Experimental Analyses of Heuristics for Horsefly-type Problems

Gaurish Telang

Contents

Part I

Overview

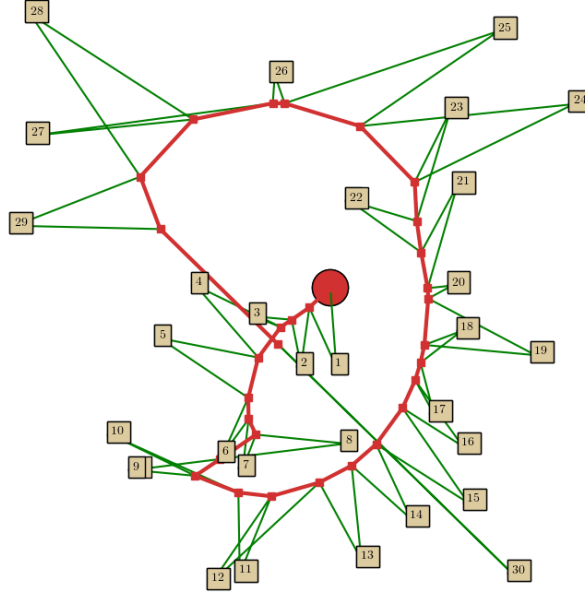# Chapter 1

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 φ. [1] [2].

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. [3] 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.

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. [4] Both the initial position of the truck and the drone are given. The sites to be serviced

---

[1]The speed of the truck is always assumed to be 1 in any of the problem variations we will be considering in this report.

[2]φ is also called the "speed ratio".

[3]The drone is assumed to be able to carry at most one package at a time

[4]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.

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. [5]

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. [6] 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° vision. Both the truck and drone can move, but the drone can move at most euclidean distance [7] 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 coordinattion with the truck) to patrol the entire polygon.

---

[5] 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 realisting model, we would need to take into account this "waiting time" too.

[6] although abstractly, the Watchman route problem can be viewed as a kind of TSP

[7] 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[1], MacOS, and almost any GNU/Linux distribution. For a quick introduction to containerization, watch the first two minutes of https://youtu.be/_dfLOzuIg2o

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.) 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 instrutions watch

   GNU/Linux https://youtu.be/KCckWweNSrM

   Windows https://youtu.be/ymlWt1MqURY

   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` [2]

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. [3]

   If you are running Windows • Follow every instruction in https://dev.to/darksmile92/run-gui-app-in-linux-Docker-container-on-windows-host-4kde. [4] Make sure you can run 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. [5]

   - Open up a Windows Powershell (possibly as administrator)

---

[1]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

[2]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.

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

[4]This step is necessary displaying the Matplotlib canvas as we do in the horseflies project for interactive testing of algorithms.

[5]you might need administrator privileges to perform this step, as pointed out by the article.

– `set-variable -name DISPLAY -value <your-ip-address>:0.0` [6]

– `docker run -ti --rm -e DISPLAY=$DISPLAY -v <location-of-horseflies-folder>:/horseflies_mnt  gtelang/u`

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 [7],
`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 φ), run `python main.py --<problem`
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 --`

If you want to delete the Ubuntu image and any associated containers run the command [8]

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

That's it! Happy horseflying!

---

[6]You can find your ip-address by the output of the `ipconfig` command in the Powershell
[7]Allowing, of course, for differences between your machine's CPU and mine when it comes to reporting absolute running time
[8]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

```
..
|-- scratch
|   |-- thoughts_on_the_airplane
|   |-- todo
|   `-- trash.org
|-- src
|   |-- expts
|   |   `-- expt_decrease_in_tour_length_with_larger_number_of_drones.py
|   |-- lib
|   |   |-- html
|   |   |   |-- api-objects.txt
|   |   |   |-- class-tree.html
|   |   |   |-- epydoc.css
|   |   |   |-- epydoc.js
|   |   |   |-- frames.html
|   |   |   |-- help.html
|   |   |   |-- identifier-index.html
|   |   |   |-- index.html
|   |   |   |-- module-tree.html
|   |   |   |-- problem_classic_horsefly.HorseFlyInput-class.html
|   |   |   |-- problem_classic_horsefly-module.html
|   |   |   |-- problem_classic_horsefly.PolicyBestInsertionNaive-class.html
|   |   |   |-- problem_classic_horsefly-pysrc.html
|   |   |   |-- redirect.html
|   |   |   |-- toc-everything.html
|   |   |   |-- toc.html
|   |   |   |-- toc-problem_classic_horsefly-module.html
|   |   |   |-- toc-utils_algo-module.html
|   |   |   |-- toc-utils_graphics-module.html
|   |   |   |-- utils_algo-module.html
|   |   |   |-- utils_algo-pysrc.html
|   |   |   |-- utils_graphics-module.html
|   |   |   `-- utils_graphics-pysrc.html
|   |   |-- problem_classic_horsefly.py
|   |   |-- problem_one_horse_multiple_flies.py
|   |   |-- scratchpad
|   |   |-- 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
|   |   |-- algo-greedy-earliest-capture.web
|   |   |-- experiment-fall-of-tour-length-with-increasing-drones.web
|   |   `-- 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
`-- weave-tangle.sh

14 directories, 62 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 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 supporing 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.)

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"` 12≡

```
    ⟨ Turn off Matplotlibs irritating DEBUG messages 13a ⟩
    ⟨ Import problem module files 13b ⟩


    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* 13a⟩ ≡
```
import logging
mpl_logger = logging.getLogger('matplotlib')
mpl_logger.setLevel(logging.WARNING)
```
    ◇
Fragment referenced in 12.

⟨ *Import problem module files* 13b⟩ ≡

```
import sys
sys.path.append('src/lib')
import problem_classic_horsefly as chf
import problem_one_horse_multiple_flies as ohmf
```
    ◇
Fragment referenced in 12.

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. intial 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, 16d.

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≡

```python
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, 16d.

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 behavioue can be achieved by sub-classing.

"../src/lib/utils_graphics.py" 15b≡

```python
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, 16d.

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.

Remove the
patches, w
the old po
horse and
is slightly p
keeping it

"../src/lib/utils_graphics.py" 15c≡

```python
def wrapperEnterRunPoints(fig, ax, run):
    def _enterPoints(event):
        if event.name      == 'button_press_event'      and \
           (event.button   == 1 or event.button == 3)      and \
            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, 16d.

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])/100.0

    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 generate a specified number of visually distinct colors, especially when dealing with multiple flies.

"../src/lib/utils_graphics.py" 16d≡

```
    # Borrowed from https://stackoverflow.com/a/9701141
    import numpy as np
    import colorsys

    def get_colors(num_colors, lightness=0.2):
        colors=[]
        for i in np.arange(60., 360., 300. / num_colors):
            hue         = i/360.0
```

```
            saturation = 0.95
            colors.append(colorsys.hls_to_rgb(hue, lightness, saturation))
        return colors
    ◇
```

File defined by 14, 15abc, 16d.

Algorithmic Utilities

4.2.1   Given a list of points $[p_0, p_1, p_2, ... p_{n-1}]$. the following function returns, $[p_1 - p_0, p_2 - p_1, ..., p_{n-1} - p_{n-2}]$ 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" 17a≡

```
    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 17ab, 18abcdef, 19c.

4.2.2   Given a polygonal chain in the form of successive points $[p_0, p_1, p_2, ... p_{n-1}]$, 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" 17b≡

```
    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 17ab, 18abcdef, 19c.

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. $[x_0, x_1, x_2, ... x_{2n}] \rightarrow [[x_0, x_1], [x_2, x_3], , .. [x_{2n-1}, x_{2n}]]$

```
"../src/lib/utils_algo.py" 18a≡
    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 pointifyed')
    ◇
```
File defined by 17ab, 18abcdef, 19c.

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

```
"../src/lib/utils_algo.py" 18b≡
    def flatten_list_of_lists(l):
        return [item for sublist in l for item in sublist]
    ◇
```
File defined by 17ab, 18abcdef, 19c.

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" 18c≡
    def print_list(xs):
        for x in xs:
            print x
    ◇
```
File defined by 17ab, 18abcdef, 19c.

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" 18d≡
    def partial_sums( xs ):
        psum = 0
        acc = []
        for x in xs:
            psum = psum+x
            acc.append( psum )
        return acc
    ◇
```
File defined by 17ab, 18abcdef, 19c.

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

```
"../src/lib/utils_algo.py" 18e≡
    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 17ab, 18abcdef, 19c.

