses greedy algorithms such as those by Prim, Kruskal or Boruvka. To get a lower-bound for Horsefly, we define a network that we call the  $\varphi$ -Prim-MST by a simple generalization of Prim. Currently, we don't have a natural interpretation of this structure means in terms of the sites. This is something we need to add to our TODO list.



This is clearly a lower-bound on the weight of *OPT* for Collinear Horsefly. However, I believe that the stronger statement is also true

**Conjecture 1.** *The weight of the  $\varphi$ -MST is a lower-bound on the length of the horse's tour in OPT for the classic horsefly problem.*

The proof of this conjecture seems to be non-trivial off-hand. I'll put a hold on all my attempts so far to prove this, since I want the experiments to guide my intuition here.

It is possible that there could be other lower bounds based on generalizing the steps in Kruskal's and Boruvka's algorithms. Based on the experimental success of the  $\varphi$ -MST's, I will think of the appropriate generalizations for them later.

One particular experiment that I would be interested would be how bad is to check the crossing structure of the edges. In the MST edges never cross. What is the structure of the crossing in  $\varphi$ -MSTs? That might help me in designing a local search operation for the Horsefly problem.

Also note, that the construction of this  $\varphi$ -Prim MST can be generalized to two or more flies (single horse) we build two separate trees; with two or more drones since we are interested in minimizing the makespan, probably we greedily them so that the trees are well-balanced.....????? dunno doesn't strike as clean now that I think of it. It certainly isn't as clean as my node-splitting horsefly framework. Hopefully, I can prove some sort of theorems on those later?

As I type this, a separate question strikes me to be of independent interest: *Given a point-cloud in the plane, preprocess the points such that for a query  $\varphi$  we can compute the  $\varphi$ -MST in linear time.* Perhaps the MST, itself could be useful for this augmented with some data-structures for performing ray-shooting in an arrangement of segments. One can use such a data-structure, for making a quick animation of the evolution of the  $\varphi$ -MST as we keep changing the  $\varphi$ -parameter, as one often does while playing with Mathematica's `Manipulate` function. Can we motivate this by saying  $\varphi$  might be uncertain? I don't know, people would only find this interesting if the particular data-structure helps in the computation of horsefly like tours.

## Computing the $\varphi$ -Prim-MST

**5.8.1** For the purposes of this section we define the notion of a rendezvous point for an edge. Given a directed segment  $\overrightarrow{XY}$  and a speed ratio  $\varphi$ , assume a horse and a fly are positioned at  $X$  and there is a site that needs to be serviced at  $Y$ . The rendezvous point of  $\overrightarrow{XY}$  is that point along  $R$  at which the horse and fly meet up at the earliest after the fly leaves  $X$ . Explicit formulae for computing this point have already been implemented in `single_site_solution`, in one of the previous sections.

**5.8.2** Prim’s algorithm for computing MSTs is essentially a greedy incremental insertion process. The same structure is visible in the code fragment below. The only essential change from Prim’s original algorithm is that we “grow” the tree only from the rendezvous points computed while inserting a new edge into the existing partial tree on the set of sites. This process is animated in ??

I have will be using the NetworkX library (<https://networkx.github.io/>) for storing and manipulating graphs. For performing efficient nearest-neighbor searches for each rendezvous point in the partially constructed MST, I will use the scikit-learn library (<https://scikit-learn.org/stable/modules/neighbors.html>). When porting my codes to C++, I will probably have to switch over to the Boost Graph library and David Mount’s ANN for the same purposes(both these libraries have been optimized for speed).

In the while loop below, `node_site_info` stores a tuple for each node in the tree consisting of

1. a node-id (this corresponds to a rendezvous point in the tree)
2. the index of the closest site in the array `sites` for the node (the site)
3. distance of the node to the site with the above index.

*< Lower bounds for classic horsefly 51 >  $\equiv$*

```
def compute_phi_prim_mst(sites, inithorseposn,phi):

    import networkx as nx
    from sklearn.neighbors import NearestNeighbors

    < Create singleton graph, with node at inithorseposn 52a >

    unmarked_sites_idx = range(len(sites))
    while unmarked_sites_idx:
        node_site_info = []

        < For each node, find the closest site 52b >
        < Find the node with the closest site, and generate the next node and edge for the  $\varphi$ -MST 53 >

        # Marking means removing from unmarked list :-D
        unmarked_sites_idx.remove(next_site_to_mark_idx)

    utils_algo.print_list(G.nodes.data())
    utils_algo.print_list(G.edges.data())
    return G

    ◇
```

Fragment referenced in 21a.

Defines: `compute_phi_prim_mst`, Never used, `unmarked_sites_idx` 52b.

**5.8.3** Every node in the tree stores its own id as an integer along with its X-Y coordinates and the X-Y coordinates of the sites that it will be joined to with a straight-line segment. At the beginning the single node of the tree at the initial position of the horse and fly has not been joined to any sites, and hence is empty.

⟨ *Create singleton graph, with node at inithorseposn* 52a ⟩ ≡

```
G = nx.Graph()
G.add_node(0, mycoordinates=inithorseposn, joined_site_coords=[])
◇
```

Fragment referenced in [51](#).

## 5.8.4

⟨ *For each node, find the closest site* 52b ⟩ ≡

```
for nodeid, nodeval in G.nodes.data():
    current_node_coordinates = np.asarray(nodeval['mycoordinates'])
    distances_of_current_node_to_sites = []

    # The following loop finds the nearest unmarked site. So far, I am
    # using brute force for this, later, I will use sklearn.neighbors.
    for j in unmarked_sites_idx:
        site_coordinates = np.asarray(sites[j])
        dist = np.linalg.norm( site_coordinates - current_node_coordinates )

        distances_of_current_node_to_sites.append( (j, dist) )

    nearest_site_idx, distance_of_current_node_to_nearest_site = \
        min(distances_of_current_node_to_sites, key=lambda (_, d): d)

    node_site_info.append((nodeid, \
                           nearest_site_idx, \
                           distance_of_current_node_to_nearest_site))
◇
```

Fragment referenced in [51](#).