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" 18f≡
    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 19a⟩

    ⟨Whatever number of points are left to be generated, generate them uniformly inside the unit-square 19b⟩

    return points
```
    ◇

File defined by 17ab, 18abcdef, 19c.
Defines: cluster_size 19ab, scale, 19a.

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* 19a⟩ ≡

```
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 18f.
Uses: cluster_size 18f, scale, 18f.

4.2.9

⟨*Whatever number of points are left to be generated, generate them uniformly inside the unit-square* 19b⟩ ≡

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

Fragment referenced in 18f.
Uses: cluster_size 18f.

4.2.10   This is the main serialization function to write out data to YAML files for later analyses of algorithm runs.

"../src/lib/utils_algo.py" 19c≡

```
def write_to_yaml_file(data, dir_name, file_name):
    import yaml
    with open(dir_name + '/' + file_name, 'w') as outfile:
      yaml.dump( data, outfile, default_flow_style = False)
```
    ◇

File defined by 17ab, 18abcdef, 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"` 20a≡

⟨*Relevant imports for classic horsefly* 20b⟩
⟨*Set up logging information relevant to this module* 21a⟩
```
def run_handler():
```
    ⟨*Define key-press handler* 21b⟩
    ⟨*Set up interactive canvas* 24b⟩

⟨*Local data-structures for classic horsefly* 25a⟩
⟨*Local utility functions for classic horsefly* 57a, … ⟩
⟨*Algorithms for classic horsefly* 27, … ⟩
⟨*Lower bounds for classic horsefly* 49a⟩
⟨*Plotting routines for classic horsefly* 58a, … ⟩
⟨*Animation routines for classic horsefly* 61⟩
◇

Module Details

5.2.1

⟨*Relevant imports for classic horsefly* 20b⟩ ≡

```python
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 20a.

5.2.2 The logger variable becomes 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* 21a⟩ ≡

```
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 20a.
Uses: logger 38b.

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 φ, 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* 21b⟩ ≡

```
# 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 22⟩
            elif event.key in ['n', 'N', 'u', 'U']:
                    ⟨Generate a bunch of uniform or non-uniform random points on the canvas 23⟩
            elif event.key in ['c', 'C']:
                    ⟨Clear canvas and states of all objects 24a⟩
        return _keyPressHandler
◇
```

Fragment referenced in 20a.
Defines: wrapperkeyPressHandler 24b.

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* 22 ⟩ ≡

```
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 21b.
Uses: algo_exact_given_specific_ordering 31a, algo_greedy_incremental_insertion, 38a, computeStructure 25d, draw_phi_mst 60b, getTour 25c, plotTour 58a.

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 no 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 incase 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* 23⟩ ≡

```
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 21b.
Uses: clearAllStates 25b.

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* 24a ⟩ ≡

```
run.clearAllStates()
ax.cla()

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

fig.texts = []
fig.canvas.draw()
◇
```

Fragment referenced in 21b.
Uses: clearAllStates 25b.

5.2.7

⟨ *Set up interactive canvas* 24b ⟩ ≡

```
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 20a.
Uses: HorseFlyInput 25a, wrapperkeyPressHandler 21b.

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* 25a⟩ ≡

```
class HorseFlyInput:
      def __init__(self, sites=[], inithorseposn=()):
            self.sites       = sites
            self.inithorseposn = inithorseposn

            ⟨Methods for HorseFlyInput 25b, … ⟩
   ◇
```
Fragment referenced in 20a.
Defines: HorseFlyInput 24b.

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

⟨ *Methods for* HorseFlyInput 25b⟩ ≡

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


   ◇
```
Fragment defined by 25bcd, 26.
Fragment referenced in 25a.
Defines: clearAllStates 23, 24a.

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 25c⟩ ≡

```
      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 25bcd, 26.
Fragment referenced in 25a.
Defines: getTour 22.
Uses: self.inithorseposn, 45b, self.sites, 45b.

5.3.4

⟨ *Methods for* HorseFlyInput 25d⟩ ≡

```
      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 25bcd, 26.
Fragment referenced in 25a.

Defines: computeStructure 22.
Uses: self.inithorseposn, 45b, self.sites, 45b.

5.3.5 This chunk prints a customized representation of the HorseFlyInput class

⟨ *Methods for* HorseFlyInput 26 ⟩ ≡

```python
    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 25bcd, 26.
Fragment referenced in 25a.

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* 27 ⟩ ≡

```
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 27, 28, 31a, 32b, 38a, 51, 54, 56.
Fragment referenced in 20a.
Uses: algo_exact_given_specific_ordering 31a, tour_length 57a.

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.
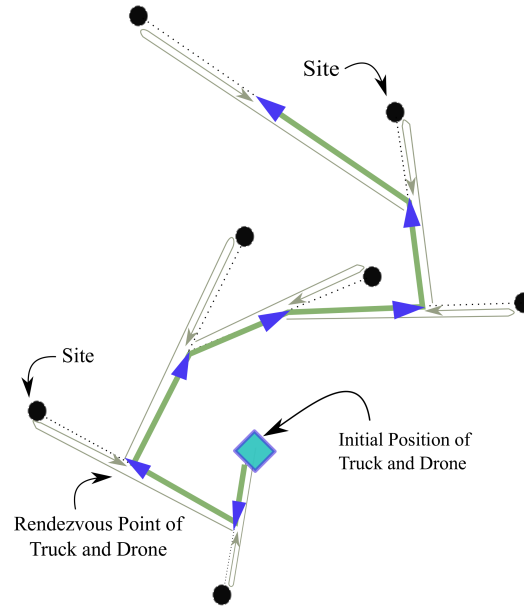
Figure 5.1: The Collinear Horsefly Problem

We can show that an optimal (unrestricted) horsfly solution can be converted to a collinear-horsefly 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 horsefly, 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 horsefly* 28⟩ ≡

```
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 29a⟩
    ⟨Define function next_rendezvous_point_for_horse_and_fly 29b⟩
    ⟨Define function greedy 30a⟩

    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 30b⟩
    ⟨Make an animation of the schedule computed by algo_greedy, if animate_schedule_p == True 30c⟩
    return answer
◇
```

Fragment defined by 27, 28, 31a, 32b, 38a, 51, 54, 56.
Fragment referenced in 20a.

Uses: greedy 30a, write_algo_states_to_disk_p 38a.

⟨ *Set log, algo-state and input-output files config for* algo_greedy 29a⟩ ≡

```
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 28.
Uses: greedy 30a, logger 38b.

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 foir a single site.

⟨ *Define function* next_rendezvous_point_for_horse_and_fly 29b⟩ ≡

```
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 28.
Uses: algo_exact_given_specific_ordering 31a.

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 nearst 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 30a ⟩ ≡

```
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 28.
Defines: greedy 28, 29a, 38b.

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* 30b ⟩ ≡

```
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 28.
Uses: io_file_name, 38b.

5.5.7

⟨ *Make an animation of the schedule computed by* algo_greedy, *if* animate_schedule_p == True 30c ⟩ ≡

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

Fragment referenced in 28.

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 calcualting 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 studies 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* 31a⟩ ≡

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

    ⟨ Useful functions for algo_exact_given_specific_ordering 31b, ... ⟩

    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 27, 28, 31a, 32b, 38a, 51, 54, 56.
Fragment referenced in 20a.
Defines: algo_exact_given_specific_ordering 22, 27, 29b.
Uses: generate_constraints 32a, tour_length 57a, tour_length_with_waiting_time_included 57b.

5.5.9 For the ith 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 31b⟩ ≡

```python
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 31b, 32a.
Fragment referenced in 31a.
Defines: ith_leg_constraint 32a.

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 32a⟩ ≡

```
    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 31b, 32a.
Fragment referenced in 31a.
Defines: generate_constraints 31a, 54.
Uses: ith_leg_constraint 31b.

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 >= 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* 32b⟩ ≡

```python
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 each
  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)
```

```python
        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, -1.0])
row += 1; task.putconbound(row, mosek.boundkey.lo, alpha  , np.inf) ; task.putarow(row, [p(0), h(0)],[1.0,  1.0])

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

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

row += 1; task.putconbound(row, mosek.boundkey.up, -np.inf, s[1] ) ; task.putarow(row, [p(1), f(1)],[1.0, -1.0])
row += 1; task.putconbound(row, mosek.boundkey.lo,  s[1]  , np.inf) ; task.putarow(row, [p(1), f(1)],[1.0,  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(2*i-2)],[1
    row+=1 ;  task.putconbound(row, mosek.boundkey.lo,  s[2*i]  , np.inf) ; task.putarow(row, [b(2*i),   p(2*i-2)],[1
    row+=1 ;  task.putconbound(row, mosek.boundkey.lo, -s[2*i+1], np.inf) ; task.putarow(row, [b(2*i+1), p(2*i-1)],[
    row+=1 ;  task.putconbound(row, mosek.boundkey.lo,  s[2*i+1], np.inf) ; task.putarow(row, [b(2*i+1), p(2*i-1)],[

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

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

◇

Fragment defined by 27, 28, 31a, 32b, 38a, 51, 54, 56.
Fragment referenced in 20a.
Uses: tour_length_with_waiting_time_included 57b.

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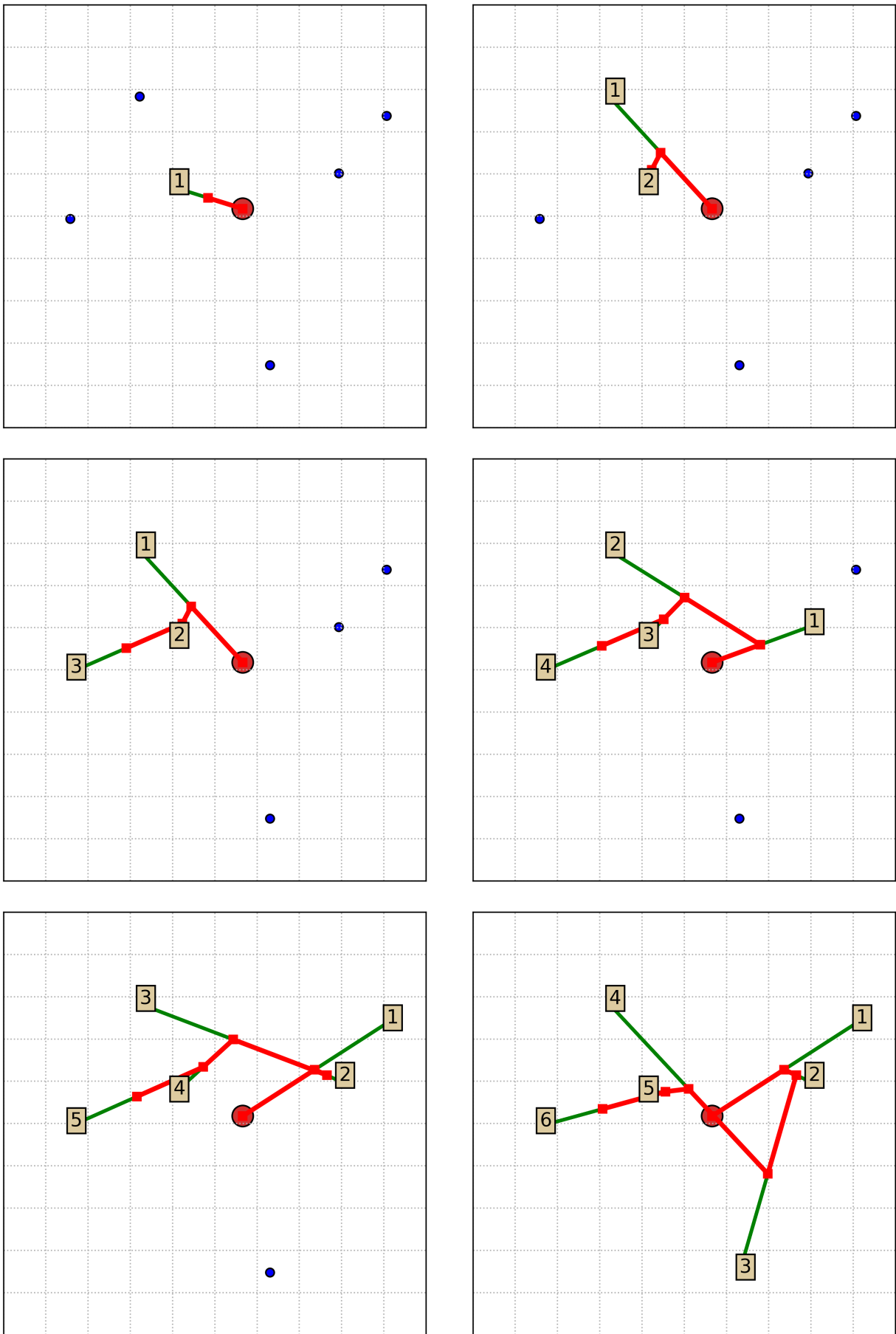
Figure 5.2: Greedy incremental insertion for collinear horseflies. $\varphi = 3.0$. Notice that the ordering of the visited sites keep changing based on where we decide to insert an unvisited site.

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* 38a⟩ ≡

```
⟨ Define auxiliary helper functions 44a, … ⟩
⟨ Define various insertion policy classes 45b⟩
def algo_greedy_incremental_insertion(sites, inithorseposn, phi,
                                      insertion_policy_name    = "naive",
                                      write_algo_states_to_disk_p = False   ,
                                      animate_schedule_p        = True    ,
                                      post_optimizer            = None):
    ⟨ Set log, algo-state and input-output files config 38b⟩
    ⟨ Set insertion policy class for current run 39a⟩

    while insertion_policy.unvisited_sites_idxs:
        ⟨ Use insertion policy to find the cheapest site to insert into current tour 39b⟩
        ⟨ Write algorithms current state to file 40a⟩

    ⟨ Write input and output to file 43a⟩
    ⟨ Make an animation of the schedule, if animate_schedule_p == True 43c⟩
    #sys.exit()
    ⟨ Make an animation of algorithm states, if write_algo_states_to_disk_p == True 43b⟩
    ⟨ Return horsefly tour, along with additional information 43d⟩
```
◇

Fragment defined by 27, 28, 31a, 32b, 38a, 51, 54, 56.
Fragment referenced in 20a.
Defines: algo_greedy_incremental_insertion, 22, write_algo_states_to_disk_p 28, 40ab, 43b.

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 containe 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* 38b⟩ ≡

```
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 = 1
    ◇
```

Fragment referenced in 38a.
Defines: io_file_name, 30b, 43a, logger 21a, 29a.
Uses: greedy 30a.

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* 39a⟩ ≡

```
    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 38a.

5.6.6 Note that while defining the body of the algorithm, we treat the insertion policy (whose name has already been passed as an 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* 39b⟩ ≡

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

Fragment referenced in 38a.

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 serialized 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* 40a⟩ ≡

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

    data = {'insertion_policy_name' : insertion_policy_name                 ,
            'unvisited_sites'       : [insertion_policy.sites[u] \
                                        for u in insertion_policy.unvisited_sites_idxs],
            '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 40b⟩

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

Fragment referenced in 38a.
Uses: write_algo_states_to_disk_p 38a.

⟨*Render current algorithm state to image file* 40b⟩ ≡

```
import utils_algo
if write_algo_states_to_disk_p:
    ⟨Set up plotting area and canvas, fig, ax, and other configs 40c⟩
    ⟨Extract x and y coordinates of the points on the horse, fly tours, visited and unvisited sites 41a⟩
    ⟨Mark initial position of horse and fly boldly on canvas 41b⟩
    ⟨Place numbered markers on visited sites to mark the order of visitation explicitly 41d⟩
    ⟨Draw horse and fly-tours 41c⟩
    ⟨Draw unvisited sites as filled blue circles 42a⟩
    ⟨Give metainformation about current picture as headers and footers 42b⟩
    ⟨Write image file 42c⟩
```
◇

Fragment referenced in 40a.
Uses: write_algo_states_to_disk_p 38a.

5.6.9

⟨*Set up plotting area and canvas, fig, ax, and other configs* 40c⟩ ≡

```
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 40b.

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* 41a⟩ ≡

```
    # 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 40b.

5.6.11

⟨ *Mark initial position of horse and fly boldly on canvas* 41b⟩ ≡

```
    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 40b.

⟨ *Draw horse and fly-tours* 41c⟩ ≡

```
    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 40b.

⟨ *Place numbered markers on visited sites to mark the order of visitation explicitly* 41d⟩ ≡

```
    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 40b.

⟨ *Draw unvisited sites as filled blue circles* 42a⟩ ≡

```
    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 40b.

5.6.12

⟨ *Give metainformation about current picture as headers and footers* 42b⟩ ≡

```
    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 40b.

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 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* 42c⟩ ≡

```
    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 40b.

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* 43a⟩ ≡

```
    # 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 38a.
Uses: io_file_name, 38b.

5.6.14 If algorithm states have been rendered to files in the run-folder, we stitch them together using ffmpeg and make an .avi animation of the changing states of the algorithms. The .avi 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 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 43b⟩ ≡

```
    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 38a.
Uses: write_algo_states_to_disk_p 38a.

5.6.15 This chunks 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 43c⟩ ≡

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

Fragment referenced in 38a.