Uses: `unmarked_sites_idx` [51](#).

## 5.8.5

*Find the node with the closest site, and generate the next node and edge for the  $\varphi$ -MST* 53  $\equiv$

```

    opt_node_idx, \
    next_site_to_mark_idx, \
    distance_to_next_site_to_mark = min(node_site_info, key=lambda (h,k,d) : d)

    tmp = sites[next_site_to_mark_idx]
    G.nodes[opt_node_idx]['joined_site_coords'].append( tmp )
    (r, h) = single_site_solution(tmp, G.nodes[opt_node_idx]['mycoordinates'], phi)

    # Remember! indexing of nodes started at 0, thats why you set
    # numnodes to the index of the newly inserted node.
    newnodeid = len(list(G.nodes))

    # joined_site_coords will be updated in the future iterations of while :
    G.add_node(newnodeid, mycoordinates=r, joined_site_coords=[])

    # insert the edge weight, will be useful later when
    # computing sum total of all the edges.
    G.add_edge(opt_node_idx, newnodeid, weight=h )
    ◇

```

Fragment referenced in 51.

Uses: `single_site_solution` 45c.

# Algorithm: Doubling the $\varphi$ -MST

## 5.9.1 Algorithmic Overview

## 5.9.2 Algorithmic Details

# Algorithm: Bottom-Up Split

## 5.10.1 Algorithmic Overview

## 5.10.2 Algorithmic Details

# Algorithm: Local Search—Swap

## 5.11.1 Algorithmic Overview

## 5.11.2 Algorithmic Details

# Algorithm: K2 Means

## 5.12.1 Algorithmic Overview

## 5.12.2 Algorithmic Details

## 5.12.3

*( Algorithms for classic horsefly 54 )*  $\equiv$

```
def algo_kmeans(sites, inithorseposn, phi, k, post_optimizer):
    """
    type Point    (Double, Double)
    type Site     Point
    type Cluster  (Point, [Site])
    type Tour     {'site_ordering':[Site],
                  'tour_points'  :[Point]}
    algo_kmeans :: [Site] -> Point -> Double -> Int
    """
    def get_clusters(site_list):
        """
        get_clusters :: [Site] -> [Cluster]
        For the given list of sites, perform k-means clustering
        and return the list of k-centers, along with a list of sites
        assigned to each center.
        """
        X      = np.array(site_list)
        kmeans = KMeans(n_clusters=k, random_state=0).fit(X)

        accum = [ (center, []) for center in kmeans.cluster_centers_ ]
        for label, site in zip(kmeans.labels_, site_list):
            accum[label][1].append(site)

        return accum

    def extract_cluster_sites_for_each_cluster(clusters):
        """
        extract_cluster_sites_for_each_cluster :: [Cluster] -> [[Site]]
        """
        return [ cluster_sites for (_, cluster_sites) in clusters ]

    def fuse_tours(tours):
        """
        fuse_tours :: [Tour] -> Tour
        """
        fused_tour = {'site_ordering':[], 'tour_points':[]}
        for tour, i in zip(tours, range(len(tours))):
            fused_tour['site_ordering'].extend(tour['site_ordering'])
            if i != len(tours)-1:
                # Remember! last point of previous tour is first point of
                # this tour, which is why we need to avoid duplication
                # Hence the[:-1]
                fused_tour['tour_points'].extend(tour['tour_points'][:-1])
            else:
                # Because this is the last tour in the iteration, we include
                # its end point also, hence no[:-1] here
                fused_tour['tour_points'].extend(tour['tour_points'])
        return fused_tour
```

---

```

def weighted_center_tour(clusters, horseflyinit):
    """
    weighted_center_tour :: [Cluster] -> Point -> [Cluster]

    Just return a permutation of the clusters.
    need to return actual weighted tour
    since we are only interested in the order
    in which the weighted center tour is performed
    on k weighted points, where k is the clustering
    number used here
    """

    #print Fore.CYAN, " Clusters: " , clusters, Style.RESET_ALL
    #print " "
    #print Fore.CYAN, " Horseflyinit: ", horseflyinit, Style.RESET_ALL

    assert( k == len(clusters) )
    tour_length_fn = tour_length(horseflyinit)

    #-----
    # For each of the k! permutations of the weighted sites
    # give the permutation with the smallest weighted tour
    # Note that k is typically small, say 2,3 or 4
    #-----
    # But first we initialize the accumulator variables prefixed with best_

    #print Fore.YELLOW , " Computing Weighted Center Tour ", Style.RESET_ALL
    clustering_centers = [ center          for (center, _)    in clusters]
    centers_weights    = [ len(site_list) for (_, site_list) in clusters]

    #utils_algo.print_list(clustering_centers)
    #utils_algo.print_list(centers_weights)
    #time.sleep(5000)

    best_perm = clusters
    best_perm_tour = algo_weighted_sites_given_specific_ordering(clustering_centers, \
                                                                centers_weights, \
                                                                horseflyinit, \
                                                                phi)

    i = 1
    for clusters_perm in list(itertools.permutations(clusters)):

        #print Fore.YELLOW , ".....Testing a new cluster permutation [ ", i , \
        #                    "/", math.factorial(k) , " ] of the sites", \
        #                    Style.RESET_ALL

        i = i + 1
        # cluster_centers_and_weights :: [(Point, Int)]
        # This is what is used for the weighted tour
        clustering_centers = [ center          for (center, _)    in clusters_perm]
        centers_weights    = [ len(site_list) for (_, site_list) in clusters_perm]

        tour_current_perm = \
            algo_weighted_sites_given_specific_ordering(clustering_centers, \
                                                        centers_weights, \
                                                        horseflyinit, \
                                                        phi)

        if tour_length_fn( utils_algo.flatten_list_of_lists(tour_current_perm ['tour_points']) ) \
            < tour_length_fn( utils_algo.flatten_list_of_lists( best_perm_tour ['tour_points']) ):