5.6.16

⟨ *Return horsefly tour, along with additional information* 43d⟩ ≡

```
    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 38a.
Uses: tour_length_with_waiting_time_included 57b.

5.6.17 We now define some of the functions that were referred to in the above chunks. Given the intial 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* 44a ⟩ ≡

```
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 44ab, 45a.
Fragment referenced in 38a.
Defines: single_site_solution 44b, 45a, 50b.

With that the tour length functions for collinear horseflies can be implemented as an elementary instance of the fold pattern of functional programming. [1]

⟨ *Define auxiliary helper functions* 44b ⟩ ≡

```
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 44ab, 45a.
Fragment referenced in 38a.
Defines: compute_collinear_horseflies_tour_length 46bd.
Uses: single_site_solution 44a.

---

[1] 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!

⟨*Define auxiliary helper functions* 45a⟩ ≡

```
    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 44ab, 45a.
Fragment referenced in 38a.
Defines: compute_collinear_horseflies_tour 47b.
Uses: single_site_solution 44a.

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.

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* 45b⟩ ≡

```
    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_idxs = range(len(sites)) # This indexes into self.sites
            self.horse_tour           = [self.inithorseposn]
```

⟨*Methods for* PolicyBestInsertionNaive 46a⟩

◇

Fragment referenced in 38a.
Defines: self.horse_tour 47b, self.inithorseposn, 25cd, 46bd, 47b, self.sites, 25cd, self.visited_sites, 46b, 47b.

5.7.3

⟨*Methods for* PolicyBestInsertionNaive 46a⟩ ≡

```
    def insert_another_unvisited_site(self):
        ⟨Compute the length of the tour that currently services the visited sites 46b⟩
        delta_increase_least_table = [] # tracking variable updated in for loop below

        for u in self.unvisited_sites_idxs:
            ⟨Set up tracking variables local to this iteration 46c⟩
            ⟨If self.sites[u] is chosen for insertion, find best insertion position and update delta_increase_least_table 46d⟩

        ⟨Find the unvisited site which on insertion increases tour-length by the least amount 47a⟩
        ⟨Update states for PolicyBestInsertionNaive 47b⟩
```

◇

Fragment referenced in 45b.
Defines: delta_increase_least_table 46d, 47a.

5.7.4

⟨*Compute the length of the tour that currently services the visited sites* 46b⟩ ≡

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

◇

Fragment referenced in 46a.
Defines: current_tour_length 46d.
Uses: compute_collinear_horseflies_tour_length 44b, self.inithorseposn, 45b, self.visited_sites, 45b.

5.7.5

⟨*Set up tracking variables local to this iteration* 46c⟩ ≡

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

◇

Fragment referenced in 46a.
Defines: delta_increase_least 46d, ibest, 46d.

5.7.6

⟨*If* self.sites[u] *is chosen for insertion, find best insertion position and update* delta_increase_least_table 46d⟩ ≡

```
    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 46a.
Uses: compute_collinear_horseflies_tour_length 44b, current_tour_length 46b, delta_increase_least 46c, delta_increase_least_table 46a, ibest, 46c, self.inithorseposn, 45b.

5.7.7

⟨ *Find the unvisited site which on insertion increases tour-length by the least amount* 47a⟩ ≡

```
    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 46a.
Uses: delta_increase_least_table 46a.

5.7.8

⟨ *Update states for* PolicyBestInsertionNaive 47b⟩ ≡

```
    # Update visited and univisted 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_idxs = filter( lambda elt: elt != unvisited_site_idx_for_insertion, \
                                        self.unvisited_sites_idxs )

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

Fragment referenced in 46a.
Uses: compute_collinear_horseflies_tour 45a, self.horse_tour 45b, self.inithorseposn, 45b, self.visited_sites, 45b.

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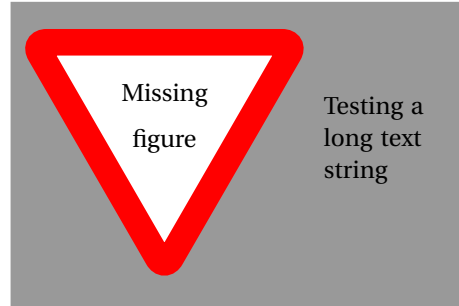
This is clearly a lower-bound on the weight of OPT for Collinear Horsefly. However, I believe that the stronger statement is also true

> **Conjecture 1.** *The weight of the φ-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 φ-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 φ-MSTs? That might help me in designing a local search operation for the Horsefly problem.

Also note, that the construction of this φ-Prim MST can be generalized to two or more flies (single horse) we buld 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 φ we can compute the φ-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 φ-MST as we keep changing the φ-parameter, as one often does while playing with Mathematica's `Manipulate` function. Can we motivate this by saying φ 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 φ-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 φ, 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 optmized 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* 49a⟩ ≡

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

    import networkx as nx
    from sklearn.neighbors import NearestNeighbors

    ⟨Create singleton graph, with node at inithorseposn 49b⟩

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

        ⟨For each node, find the closest site 50a⟩
        ⟨Find the node with the closest site, and generate the next node and edge for the φ-MST 50b⟩

        # Marking means removing from unmarked list :-D
        unmarked_sites_idxs.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 20a.
Defines: compute_phi_prim_mst, Never used, unmarked_sites_idxs 50a.

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 49b⟩ ≡

```
    G = nx.Graph()
    G.add_node(0, mycoordinates=inithorseposn, joined_site_coords=[])
```
    ◇
Fragment referenced in 49a.

5.8.4

⟨ *For each node, find the closest site* 50a ⟩ ≡

```
        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_idxs:
                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 49a.
Uses: unmarked_sites_idxs 49a.

5.8.5

⟨ *Find the node with the closest site, and generate the next node and edge for the φ-MST* 50b ⟩ ≡

```
        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 49a.
Uses: single_site_solution 44a.

Algorithm : Doubling the φ-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* 51⟩ ≡

```
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  ]
```

```python
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 27, 28, 31a, 32b, 38a, 51, 54, 56.
Fragment referenced in 20a.
Uses: tour_length 57a.

5.12.4

⟨*Algorithms for classic horsefly* 54⟩ ≡

```
     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 27, 28, 31a, 32b, 38a, 51, 54, 56.
Fragment referenced in 20a.
Uses: generate_constraints 32a, tour_length 57a.

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 heurtiustics 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* 56⟩ ≡

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

    _, tsp_idxs = tsp.tsp(horseinit_and_sites)


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

    if h != len(tsp_idxs)-1:
        idx_vec = tsp_idxs[h+1:] + tsp_idxs[:h]
    else:
        idx_vec = tsp_idxs[: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 27, 28, 31a, 32b, 38a, 51, 54, 56.
Fragment referenced in 20a.

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* 57a⟩ ≡

```
    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 57ab.
Fragment referenced in 20a.
Defines: tour_length 27, 31a, 51, 54, 59d, 60a.

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* 57b⟩ ≡

```
    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 inital 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 57ab.
Fragment referenced in 20a.
Defines: tour_length_with_waiting_time_included 31a, 32b, 43d, 59d.

Plotting Routines

5.15.1

⟨ *Plotting routines for classic horsefly* 58a⟩ ≡

```
    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* 58b⟩
⟨ *Get x and y coordinates of the sites* 58c⟩
⟨ *Construct the fly-tour from the information about horse tour and sites* 59a⟩
⟨ *Print information about the horse tour* 59b⟩
⟨ *Print information about the fly tour* 59c⟩
⟨ *Print meta-data about the algorithm run* 59d⟩
⟨ *Plot everything* 60a⟩
◇

Fragment defined by 58a, 60b.
Fragment referenced in 20a.
Defines: plotTour 22.

5.15.2

⟨ *Get x and y coordinates of the endpoints of segments on the horse-tour* 58b⟩ ≡

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

Fragment referenced in 58a.

5.15.3

⟨ *Get x and y coordinates of the sites* 58c⟩ ≡

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

Fragment referenced in 58a.

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* 59a⟩ ≡

```
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 58a.

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

⟨ *Print information about the horse tour* 59b⟩ ≡

```
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 58a.

5.15.6

⟨ *Print information about the fly tour* 59c⟩ ≡

```
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 58a.

5.15.7

⟨ *Print meta-data about the algorithm run* 59d⟩ ≡

```
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 58a.
Uses: tour_length 57a, tour_length_with_waiting_time_included 57b.

5.15.8

⟨ *Plot everything* 60a ⟩ ≡

```
    #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 58a.
Uses: tour_length 57a.

5.15.9

⟨ *Plotting routines for classic horsefly* 60b ⟩ ≡

```
    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 58a, 60b.
Fragment referenced in 20a.
Defines: draw_phi_mst 22.

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 scheudle itself.

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

⟨*Animation routines for classic horsefly* 61⟩ ≡

```
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 62a⟩
    ⟨Parse input-output file and set up required data-structures 62b⟩
    ⟨Construct and store every frame of the animation in ims 63a⟩
    ⟨Write animation of schedule to disk and display in live window 65b⟩
```
◇
Fragment referenced in 20a.

5.16.2 In the animation, we are going to show the process of the fly delivering packages to the sites according to the precomputed 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 LaTeX for typesetiing 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* 62a ⟩ ≡

```
    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 61.

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* 62b ⟩ ≡

```
    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 61.

5.16.4 The `ims` array stores each frame of the animation. Every frame consists of various "artist" objects [2] (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` 63a⟩ ≡

```
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 65a⟩

    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_serviced = 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 63b⟩
    ◇
```

Fragment referenced in 61.
Defines: `number_of_sites_serviced` 63b.

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` 63b⟩ ≡

```
    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]]
    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])
```

---

[2]This is Matplotlib terminology

```
        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 63a.
Uses: `number_of_sites_serviced` 63a.

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 <u>and</u> 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* 65a⟩ ≡

```python
    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 63a.

5.16.7

⟨*Write animation of schedule to disk and display in live window* 65b⟩ ≡

```python
    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 61.

Chapter Index of Fragments

Chapter Index of Identifiers

Chapter 6

Reverse Horsefly

Chapter 7

One Horse, Multiple Flies

Module Overview

If one fly wasn't exciting enough for you, how about multiple flies?! The added complexity in the problem comes from finding which sites need to be serviced by each of the flies *and* the order in which these sites need to be serviced. To play around with the algorithms in interactive mode, run `main.py` as

python main.py --problem-one-horse-multiple-flies.

The structure of this chapter is similar to chapter 5. In fact, we will be using some of the algorithms from that chapter as black-box routines in the algorithms to be described here.

All algorithms to solve the multiple flies [1] problem have been implemented in `problem_one_horse_multiple_flies.py`. As bfore, 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_one_horse_multiple_flies.py"` 69≡

⟨ *Relevant imports* 70a ⟩
⟨ *Set up logging information relevant to this module* 70b ⟩
def run_handler():
⟨ *Define key-press handler* 71a ⟩
⟨ *Set up interactive canvas* 73b ⟩

⟨ *Local data-structures* 74a ⟩
⟨ *Algorithms for multiple flies* 76a ⟩
⟨ *Plotting routines* 83 ⟩
⟨ *Animation routines* 84a ⟩
◇

---

[1] For the rest of this chapter we will refer to the one horse, multiple flies problem simply as the multiple flies problem.

Module Details

7.2.1

⟨*Relevant imports* 70a⟩ ≡

```
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

import problem_classic_horsefly as chf
```
◇

Fragment referenced in 69.

7.2.2 The logger variable becomes 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* 70b⟩ ≡

```
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 69.

7.2.3 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 φ, 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* 71a ⟩ ≡

```
# 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 71b⟩
            elif event.key in ['n', 'N', 'u', 'U']:
                ⟨Generate a bunch of uniform or non-uniform random points on the canvas 72b⟩
            elif event.key in ['c', 'C']:
                ⟨Clear canvas and states of all objects 73a⟩
        return _keyPressHandler
    ⋄
```

Fragment referenced in 69.
Defines: wrapperkeyPressHandler 73b.

7.2.4 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.

⟨ *Start entering input from the command-line* 71b ⟩ ≡

```
    ⟨Set speed and number of flies 71c⟩
    ⟨Select algorithm to execute 72a⟩
    ⋄
```

Fragment referenced in 71a.

7.2.5 We assume that all flies have the same velocity

⟨ *Set speed and number of flies* 71c ⟩ ≡

```
    phi_str = raw_input(Fore.YELLOW + "What should I set the speed of each of the flies to be (should be >1)? : " + Style.RESET_ALL)
    nof_str = raw_input(Fore.YELLOW + "How many flies do you want me to assign to the horse? : " + Style.RESET_ALL)

    phi = float(phi_str)
    nof = int(nof_str)
    ⋄
```

Fragment referenced in 71b.

7.2.6 Each of the algorithms can have several tuning strategies. Depending on the algorithm selected, further sub-menus will have to be generated for selecting these sub-strategies. It is best, if all these strategies, are all set through a configuration file, like say YAML, rather than have to generate the menus.

What configurations are valid or not will have to be set later. However, for now, I will only implement a simple menu ala classic horsefly to get something working. For now, I am implementing the super-drone heuristic with the greedy-incremental strategy for the super-drone. For the super-drone category, we will also have to specify a partitioning scheme of which sites get assigned to which drones.

Also the post-optimizer for the super-drone will have to be specified. Too....many....flags! Needs a careful documenting in terms of tables of what is allowed and what is not allowed that is available for ready-reference for the user, and hopefully one that is updated automatically when the combination is made. Maybe this can be useful for the defense.

For now, we just stick to super-drones

⟨ *Select algorithm to execute* 72a ⟩ ≡

```
algo_str = raw_input(Fore.YELLOW                                        +\
        "Enter algorithm to be used to compute the tour:\n Options are:\n"  +\
        " (ec)   Earliest Capture \n"                                   +\
        Style.RESET_ALL)


algo_str = algo_str.lstrip()


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


if   algo_str == 'ec':
     tour = run.getTour( algo_greedy_earliest_capture, phi, \
                            number_of_flies = nof)
else:
     print "Unknown option. No horsefly for you! ;-D "
     sys.exit()


utils_graphics.applyAxCorrection(ax)
plot_tour(ax, tour)
fig.canvas.draw()
◇
```

Fragment referenced in 71b.
Uses: getTour 74c.

7.2.7   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 no 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 incase 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* 72b ⟩ ≡

```
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 71a.

Uses: clearAllStates 74b.

7.2.8   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* 73a⟩ ≡

```
run.clearAllStates()
ax.cla()

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

fig.texts = []
fig.canvas.draw()
◇
```

Fragment referenced in 71a.
Uses: clearAllStates 74b.

7.2.9

⟨ *Set up interactive canvas* 73b⟩ ≡

```
fig, ax =  plt.subplots()
run = MultipleFliesInput()
#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 69.
Uses: wrapperkeyPressHandler 71a.

Local Data Structures

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

⟨*Local data-structures* 74a⟩ ≡

```
class MultipleFliesInput:
      def __init__(self, sites=[], inithorseposn=()):
            self.sites          = sites
            self.inithorseposn  = inithorseposn

      ⟨Methods for MultipleFliesInput 74b,... ⟩
```
◇

Fragment referenced in 69.
Defines: HorseFlyInput Never used.

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

⟨*Methods for* MultipleFliesInput 74b⟩ ≡

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

Fragment defined by 74bc.
Fragment referenced in 74a.
Defines: clearAllStates 72b, 73a.

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

⟨*Methods for* MultipleFliesInput 74c⟩ ≡

```
def getTour(self, algo, speedratio, number_of_flies):
      return algo(self.sites, self.inithorseposn, speedratio, number_of_flies)
```
◇

Fragment defined by 74bc.
Fragment referenced in 74a.
Defines: getTour 72a, 92a.

Algorithm: Greedy: Earliest Capture

Algorithmic Overview

7.5.1 This algorithm is an attempt to directly generalize the greedy nearest neighbor algorithm for collinear horseflies. The intuition behind this strategy, is that we try to greedily minimize the time between two successive rendezvous points of the horse with any of the flies. Once the horse meets up with a fly, there are several alternatives for the next site that fly should be deployed to. We again follow a greedy strategy here, and deploy it to the nearest "unclaimed" site [2]. The next chunk introduces the necessary terminology and a detailed algorithmic implementation of the earliest capture heuristic. See Figure?? for an illustration of the progress of the algorithm for 2 drones and 6 sites.

7.5.2 We say that a site is "claimed" when some fly is heading towards it *or* has been serviced by a fly already. In each iteration of the `while` loop, the horse meets up with one of the deployed flies. The horse moves towards the site assigned to the selected fly along the segment joining the horse's current position and that site. The horse keeps moving along this segment until it meets the fly.

As the horse moves towards this site, the remaining flies, if they are returning from sites that they have just serviced, change their direction of motion and move towards the rendezvous point of the horse and selected fly. On meeting up with a fly, the horse deploys it to an unclaimed site, if one exists and updates the corresponding `FlyState` object. We keep repeating this process until all sites have been serviced and all flies have been retired.

A fly is deemed "retired" when it returns to the horse and will no longer be deployed to *any* site [3]. This will typically happen when the number of unclaimed sites near the end of the algorithm's run is less than the total number of flies. The `FlyState` class tracks information (such as trajectory, current assigned site etc.) about a single fly from the moment it was deployed from `inithorseposn` till it returns to the horse after its last delivery.