```

```

        print Fore.RED + ".....Found better cluster order" + Style.RESET_ALL
        best_perm = clusters_perm

    return best_perm

def get_tour (site_list, horseflyinit):
    """
    get_tour :: [Site] -> Point -> Tour

    A recursive function which does the job
    of extracting a tour
    """

    if len (site_list) <= k: # Base-case for the recursion
        #print Fore.CYAN + ".....Reached Recursion Base case" + Style.RESET_ALL
        result = algo_dumb(site_list, horseflyinit, phi)
        return result
    else: # The main recursion
        # Perform k-means clustering and get the clusters
        clusters = get_clusters(site_list)

        #utils_algo.print_list(clusters)

        #####
        # Permute the clusters depending on which is better to visit first
        clusters_perm = weighted_center_tour(clusters, horseflyinit)
        #####

        # Extract cluster sites for each cluster
        cluster_sites_for_each_cluster = \
            extract_cluster_sites_for_each_cluster(clusters_perm)

        # Apply the get_tour function on each chunk while folding across
        # using the last point of the tour of the previous cluster
        # as the first point of this current one. This is a kind of recursion
        # that pays forward.
        tours = []
        for site_list, i in zip(cluster_sites_for_each_cluster,
                                range(len(cluster_sites_for_each_cluster))):

            if i == 0: # first point is horseflyinit. The starting fold value!!
                tours.append( get_tour(site_list, inithorseposn) )
            else: # use the last point of the previous tour (i-1 index)
                # as the first point of this one !!
                prev_tour = tours[i-1]
                tours.append( get_tour(site_list, prev_tour['tour_points'][-1]))

        # Fuse the tours you obtained above to get a site ordering
        return fuse_tours(tours)

print Fore.MAGENTA + "RUNNING algo_kmeans....." + Style.RESET_ALL
sites1 = get_tour(sites, inithorseposn)['site_ordering']
return post_optimizer(sites1, inithorseposn, phi )

```

◇

Fragment defined by [28](#), [29](#), [32a](#), [34](#), [39a](#), [54](#), [57](#), [59](#).  
 Fragment referenced in [21a](#).  
 Uses: `tour_length` [60a](#).

## 5.12.4



⟨ *Algorithms for classic horsefly 57* ⟩ ≡

```
def algo_weighted_sites_given_specific_ordering (sites, weights, horseflyinit, phi):

    def site_constraints(i, sites, weights):
        """
        site_constraints :: Int -> [Site] -> [Double]
                        -> [ [Double] -> Double ]

        Generate a list of constraint functions for the ith site
        The number of constraint functions is equal to the weight
        of the site!
        """

        #print Fore.RED, sites, Style.RESET_ALL

        psum_weights = utils_algo.partial_sums( weights ) # partial sum of ALL the site-weights
        accum         = [ ]
        site_weight   = weights[i]

        for j in range(site_weight):

            if i == 0 and j == 0:

                #print "i= ", i, " j= ", j
                def _constraint_function(x):
                    """
                    constraint_function :: [Double] -> Double
                    """
                    start = np.array (horseflyinit)
                    site   = np.array (sites[0])
                    stop   = np.array ([x[0],x[1]])

                    horsetime = np.linalg.norm( stop - start )

                    flytime_to_site   = 1/phi * np.linalg.norm( site - start )
                    flytime_from_site = 1/phi * np.linalg.norm( stop - site )
                    flytime           = flytime_to_site + flytime_from_site
                    return horsetime-flytime

                accum.append( _constraint_function )

            elif i == 0 and j != 0 :

                #print "i= ", i, " j= ", j
                def _constraint_function(x):
                    """
                    constraint_function :: [Double] -> Double
                    """
                    start = np.array( [x[2*j-2], x[2*j-1]] )
                    site   = np.array(sites[0])
                    stop   = np.array( [x[2*j]   , x[2*j+1]] )

                    horsetime = np.linalg.norm( stop - start )

                    flytime_to_site   = 1/phi * np.linalg.norm( site - start )
                    flytime_from_site = 1/phi * np.linalg.norm( stop - site )
                    flytime           = flytime_to_site + flytime_from_site
                    return horsetime-flytime

                accum.append( _constraint_function )
```

```

else:

    #print "i= ", i, " j= ", j
    def _constraint_function(x):
        """
        constraint_function :: [Double] -> Double
        """

        offset = 2 * psum_weights[i-1]

        start = np.array( [ x[offset + 2*j-2 ], x[offset + 2*j-1 ] ] )
        site = np.array(sites[i])
        stop = np.array( [ x[offset + 2*j ] , x[offset + 2*j+1 ] ] )

        horsetime = np.linalg.norm( stop - start )

        flytime_to_site = 1/phi * np.linalg.norm( site - start )
        flytime_from_site = 1/phi * np.linalg.norm( stop - site )
        flytime = flytime_to_site + flytime_from_site
        return horsetime-flytime

    accum.append( _constraint_function )

return accum

def generate_constraints(sites, weights):
    return [site_constraints(i, sites, weights) for i in range(len(sites))]

#####
#print weights
#### For debugging
weights = [1 for wt in weights]
####

cons = utils_algo.flatten_list_of_lists (generate_constraints(sites, weights))
cons1 = [ {'type':'eq', 'fun':f} for f in cons]

# Since the horsely tour lies inside the square,
# the bounds for each coordinate is 0 and 1
x0 = np.empty(2*sum(weights))
x0.fill(0.5) # choice of filling vector with 0.5 is arbitrary

# Run scipy's minimization solver
sol = minimize(tour_length(horseflyinit), x0, method= 'SLSQP', constraints=cons1)
tour_points = utils_algo.pointify_vector(sol.x)

#print sol

#time.sleep(5000)
return {'tour_points' : tour_points,
        'site_ordering': sites}

```

◇

Fragment defined by [28](#), [29](#), [32a](#), [34](#), [39a](#), [54](#), [57](#), [59](#).

Fragment referenced in [21a](#).

Uses: [generate\\_constraints 33](#), [tour\\_length 60a](#).

# Algorithm: TSP ordering

## 5.13.1 Algorithmic Overview

## 5.13.2 Algorithmic Details

**5.13.3** Use the TSP ordering for the horsefly tour, irrespective of the speedratio. Useful to see the benefit obtained from the various heuristics you will be designing.

This will be especially useful for larger ratios of speeds

I use the tsp package for this: <https://pypi.org/project/tsp/#files> If the tsp ordering has already been pre-computed, then use it.

*( Algorithms for classic horsefly 59 )*  $\equiv$

```
def algo_tsp_ordering(sites, inithorseposn, phi, post_optimizer):
    import tsp
    horseinit_and_sites = [inithorseposn] + sites

    _, tsp_idxss = tsp.tsp(horseinit_and_sites)

    # Get the position of the horse in tsp_idxss
    h = tsp_idxss.index(0) # 0 because the horse was placed first in the above vector

    if h != len(tsp_idxss)-1:
        idx_vec = tsp_idxss[h+1:] + tsp_idxss[:h]
    else:
        idx_vec = tsp_idxss[:h]

    # idx-1 because all the indexes of the sites were pushed forward
    # by 1 when we tacked on inithorseposn at the very beginning
    # of horseinit_and_sites, hence we auto-correct for that
    sites_tsp = [sites[idx-1] for idx in idx_vec]

    tour0 = post_optimizer (sites_tsp, inithorseposn, phi)
    tour1 = post_optimizer (list(reversed(sites_tsp)), inithorseposn, phi)

    tour0_length = utils_algo.length_polygonal_chain([inithorseposn] + tour0['site_ordering'])
    tour1_length = utils_algo.length_polygonal_chain([inithorseposn] + tour1['site_ordering'])

    print Fore.RED, " TSP paths in either direction are ", tour0_length, " ", tour1_length, Style.RESET_ALL

    if tour0_length < tour1_length:
        print Fore.RED, "Selecting tour0 ", Style.RESET_ALL
        return tour0
    else:
        print Fore.RED, "Selecting tour1 ", Style.RESET_ALL
        return tour1
```

◇

Fragment defined by 28, 29, 32a, 34, 39a, 54, 57, 59.

Fragment referenced in 21a.

# Local Utility Functions

**5.14.1** For a given initial position of horse and fly return a function computing the tour length. The returned function computes the tour length in the order of the list of stops provided beginning with the initial position of horse and fly. Since the horse speed = 1, the tour length = time taken by horse to traverse the route.

This is in other words the objective function.

*( Local utility functions for classic horsefly 60a )*  $\equiv$

```
def tour_length(horseflyinit):
    def _tourlength (x):

        # the first point on the tour is the
        # initial position of horse and fly
        # Append this to the solution x = [x0,x1,x2,...]
        # at the front
        htour = np.append(horseflyinit, x)
        length = 0

        for i in range(len(htour))[:-3:2]:
            length = length + \
                np.linalg.norm([htour[i+2] - htour[i], \
                               htour[i+3] - htour[i+1]])

        return length

    return _tourlength
◇
```

Fragment defined by [60ab](#).

Fragment referenced in [21a](#).

Defines: `tour_length` [28](#), [32a](#), [54](#), [57](#), [63b](#), [64a](#).

**5.14.2** It is possible that some heuristics might return non-negligible waiting times. Hence I am writing a separate function which adds the waiting time (if it is positive) to the length of each link of the tour. Again note that because speed of horse = 1, we can add “time” to “distance”.

*( Local utility functions for classic horsefly 60b )*  $\equiv$

```
def tour_length_with_waiting_time_included(tour_points, horse_waiting_times, horseflyinit):
    tour_points = np.asarray([horseflyinit] + tour_points)
    tour_links = zip(tour_points, tour_points[1:])

    # the +1 because the initial position has been tacked on at the beginning
    # the solvers written the tour points except for the starting position
    # because that is known and part of the input. For this function
    # I need to tack it on for tour length
    assert(len(tour_points) == len(horse_waiting_times)+1)

    sum = 0
    for i in range(len(horse_waiting_times)):

        # Negative waiting times means drone/fly was waiting
        # at rendezvous point
        if horse_waiting_times[i] >= 0:
            wait = horse_waiting_times[i]
        else:
            wait = 0

        sum += wait + np.linalg.norm(tour_links[i][0] - tour_links[i][1], ord=2) #
    return sum
```

---

◇

Fragment defined by [60ab](#).

Fragment referenced in [21a](#).

Defines: `tour_length_with_waiting_time_included` [32a](#), [34](#), [45b](#), [63b](#).

# Plotting Routines

## 5.15.1

*Plotting routines for classic horsefly 62a*  $\equiv$

```
def plotTour(ax, horseflytour, horseflyinit, phi, algo_str, tour_color='#d13131'):

    < Get x and y coordinates of the endpoints of segments on the horse-tour 62b >
    < Get x and y coordinates of the sites 62c >
    < Construct the fly-tour from the information about horse tour and sites 62d >
    < Print information about the horse tour 62e >
    < Print information about the fly tour 63a >
    < Print meta-data about the algorithm run 63b >
    < Plot everything 64a >
    ◇
```

Fragment defined by [62a](#), [64b](#).

Fragment referenced in [21a](#).

Defines: plotTour [23](#).

## 5.15.2

*< Get x and y coordinates of the endpoints of segments on the horse-tour 62b >*  $\equiv$

```
xhs, yhs = [horseflyinit[0]], [horseflyinit[1]]
for pt in horseflytour['tour_points']:
    xhs.append(pt[0])
    yhs.append(pt[1])
    ◇
```

Fragment referenced in [62a](#).

## 5.15.3

*< Get x and y coordinates of the sites 62c >*  $\equiv$

```
xsites, ysites = [], []
for pt in horseflytour['site_ordering']:
    xsites.append(pt[0])
    ysites.append(pt[1])
    ◇
```

Fragment referenced in [62a](#).

## 5.15.4 Route for the fly keeps alternating between the site and the horse

*< Construct the fly-tour from the information about horse tour and sites 62d >*  $\equiv$

```
xfs , yfs = [xhs[0]], [yhs[0]]
for site, pt in zip (horseflytour['site_ordering'],
                    horseflytour['tour_points']):
    xfs.extend([site[0], pt[0]])
    yfs.extend([site[1], pt[1]])
    ◇
```

Fragment referenced in [62a](#).