In this implementation, we will deploy a fly to the nearest *unclaimed* site at each rendezvous point with the horse. [4]

Also note that we need to take into special consideration the case where the number of flies is greater than the number of sites. In this case, the extra flies won't help reduce the makespan, so we set the total number of flies to the number of sites inside the function.

However, if we introduce assumptions ala Package Handoff, where the extra drones act as "butlers" for the returning drones meeting them midway and handing off packages to them then of course having the extra drones would indeed help. This package handoff situation will need to be explored in detail later.

---

[2]There might be several interesting algorithmic questions lurking on how to select the next unclaimed site that the fly should be deployed to. Can we do some analogue of incremental? Fast methods for detecting the best place of insertion would be eminently interesting

[3]It continutes lugging along with the horse after this point in time, so to speak

[4]Although I will need to be able to configure this via some policy argument later.

⟨ *Algorithms for multiple flies* 76a ⟩ ≡

    ⟨ *Helper functions for* algo_greedy_earliest_capture 79a ⟩
    ⟨ *Definition of the* FlyState *class* 76b ⟩

```
def algo_greedy_earliest_capture(sites, inithorseposn, phi, number_of_flies,\
                                 write_algo_states_to_disk_p = True,\
                                 write_io_p                  = True,\
                                 animate_tour_p              = True) :
```

    ⟨ *Set algo-state and input-output files config* 81b ⟩

```
    if number_of_flies > len(sites):
          number_of_flies = len(sites)

    current_horse_posn = np.asarray(inithorseposn)
    horse_traj         = [(current_horse_posn, None)]
```

    ⟨ *Find the k-nearest sites to* inithorseposn *for k =* number_of_flies *and claim them* 77a ⟩
    ⟨ *Initialize one* FlyState *object per fly for all flies* 77b ⟩

```
    all_flies_retired_p = False

    while (not all_flies_retired_p):
```
        ⟨ *Find the index of the fly F which can meet the horse at the earliest, the rendezvous point R, and time till rendezvous* 78a ⟩
        ⟨ *Update fly trajectory in each* FlyState *object till F meets the horse at R* 79b ⟩
        ⟨ *Update* current_horse_posn *and horse trajectory* 80c ⟩
        ⟨ *Deploy F to an unclaimed site if one exists and claim that site, otherwise retire F* 80d ⟩
        ⟨ *Calculate value of* all_flies_retired_p 81a ⟩
        ⟨ *Write algorithms current state to file, if* write_algo_states_to_disk_p == True 81c ⟩

    ⟨ *Write input and output to file if* write_io_p == True 82a ⟩
    ⟨ *Animate compute tour if* animate_tour_p == True 82b ⟩
    ⟨ *Return multiple flies tour* 82c ⟩

   ◇

Fragment referenced in 69.

Algorithmic Details

7.6.1

⟨ *Definition of the* FlyState *class* 76b ⟩ ≡

```
    class FlyState:
        def __init__(self, idx, initflyposn, site, flyspeed):

            self.idx                           = idx
            self._flytraj                      = [ {'coordinates': np.asarray(initflyposn), 'type':'gen_pt'} ]
            self._current_assigned_site        = np.asarray(site)
            self._speed                        = flyspeed
            self._current_assigned_site_serviced_p  = False
            self._fly_retired_p                = False

        def retire_fly(self):
            self._fly_retired_p = True

        def deploy_to_site(self,site):
            self._current_assigned_site             = np.asarray(site)
            self._current_assigned_site_serviced_p = False

        def is_retired(self):
```

```
            return self._fly_retired_p

    def is_current_assigned_site_serviced(self):
        return self._current_assigned_site_serviced_p

    def get_current_fly_position(self):
        return self._flytraj[-1]['coordinates']

    def get_trajectory(self):
        return self._flytraj
```

⟨*Definition of method* update_fly_trajectory 79c⟩
⟨*Definition of method* rendezvous_time_and_point_if_selected_by_horse 78b⟩

◇

Fragment referenced in 76a.

7.6.2 At the beginning of the algorithm, if the horse has k flies, the k nearest sites to the initial position of the horse will be claimed by the flies for service.

⟨*Find the k-nearest sites to* inithorseposn *for k =*number_of_flies *and claim them* 77a⟩ ≡

```
    from sklearn.neighbors import NearestNeighbors

    neigh = NearestNeighbors(n_neighbors=number_of_flies)
    neigh.fit(sites)

    _, knn_idxss = neigh.kneighbors([inithorseposn])
    knn_idxs    = knn_idxss.tolist()[0]
    knns        = [sites[i] for i in knn_idxs]
    unclaimed_sites_idxs = list(set(range(len(sites))) - set(knn_idxs)) # https://stackoverflow.com/a/3462160
```
◇

Fragment referenced in 76a.

7.6.3

⟨*Initialize one* FlyState *object per fly for all flies* 77b⟩ ≡
```
    flystates = []
    for i in range(number_of_flies):
        flystates.append(FlyState(i,inithorseposn, knns[i], phi))
```
◇

Fragment referenced in 76a.

7.6.4 We need to find the index of a fly that the horse can rendezvous with at the earliest. To do this, I just do a linear search over all fly-states. While this search is linear in the number of flies, it will be interesting to see what we can do to make such repeated queries faster? Seems like we will have to keep some sort of priority queue to speed up future searches. I am sure we can put together some standard computational geometry ideas together for this. This **is** an interesting little data-structural problem in its own right though.

⟨ *Find the index of the fly **F** which can meet the horse at the earliest, the rendezvous point R, and time till rendezvous* 78a⟩ ≡

```
imin  = 0
rtmin = np.inf
rptmin= None
for i in range(number_of_flies):
    if flystates[i].is_retired():
        continue
    else:
        rt, rpt = flystates[i].rendezvous_time_and_point_if_selected_by_horse(current_horse_posn)
        if rt < rtmin:
            imin  = i
            rtmin = rt
            rptmin = rpt
```

◇

Fragment referenced in 76a.
Uses: rendezvous_time_and_point_if_selected_by_horse 78b.

7.6.5

⟨ *Definition of method* rendezvous_time_and_point_if_selected_by_horse 78b⟩ ≡

```
def rendezvous_time_and_point_if_selected_by_horse(self, horseposn):
    assert(self._fly_retired_p != True)

    if self._current_assigned_site_serviced_p:
         rt = meeting_time_horse_fly_opp_dir(horseposn, self.get_current_fly_position(), self._speed)
         horseheading = self.get_current_fly_position() - horseposn
    else:
        distance_to_site    = np.linalg.norm(self.get_current_fly_position() -\
                                              self._current_assigned_site)
        time_of_fly_to_site = 1/self._speed * distance_to_site

        horse_site_vec   = self._current_assigned_site - horseposn
        displacement_vec = time_of_fly_to_site * horse_site_vec/np.linalg.norm(horse_site_vec)
        horseposn_tmp    = horseposn + displacement_vec

        time_of_fly_from_site = \
                meeting_time_horse_fly_opp_dir(horseposn_tmp, self._current_assigned_site, self._speed)

        rt = time_of_fly_to_site + time_of_fly_from_site
        horseheading = self._current_assigned_site - horseposn

    uhorseheading = horseheading/np.linalg.norm(horseheading)
    return rt, horseposn + uhorseheading * rt
```

◇

Fragment referenced in 76b.
Defines: rendezvous_time_and_point_if_selected_by_horse 78a.
Uses: meeting_time_horse_fly_opp_dir 79a.

7.6.6 This fragment defines the function meeting_time_horse_fly_opp_dir that was used in the previous fragment. If a horse with speed 1.0 and fly with speed φ are present at opposite endpoints of a segment of length L it takes time $\frac{L}{\varphi+1}$ to meet up if they travel towards each other along the segment.

Describe a
via asympt
what calcu
performed
into two ca
turning aft
a fly heade
site it is su
vice

⟨ *Helper functions for* `algo_greedy_earliest_capture` 79a⟩ ≡

```
def meeting_time_horse_fly_opp_dir(horseposn, flyposn, flyspeed):
    horseposn = np.asarray(horseposn)
    flyposn   = np.asarray(flyposn)
    return 1/(flyspeed+1) * np.linalg.norm(horseposn-flyposn)
```
    ◇

Fragment referenced in 76a.
Defines: `meeting_time_horse_fly_opp_dir` 78b.

7.6.7   Now that we know the rendezvous point for the horse with one of the flies and the time it takes for the horse to get there, update the trajectories of all the flies accordingly.