## 5.15.5 Note that the waiting time at the starting point is 0

*< Print information about the horse tour 62e >*  $\equiv$

```
print "\n-----", "\nHorse Tour", "\n-----"
```

```

waiting_times = [0.0] + horseflytour['horse_waiting_times'].tolist()
#print waiting_times
for pt, time in zip(zip(xhs,yhs), waiting_times) :
    print pt, Fore.GREEN, " ---> Horse Waited ", time, Style.RESET_ALL
◇

```

Fragment referenced in [62a](#).

## 5.15.6

*⟨ Print information about the fly tour 63a ⟩ ≡*

```

print "\n-----", "\nFly Tour", "\n-----"
for item, i in zip(zip(xfs,yfs), range(len(xfs))):
    if i%2 == 0:
        print item
    else :
        print Fore.RED + str(item) + "----> Site" + Style.RESET_ALL
◇

```

Fragment referenced in [62a](#).

## 5.15.7

*⟨ Print meta-data about the algorithm run 63b ⟩ ≡*

```

print "-----"
print Fore.GREEN, "\nSpeed of the drone was set to be", phi
#tour_length = utils_algo.length_polygonal_chain( zip(xhs, yhs))
tour_length = horseflytour['tour_length_with_waiting_time_included']
print "Tour length of the horse is ", tour_length
print "Algorithm code-Key used " , algo_str, Style.RESET_ALL
print "-----\n"
◇

```

Fragment referenced in [62a](#).

Uses: [tour\\_length 60a](#), [tour\\_length\\_with\\_waiting\\_time\\_included 60b](#).

## 5.15.8

⟨ *Plot everything 64a* ⟩ ≡

```
#kwargs = {'size':'large'}
for x,y,i in zip(xsites, ysites, range(len(xsites))):
    ax.text(x, y, str(i+1), bbox=dict(facecolor='#ddcba0', alpha=1.0))

ax.plot(xfs,yfs,'g-')
ax.plot(xhs, yhs, color=tour_color, marker='s', linewidth=3.0)

ax.add_patch( mpl.patches.Circle( horseflyinit, radius = 1/140.0,
                                facecolor= '#D13131', edgecolor='black'    ) )

fontsize = 20

plt.rc('text', usetex=True)
plt.rc('font', family='serif')
ax.set_title( r'Algorithm Used: ' + algo_str + '\nTour Length: ' \
              + str(tour_length)[:7], fontdict={'fontsize':fontsize})
ax.set_xlabel(r'Number of sites: ' + str(len(xsites)) + '\nDrone Speed: ' + str(phi) ,
              fontdict={'fontsize':fontsize})
```

◇

Fragment referenced in [62a](#).

Uses: `tour_length` [60a](#).

## 5.15.9

⟨ *Plotting routines for classic horsefly 64b* ⟩ ≡

```
def draw_phi_mst(ax, phi_mst, inithorseposn, phi):

    # for each tree node draw segments joining to sites (green segs)
    for (nodeidx, nodeinfo) in list(phi_mst.nodes.data()):
        mycoords = nodeinfo['mycoordinates']
        joined_site_coords = nodeinfo['joined_site_coords']

        for site in joined_site_coords:
            ax.plot([mycoords[0],site[0]], [mycoords[1], site[1]], 'g-', linewidth=1.5)
            ax.add_patch( mpl.patches.Circle( [site[0],site[1]], radius = 0.007, \
                                              facecolor='blue', edgecolor='black'))

    # draw each tree edge (red segs)
    edges = list(phi_mst.edges.data())
    for (idx1, idx2, edgeinfo) in edges:
        (xn1, yn1) = phi_mst.nodes[idx1]['mycoordinates']
        (xn2, yn2) = phi_mst.nodes[idx2]['mycoordinates']
        ax.plot([xn1,xn2],[yn1,yn2], 'ro-', linewidth=1.7)

    ax.set_title(r'$\varphi$-MST', fontdict={'fontsize':30})
```

◇

Fragment defined by [62a](#), [64b](#).

Fragment referenced in [21a](#).

Defines: `draw_phi_mst` [23](#).



# Animation routines

**5.16.1** After writing out the schedule, it would be nice to have a function that animates the delivery process of the schedule. Every problem will have animation features unique to its features. Any abstraction will reveal itself only after I design the various algorithms and extract the various features, which is why I will develop these animation routines on the fly.

In general, all algorithms for a problem will write out a YAML file containing the schedule in the outputted run-folder. To animate a schedule and write the resulting movie to disk we just pass the name of the file containing the schedule. Since the output file-format of the schedule is identical for all algorithms of a problem, it is sufficient to have just one animation function.

Schedules will typically be animated iff there is a `animate_schedule_p` boolean flag set to `True` in the arguments of every algorithm’s function.

Here we render the Horse and Fly moving according to their assigned tours at their respective speeds, we don’t need to “coordinate” the plotting since that has already been done by the schedule itself.

A site that has been unserved is represented by a blue dot. A site that has been serviced is represented by a yellow dot.

*⟨ Animation routines for classic horsefly 65 ⟩ ≡*

```
def animateSchedule(schedule_file_name):
    import yaml
    import numpy as np
    import matplotlib.animation as animation
    from matplotlib.patches import Circle
    import matplotlib.pyplot as plt

    ⟨ Set up configurations and parameters for animation and plotting 66a ⟩
    ⟨ Parse input-output file and set up required data-structures 66b ⟩
    ⟨ Construct and store every frame of the animation in ims 67a ⟩
    ⟨ Write animation of schedule to disk and display in live window 69b ⟩
```

◇

Fragment referenced in [21a](#).

**5.16.2** In the animation, we are going to show the process of the fly delivering packages to the sites according to the pre-computed schedule. Thus the canvas must reflect the underlying euclidean space. For this, we need to set the bounding box of the Axes object to an axis-parallel unit-square whose lower-left corner is at the origin.

While displaying the animation it also helps to have a major and minor grid lightly visible to get a rough sense of distances between the sites. The settings for setting up these grids were done following the tutorial on <http://jonathansoma.com/lede/data-studio/matplotlib/adding-grid-lines-to-a-matplotlib-chart/>

We also use L<sup>A</sup>T<sub>E</sub>X for typesetting symbols and equations and the Computer Modern font for text on the plot canvas. Unfortunately, Matplotlib’s present default font for text seems to be DejaVu Sans Mono, which isn’t pretty for publications.

*( Set up configurations and parameters for animation and plotting 66a )*  $\equiv$

```
plt.rc('text', usetex=True)
plt.rc('font', family='serif')

fig, ax = plt.subplots()
ax.set_xlim([0,1])
ax.set_ylim([0,1])
ax.set_aspect('equal')

ax.set_xticks(np.arange(0, 1, 0.1))
ax.set_yticks(np.arange(0, 1, 0.1))

# Turn on the minor TICKS, which are required for the minor GRID
ax.minorticks_on()

# customize the major grid
ax.grid(which='major', linestyle='--', linewidth='0.3', color='red')

# Customize the minor grid
ax.grid(which='minor', linestyle=':', linewidth='0.3', color='black')

ax.get_xaxis().set_ticklabels([])
ax.get_yaxis().set_ticklabels([])
◇
```

Fragment referenced in 65.

**5.16.3** In this chunk, by `horse_leg` we mean the segment of a horse's tour between two successive rendezvous points with a fly while a `fly_leg` stands for the part of a fly tour when the fly leaves the horse, reaches a site, and returns back to the horse. These concepts are illustrated in the diagram below. The frames of the animation are constructed by first extracting the `horse_legs` and `fly_legs` of the horse and fly-tours and then animating the horse and fly moving along each of their respective legs.

*( Parse input-output file and set up required data-structures 66b )*  $\equiv$

```
with open(schedule_file_name, 'r') as stream:
    schedule = yaml.load(stream)

phi = float(schedule['phi'])
inithorseposn = schedule['inithorseposn']

# Get legs of the horse and fly tours
horse_tour = map(np.asarray, schedule['horse_tour'])
sites = map(np.asarray, schedule['visited_sites'])

# set important meta-data for plot
ax.set_title("Number of sites: " + str(len(sites)), fontsize=25)
ax.set_xlabel(r"$\varphi$ = " + str(phi), fontsize=20)

xhs = [ horse_tour[i][0] for i in range(len(horse_tour))]
yhs = [ horse_tour[i][1] for i in range(len(horse_tour))]
xfs, yfs = [xhs[0]], [yhs[0]]
for site, pt in zip (sites, horse_tour[1:]):
    xfs.extend([site[0], pt[0]])
    yfs.extend([site[1], pt[1]])
fly_tour = map(np.asarray, zip(xfs, yfs))

horse_legs = zip(horse_tour, horse_tour[1:])
fly_legs = zip(fly_tour, fly_tour[1:], fly_tour[2:]) [0::2]

assert(len(horse_legs) == len(fly_legs))
```

◇

Fragment referenced in 65.

**5.16.4** The `ims` array stores each frame of the animation. Every frame consists of various “artist” objects <sup>2</sup> (e.g. circles and segments) which change dynamically as the positions of the horse and flies change.

```

⟨ Construct and store every frame of the animation in ims 67a ⟩ ≡
    ims = []
    for horse_leg, fly_leg, leg_idx in zip(horse_legs, \
                                           fly_legs, \
                                           range(len(horse_legs))):
        debug(Fore.YELLOW + "Animating leg: " + str(leg_idx) + Style.RESET_ALL)

        ⟨ Define function to place points along a leg 69a ⟩

        horse_posns = discretize_leg(horse_leg)
        fly_posns    = discretize_leg(fly_leg)
        assert(len(horse_posns) == len(fly_posns))

        hxs = [xhs[i] for i in range(0,leg_idx+1) ]
        hys = [yhs[i] for i in range(0,leg_idx+1) ]

        fxs , fys = [hxs[0]], [hys[0]]
        for site, pt in zip (sites,(zip(hxs,hys))[1:]):
            fxs.extend([site[0], pt[0]])
            fys.extend([site[1], pt[1]])

        number_of_sites_served = leg_idx
        for horse_posn, fly_posn, subleg_idx in zip(horse_posns, \
                                                    fly_posns, \
                                                    range(len(horse_posns))):

            ⟨ Render frame and append it to ims 67b ⟩





```

◇

Fragment referenced in 65.

Defines: `number_of_sites_served` 67b.

**5.16.5** While rendering the horse and fly tours we need to keep track of the horse and fly-legs and sites that have been serviced so far.

- The path covered by the horse from the initial point till its current position is colored red 
- The path covered by the fly from the initial point till its current position is colored green 
- Unserviced sites are marked blue .
- When sites get serviced, they are marked yellow .

While iterating through all the sublegs of the current fly-leg, we need to keep track if the fly has serviced the site or not. That is the job of the `if subleg_idx==9` block in the code-fragment below. The magic-number “9” is related to the 10 and 19 constants from the `discretize_leg` function defined later in [subsection 5.16.6](#).

```

⟨ Render frame and append it to ims 67b ⟩ ≡

    debug(Fore.RED + "Rendering subleg " + str(subleg_idx) + Style.RESET_ALL)
    hxs1 = hxs + [horse_posn[0]]
    hys1 = hys + [horse_posn[1]]

    fxs1 = fxs + [fly_posn[0]]

```

---

<sup>2</sup>This is Matplotlib terminology

```

fys1 = fys + [fly_posn[1]]

# There is a midway update for new site check is site
# has been serviced. If so, update fxs and fys
if subleg_idx == 9:
    fxs.append(sites[leg_idx][0])
    fys.append(sites[leg_idx][1])
    number_of_sites_serviced += 1

horseline, = ax.plot(hxs1,hys1,'ro-', linewidth=5.0, markersize=6, alpha=1.00)
flyline,   = ax.plot(fxs1,fys1,'go-', linewidth=1.0, markersize=3)

objs = [flyline,horseline]

# Mark serviced and unserviced sites with different colors.
# Use https://htmlcolorcodes.com/ for choosing good colors along with their hex-codes.

for site, j in zip(sites, range(len(sites))):
    if j < number_of_sites_serviced:      # site has been serviced
        sitecolor = '#DBC657' # yellowish
    else:                                  # site has not been serviced
        sitecolor = 'blue'

    circle = Circle((site[0], site[1]), 0.02, \
                    facecolor = sitecolor, \
                    edgecolor = 'black', \
                    linewidth=1.4)
    sitepatch = ax.add_patch(circle)
    objs.append(sitepatch)

debug(Fore.CYAN + "Appending to ims " + Style.RESET_ALL)
ims.append(objs[:-1])
◇

```

Fragment referenced in [67a](#).