⟨ *Update fly trajectory in each* `FlyState` *object till* ***F*** *meets the horse at R* 79b⟩ ≡
```
    for flystate in flystates:
        flystate.update_fly_trajectory(rtmin, rptmin)
```
    ◇

Fragment referenced in 76a.

7.6.8   Depending on the position of the rendezvous point, the time it will take for the horse to get there and the current position of a non-retired fly [5], we update its state according to one of three mutually exclusive cases (also illustrated graphically in the next three fragments.)

1. The site currently assigned to the fly been serviced and the fly is headed back for picking up its next package from the horse

2. The fly is headed towards its currently assigned site, but won't be able to make it to the site in the time dt it takes for the horse to reach the rendezvous point.

3. The site assigned to the fly has not yet been serviced, but <u>will</u> reach to the site within time dt. In this case, once the fly reaches the site it needs to make a sort of "u-turn" at the site and head towards the rendezvous point.

⟨ *Definition of method* `update_fly_trajectory` 79c⟩ ≡

```
    def update_fly_trajectory(self, dt, rendezvous_pt):

        if self.is_retired():
            return

        dx = self._speed * dt

        if self._current_assigned_site_serviced_p :
            ⟨Move towards the provided rendezvous point 79d⟩

        elif dx < np.linalg.norm(self._current_assigned_site - self.get_current_fly_position()) :
            ⟨Continue moving towards the site 80a⟩
        else:
            ⟨Move towards the site mark site as serviced and then head towards rendezvous point 80b⟩
```
    ◇

Fragment referenced in 76b.

7.6.9   Insert figure

⟨ *Move towards the provided rendezvous point* 79d⟩ ≡

```
    heading  = rendezvous_pt - self.get_current_fly_position()
    uheading = heading / np.linalg.norm(heading)
    newpt    = self.get_current_fly_position() + dx * uheading
    self._flytraj.append(  {'coordinates': newpt, 'type': 'gen_pt'}  )
```
    ◇

Fragment referenced in 79c.

7.6.10   Insert figure

---

[5] If the fly has been retired there is no need for an update to its trajectory

⟨ *Continue moving towards the site* 80a⟩ ≡

```
        heading  = self._current_assigned_site - self.get_current_fly_position()
        uheading = heading / np.linalg.norm(heading)
        newpt    = self.get_current_fly_position() + dx * uheading
        self._flytraj.append(  {'coordinates': newpt, 'type': 'gen_pt'}  )
        ◇
```
Fragment referenced in 79c.

7.6.11

Insert figu

⟨ *Move towards the site mark site as serviced and then head towards rendezvous point* 80b⟩ ≡

```
        dx_reduced = dx - np.linalg.norm(self._current_assigned_site -\
                                        self.get_current_fly_position())
        heading  = rendezvous_pt - self._current_assigned_site
        uheading = heading/np.linalg.norm(heading)

        newpt = self._current_assigned_site + uheading * dx_reduced
        self._current_assigned_site_serviced_p = True
        self._flytraj.extend([{'coordinates':self._current_assigned_site, 'type':'site'},
                              {'coordinates':newpt,                        'type':'gen_pt'}])
        ◇
```
Fragment referenced in 79c.

7.6.12   Now that the horse has reached the rendezvous point and met **F**, we need to update the horse's trajectory and decide which unclaimed site **F** should go to next.

⟨ *Update* current_horse_posn *and horse trajectory* 80c⟩ ≡
```
        current_horse_posn = rptmin
        horse_traj.append((np.asarray(rptmin),imin))
        ◇
```
Fragment referenced in 76a.

7.6.13

⟨ *Deploy* **F** *to an unclaimed site if one exists and claim that site, otherwise retire* **F** 80d⟩ ≡

```
        if  unclaimed_sites_idxs:
            unclaimed_sites = [sites[i] for i in unclaimed_sites_idxs]

            neigh = NearestNeighbors(n_neighbors=1)
            neigh.fit(unclaimed_sites)

            _, nn_idxss = neigh.kneighbors([current_horse_posn])
            nn_idx      = nn_idxss.tolist()[0][0]

            flystates[imin].deploy_to_site(unclaimed_sites[nn_idx])
            unclaimed_sites_idxs = list(set(unclaimed_sites_idxs) - \
                                        set([unclaimed_sites_idxs[nn_idx]]))

        else:
            flystates[imin].retire_fly()
        ◇
```
Fragment referenced in 76a.

7.6.14   This chunk just loops through all flies and checks if all flies have been retired or not. The algorithm stops when all flies have been retired which indicates all sites have been serviced and all flies have returned back to the horse.

⟨ *Calculate value of* `all_flies_retired_p` 81a ⟩ ≡

```
acc = True
for i in range(number_of_flies):
    acc = acc and flystates[i].is_retired()
all_flies_retired_p = acc
```
◇

Fragment referenced in 76a.

7.6.15 As before, for the purposes of algorithmic analysis, we will need to write to disk the algorithm's state at the end of each iteration of the `while` loop. Each algorithm state will consist of

1. The coordinates of the sites serviced so far, and the index of the fly that serviced each of them

2. The coordinates unserviced sites

3. The trajectory of each fly stored as a list of points. We also mark each point on a fly's trajectory as either

   - A site serviced by a fly

   - The fly's position $\mathbb{R}^2$ where the fly either rendezvous with the horse or its position when some *other* fly is rendezvousing with the horse at that moment at that moment.

4. The trajectory of the horse, which consists of a sequence of rendezvous points with the flies, one fly for each rendezvous point, whose index we store along the rendezvous point's coordinates.

Finally, we write out the input provided (sites, initial horse position, number of flies and φ) and the trajectories of the horse and flies (in the format described in the a above list) to reconstruct a problem exactly and to animate the trajectories. The animation is constructed as an `.avi` file

All these files are written to a folder named after the algorithm and the time-stamp of the algorithm's run.

⟨ *Set algo-state and input-output files config* 81b ⟩ ≡

```
import sys, datetime, os, errno
algo_name     = 'algo-greedy-earliest-capture'
time_stamp    = datetime.datetime.now().strftime('Day-%Y-%m-%d_ClockTime-%H:%M:%S')
dir_name      = algo_name + '---' + time_stamp
io_file_name  = 'input_and_output.yml'

try:
    os.makedirs(dir_name)
except OSError as e:
    if e.errno != errno.EEXIST:
        raise
algo_state_counter = 1
```
◇

Fragment referenced in 76a.

7.6.16

⟨ *Write algorithms current state to file, if* `write_algo_states_to_disk_p == True` 81c ⟩ ≡

```
print "Algorithm State Number: ", algo_state_counter
if write_algo_states_to_disk_p:
    algo_state_file_name = 'algo_state_' + str(algo_state_counter).zfill(5) + '.yml'

    data = {'horse_trajectory' : horse_traj, \
            'fly_trajectories' : [flystates[i].get_trajectory() for i in range(number_of_flies)] }
    utils_algo.write_to_yaml_file(data, dir_name=dir_name, file_name=algo_state_file_name)
algo_state_counter += 1
```
◇

Fragment referenced in 76a.

7.6.17 Store input and output data for post analyses and plotting.

⟨*Write input and output to file if* write_io_p == True 82a⟩ ≡

```
if write_io_p:
    data = { 'sites' : sites, \
             'inithorseposn' : inithorseposn,\
             'phi':phi,\
             'horse_trajectory' : horse_traj, \
             'fly_trajectories' : [flystates[i].get_trajectory()
                                   for i in range(number_of_flies)] }
    utils_algo.write_to_yaml_file(data, dir_name = dir_name, file_name = io_file_name)
```
    ◇

Fragment referenced in 76a.

7.6.18 The animation of the computed tour is done in a separate fig window.

⟨*Animate compute tour if* animate_tour_p == True 82b⟩ ≡

```
if animate_tour_p:
    animate_tour(sites            = sites,
                 inithorseposn    = inithorseposn,
                 phi              = phi,
                 horse_trajectory = horse_traj,
                 fly_trajectories = [flystates[i].get_trajectory() for i in range(number_of_flies)],
                 animation_file_name_prefix = dir_name + '/' + io_file_name)
```
    ◇

Fragment referenced in 76a.

7.6.19

⟨*Return multiple flies tour* 82c⟩ ≡

```
return {'sites' : sites, \
        'inithorseposn' : inithorseposn,\
        'phi':phi,\
        'horse_trajectory': horse_traj, \
        'fly_trajectories': [flystates[i].get_trajectory() for i in range(number_of_flies)]}
```
    ◇

Fragment referenced in 76a.

Plotting Routines

7.7.1 When an algorithm returns the computed tour, plot that tour upon the provided axis object. This is good for quick interactive testing of algorithms, where you can immediately see the output of various algorithms on the canvas.