Uses: `number_of_sites_serviced` [67a](#).

**5.16.6** The numbers 19 and 10 to discretize the horse and fly legs have been arbitrarily chosen. These seem to work well for giving smooth real-time animation. However, you will notice both the horse and fly seem to speed up or sometimes slow down.

That's why ideally, these discretization params should actually depend on the length of the legs, and the speeds of the horse and fly. However, just using constants is good enough for now. I just want a working animation.

A leg consists of either one segment (for horse) or two segments(for fly).

For a horse-leg, we must make sure that the leg-end points are part of the discretization of the leg.

For a fly-leg, we must ensure that the leg-end points and the site being serviced during the leg are in its discretization. Note that in this case, since each of the two segments are being discretized with `np.linspace`, we need to make sure that the site corresponding to the fly-leg is not counted twice, which explains the odd-looking `subleg_pts.extend(tmp[:-1])` statement in the code-fragment below.

⟨ Define function to place points along a leg 69a ⟩ ≡

```
def discretize_leg(pts):
    subleg_pts = []
    numpts = len(pts)

    if numpts == 2:
        k = 19 # horse
    elif numpts == 3:
        k = 10 # fly

    for p,q in zip(pts, pts[1:]):
        tmp = []
        for t in np.linspace(0,1,k):
            tmp.append( (1-t)*p + t*q )
        subleg_pts.extend(tmp[:-1])

    subleg_pts.append(pts[-1])
    return subleg_pts
```

◇

Fragment referenced in 67a.

## 5.16.7

⟨ Write animation of schedule to disk and display in live window 69b ⟩ ≡

```
from colorama import Back

debug(Fore.BLACK + Back.WHITE + "\nStarted constructing ani object"+ Style.RESET_ALL)
ani = animation.ArtistAnimation(fig, ims, interval=50, blit=True, repeat_delay=1000)
debug(Fore.BLACK + Back.WHITE + "\nFinished constructing ani object"+ Style.RESET_ALL)

#plt.show() # For displaying the animation in a live window.

debug(Fore.MAGENTA + "\nStarted writing animation to disk"+ Style.RESET_ALL)
ani.save(schedule_file_name+'.avi', dpi=150)
debug(Fore.MAGENTA + "\nFinished writing animation to disk"+ Style.RESET_ALL)
```

◇

Fragment referenced in 65.

# Chapter Index of Fragments

⟨ Algorithms for classic horsefly 28, 29, 32a, 34, 39a, 54, 57, 59 ⟩ Referenced in 21a.  
 ⟨ Animation routines for classic horsefly 65 ⟩ Referenced in 21a.  
 ⟨ Clear canvas and states of all objects 25a ⟩ Referenced in 22b.  
 ⟨ Compute the length of the tour that currently services the visited sites 48a ⟩ Referenced in 47b.  
 ⟨ Construct and store every frame of the animation in ims 67a ⟩ Referenced in 65.  
 ⟨ Construct the fly-tour from the information about horse tour and sites 62d ⟩ Referenced in 62a.  
 ⟨ Create singleton graph, with node at inithorseposn 52a ⟩ Referenced in 51.  
 ⟨ Define auxiliary helper functions 45c, 46ab ⟩ Referenced in 39a.  
 ⟨ Define function to place points along a leg 69a ⟩ Referenced in 67a.  
 ⟨ Define function greedy 31a ⟩ Referenced in 29.  
 ⟨ Define function next\_rendezvous\_point\_for\_horse\_and\_fly 30b ⟩ Referenced in 29.  
 ⟨ Define key-press handler 22b ⟩ Referenced in 21a.  
 ⟨ Define various insertion policy classes 47a ⟩ Referenced in 39a.  
 ⟨ Draw horse and fly-tours 43a ⟩ Referenced in 41b.  
 ⟨ Draw unvisited sites as filled blue circles 43c ⟩ Referenced in 41b.

< Extract  $x$  and  $y$  coordinates of the points on the horse, fly tours, visited and unvisited sites [42a](#) > Referenced in [41b](#).  
 < Find the node with the closest site, and generate the next node and edge for the  $\varphi$ -MST [53](#) > Referenced in [51](#).  
 < Find the unvisited site which on insertion increases tour-length by the least amount [48d](#) > Referenced in [47b](#).  
 < For each node, find the closest site [52b](#) > Referenced in [51](#).  
 < Generate a bunch of uniform or non-uniform random points on the canvas [24](#) > Referenced in [22b](#).  
 < Get  $x$  and  $y$  coordinates of the endpoints of segments on the horse-tour [62b](#) > Referenced in [62a](#).  
 < Get  $x$  and  $y$  coordinates of the sites [62c](#) > Referenced in [62a](#).  
 < Give metainformation about current picture as headers and footers [43d](#) > Referenced in [41b](#).  
 < If `self.sites[u]` is chosen for insertion, find best insertion position and update `delta_increase_least_table` [48c](#) > Referenced in [47b](#).  
 < Local data-structures for classic horsefly [26a](#) > Referenced in [21a](#).  
 < Local utility functions for classic horsefly [60ab](#) > Referenced in [21a](#).  
 < Lower bounds for classic horsefly [51](#) > Referenced in [21a](#).  
 < Make an animation of algorithm states, if `write_algo_states_to_disk_p == True` [44b](#) > Referenced in [39a](#).  
 < Make an animation of the schedule computed by `algo_greedy`, if `animate_schedule_p == True` [31c](#) > Referenced in [29](#).  
 < Make an animation of the schedule, if `animate_schedule_p == True` [45a](#) > Referenced in [39a](#).  
 < Mark initial position of horse and fly boldly on canvas [42b](#) > Referenced in [41b](#).  
 < Methods for `HorseFlyInput` [26bcd](#), [27](#) > Referenced in [26a](#).  
 < Methods for `PolicyBestInsertionNaive` [47b](#) > Referenced in [47a](#).  
 < Parse input-output file and set up required data-structures [66b](#) > Referenced in [65](#).  
 < Place numbered markers on visited sites to mark the order of visitation explicitly [43b](#) > Referenced in [41b](#).  
 < Plot everything [64a](#) > Referenced in [62a](#).  
 < Plotting routines for classic horsefly [62a](#), [64b](#) > Referenced in [21a](#).  
 < Print information about the fly tour [63a](#) > Referenced in [62a](#).  
 < Print information about the horse tour [62e](#) > Referenced in [62a](#).  
 < Print meta-data about the algorithm run [63b](#) > Referenced in [62a](#).  
 < Relevant imports for classic horsefly [21b](#) > Referenced in [21a](#).  
 < Render current algorithm state to image file [41b](#) > Referenced in [41a](#).  
 < Render frame and append it to `ims` [67b](#) > Referenced in [67a](#).  
 < Return horsefly tour, along with additional information [45b](#) > Referenced in [39a](#).  
 < Set insertion policy class for current run [40a](#) > Referenced in [39a](#).  
 < Set log, algo-state and input-output files config [39b](#) > Referenced in [39a](#).  
 < Set log, algo-state and input-output files config for `algo_greedy` [30a](#) > Referenced in [29](#).  
 < Set up configurations and parameters for animation and plotting [66a](#) > Referenced in [65](#).  
 < Set up interactive canvas [25b](#) > Referenced in [21a](#).  
 < Set up logging information relevant to this module [22a](#) > Referenced in [21a](#).  
 < Set up plotting area and canvas, fig, ax, and other configs [41c](#) > Referenced in [41b](#).  
 < Set up tracking variables local to this iteration [48b](#) > Referenced in [47b](#).  
 < Start entering input from the command-line [23](#) > Referenced in [22b](#).  
 < Update states for `PolicyBestInsertionNaive` [49](#) > Referenced in [47b](#).  
 < Use insertion policy to find the cheapest site to insert into current tour [40b](#) > Referenced in [39a](#).  
 < Useful functions for `algo_exact_given_specific_ordering` [32b](#), [33](#) > Referenced in [32a](#).  
 < Write algorithms current state to file [41a](#) > Referenced in [39a](#).  
 < Write animation of schedule to disk and display in live window [69b](#) > Referenced in [65](#).  
 < Write image file [43e](#) > Referenced in [41b](#).  
 < Write input and output of `algo_greedy` to file [31b](#) > Referenced in [29](#).  
 < Write input and output to file [44a](#) > Referenced in [39a](#).

## Chapter Index of Identifiers

`algo_exact_given_specific_ordering`: [23](#), [28](#), [30b](#), [32a](#).  
`algo_greedy_incremental_insertion`,: [23](#), [39a](#).  
`clearAllStates`: [24](#), [25a](#), [26b](#).  
`computeStructure`: [23](#), [26d](#).  
`compute_collinear_horseflies_tour`: [46b](#), [49](#).  
`compute_collinear_horseflies_tour_length`: [46a](#), [48ac](#).  
`current_tour_length`: [48a](#), [48c](#).

---

delta\_increase\_least: [48b](#), [48c](#).  
delta\_increase\_least\_table: [47b](#), [48cd](#).  
draw\_phi\_mst: [23](#), [64b](#).  
generate\_constraints: [32a](#), [33](#), [57](#).  
getTour: [23](#), [26c](#).  
greedy: [29](#), [30a](#), [31a](#), [39b](#).  
HorseFlyInput: [25b](#), [26a](#).  
ibest,: [48b](#), [48c](#).  
io\_file\_name,: [31b](#), [39b](#), [44a](#).  
ith\_leg\_constraint: [32b](#), [33](#).  
logger: [22a](#), [30a](#), [39b](#).  
number\_of\_sites\_serviced: [67a](#), [67b](#).  
plotTour: [23](#), [62a](#).  
self.horse\_tour: [47a](#), [49](#).  
self.inithorseposn,: [26cd](#), [47a](#), [48ac](#), [49](#).  
self.sites,: [26cd](#), [47a](#).  
self.visited\_sites,: [47a](#), [48a](#), [49](#).  
single\_site\_solution: [45c](#), [46ab](#), [53](#).  
tour\_length: [28](#), [32a](#), [54](#), [57](#), [60a](#), [63b](#), [64a](#).  
tour\_length\_with\_waiting\_time\_included: [32a](#), [34](#), [45b](#), [60b](#), [63b](#).  
unmarked\_sites\_idx: [51](#), [52b](#).  
wrapperkeyPressHandler: [22b](#), [25b](#).  
write\_algo\_states\_to\_disk\_p: [29](#), [39a](#), [41ab](#), [44b](#).

## Chapter 6

# Reverse Horsefly



## Chapter 7

# One Horse, Multiple Flies

# Appendices

# Appendix A

## Index of Files

"../main.py" Defined by [12a](#).  
"../src/lib/problem\_classic\_horsefly.py" Defined by [21a](#).  
"../src/lib/utils\_algo.py" Defined by [17b](#), [18abcd](#), [19abc](#).  
"../src/lib/utils\_graphics.py" Defined by [14](#), [15abc](#), [17a](#).

## Appendix B

### Man-page for **main.py**

# Bucketlist of TODOS

■ Add an item containing the interface files. Do this for the Haskell files that you will ultimately add in later. . . . .	11
■ Remove the previous red patches, which contain the old position of the horse and fly. Doing this is slightly painful, hence keeping it for later. . . . .	15
Figure: Testing a long text string . . . . .	50