⟨ *Plotting routines* 83 ⟩ ≡

```python
def plot_tour(ax, tour):

    sites           = tour['sites']
    inithorseposn   = tour['inithorseposn']
    phi             = tour['phi']
    horse_trajectory = tour['horse_trajectory']
    fly_trajectories = tour['fly_trajectories']

    xhs = [ horse_trajectory[i][0][0] for i in range(len(horse_trajectory))]
    yhs = [ horse_trajectory[i][0][1] for i in range(len(horse_trajectory))]

    number_of_flies = len(fly_trajectories)
    colors          = utils_graphics.get_colors(number_of_flies, lightness=0.4)

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

    # Plot fly trajectories
    xfss = [[point['coordinates'][0] for point in fly_trajectories[i]] for i in range(len(fly_trajectories))]
    yfss = [[point['coordinates'][1] for point in fly_trajectories[i]] for i in range(len(fly_trajectories))]

    for xfs, yfs,i in zip(xfss,yfss,range(number_of_flies)):
        ax.plot(xfs,yfs,color=colors[i], alpha=0.7)

    # Plot sites along each flys tour
    xfsitess = [ [point['coordinates'][0] for point in fly_trajectories[i] if point['type'] == 'site']
                for i in range(len(fly_trajectories))]
    yfsitess = [ [point['coordinates'][1] for point in fly_trajectories[i] if point['type'] == 'site']
                for i in range(len(fly_trajectories))]

    for xfsites, yfsites, i in zip(xfsitess, yfsitess, range(number_of_flies)):
        for xsite, ysite, j in zip(xfsites, yfsites, range(len(xfsites))):
            ax.add_patch(mpl.patches.Circle((xsite,ysite), radius = 1.0/140, \
                                            facecolor=colors[i], edgecolor='black'))
            ax.text(xsite, ysite, str(j+1), horizontalalignment='center',
                                            verticalalignment='center'  ,
                                            bbox=dict(facecolor=colors[i], alpha=1.0))
    # Plot horse tour
    ax.plot(xhs,yhs,'o-',markersize=5.0, linewidth=2.5, color='#D13131')

    # Plot initial horseposn
    ax.add_patch( mpl.patches.Circle( inithorseposn,radius = 1.0/100,
                                    facecolor= '#D13131', edgecolor='black'))
```

◇

Fragment referenced in 69.

Animation routines

It is even more important to animate a multiple horseflies tour than for classic horsefly since it will yield greater geometric understanding of the heuristics implemented. With multiple flies the final result is just a mess of lines, with the final horse and flies tours together resembling a kindergarterner's immitation of a Jackson Pollock painting.

Here are the goal for the animation

- The path of the horse is rendered in a bold shade of red.

- Unserviced sites are marked by black dots

- Each drone is assigned a color for its trajectory. The trajectory lines are all polygonal curves and are rendered with a somewhat transparent shade of its assigned color

- Serviced sites have a face-color which is the same assigned to the drone that served it. The face-colors are delibertely chosen using the HSV or HSL colorschemes; the colors chosen are on the brighter side.

Ideally, I would like the animation to look like as if all the black dots (the circles) are disappearing one-by-one. Further one can also see which site was serviced by which drone.

A feature you might also want to add, is that at a key-press, you can focus only on the tours of the horse and a particular fly along with the ordering chosen for that fly. Thus for instance pressing "a 4 12" would animate the motion of drones 4 and 12, if there are say 20 drones being used. Similarly, "p 4 12" would just plot the tour of the 4th 12th drone. i.e. we just specify a list of indices after "a" or "p". If one of the numbers in the list is more than the number of drones, then we bleep a message at the command-line terminal to enter the corrected drone list again.

Now that the plan has been outlined, it is time to get down the nitty-gritties of the implementation.

7.8.1

⟨*Animation routines* 84a⟩ ≡

```
def animate_tour (sites, inithorseposn, phi, horse_trajectory, fly_trajectories, animation_file_name_prefix):
    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 all necessary graphics 84b⟩
    ⟨Parse trajectory information and convert trajectory representation to leg list form 86a⟩
    ⟨Construct and store every frame of the animation in the ims array 86b⟩
    ⟨Write animation of tour to disk and display in live window 89⟩

◇
```
Fragment referenced in 69.

7.8.2 It is important that we have a background grid (both a major and a minor) one for tracking the motion of the horse and drones to get a rough sense of estimation of the distances involved while looking at the motion. Since multiple drones are used I set up a `colors` array containig the rgb values of colors that are chromatically distinct when viewed against a white canvas.

⟨*Set up configurations and parameters for all necessary graphics* 84b⟩ ≡

```
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([])

# Visually distinct colors for displaying each flys trajectory in a different color
number_of_flies = len(fly_trajectories)
colors          = utils_graphics.get_colors(number_of_flies, lightness=0.5)

ax.set_title("Number of sites: " + str(len(sites)), fontsize=25)
ax.set_xlabel(r"$\varphi=$ " + str(phi) + "\nNumber of flies: " + str(number_of_flies), fontsize=20)
◇
```

Fragment referenced in 84a.

7.8.3   For the purposes of animation it is most convenient to represent a trajectory not as a list of points, but as a list of "legs". In the case of the horse a leg is simply a single straight line segment joining two consecutive points on the tour.

For the fly a leg is either a single segment joining two successive points on the tour of type genpt or two successive segments whose vertices are respectively of type genpt, site and genpt. _____ Insert diag senting wh

To make the programming convenient, we will enforce the condition that all fly trajectories flies have the same number of legs as that of the horse's by padding None elements at the end of each fly trajectory's "leg-list" representation. Legs of type None on a fly's trajectory indicate that the fly has been retired and hence has stopped moving.

The leg list version of the horse and fly trajectories are respectively named horse_traj_ll and fly_trajs_ll.

Note that the segment corresponding to each horse-leg also stores the index of the fly it meets up with at the head of the corresponding segment vector.

⟨ *Parse trajectory information and convert trajectory representation to leg list form* 86a⟩ ≡

```
    # Leg list form for all horse trajectories
    horse_traj_ll = []
    for i in range(len(horse_trajectory)-1):
        horse_traj_ll.append((horse_trajectory[i][0], horse_trajectory[i+1][0],
                               horse_trajectory[i+1][1]))

    # Leg list form for all fly trajectories
    fly_trajs_ll = []
    for fly_traj in fly_trajectories:
        fly_traj_ll = []
        for i in range(len(fly_traj)-1):
            if fly_traj[i]['type'] == 'gen_pt':

                if fly_traj[i+1]['type'] == 'gen_pt':
                    fly_traj_ll.append((fly_traj[i],
                                        fly_traj[i+1]))


                elif fly_traj[i+1]['type'] == 'site':
                    fly_traj_ll.append((fly_traj[i], \
                                        fly_traj[i+1], \
                                        fly_traj[i+2]))
        fly_trajs_ll.append(fly_traj_ll)

    num_horse_legs = len(horse_traj_ll)

    # Append empty legs to fly trajectories so that leg counts
    # for all fly trajectories are the same as that of the horse
    # trajectory
    for fly_traj in fly_trajs_ll:
        m = len(fly_traj)
        empty_legs = [None for i in range(num_horse_legs-len(fly_traj))]
        fly_traj.extend(empty_legs)
```

◇

Fragment referenced in 84a.

7.8.4

⟨ *Construct and store every frame of the animation in the* ims *array* 86b⟩ ≡

```
    ⟨ Define discretization function for a leg of the horse or fly tour 88⟩
    ims                 = []
    horse_points_so_far = []
    fly_points_so_far   = [[] for i in range(number_of_flies)]
    fly_sites_so_far    = [[] for i in range(number_of_flies)] # each list is definitely a sublist of corresponding list in fly_poi
    for idx in range(len(horse_traj_ll)):
        # Get current horse-leg and update the list of points covered so far by the horse
        horse_leg = (horse_traj_ll[idx][0], horse_traj_ll[idx][1])
        horse_points_so_far.append(horse_leg[0]) # attach the beginning point of the horse leg
        horse_leg_pts = horse_leg
        #utils_algo.print_list(horse_points_so_far)
        #print ".................................................."

        fly_legs  = [fly_trajs_ll[i][idx] for i in range(len(fly_trajs_ll)) ]
        fly_legs_pts  = []
        for fly_leg, i in zip(fly_legs, range(len(fly_legs))):
            if fly_leg != None:
                coods = []
                for pt in fly_leg:
                    coods.append(pt['coordinates'])
```

```
            fly_legs_pts.append(coods)
            fly_points_so_far[i].append(coods[0]) # attaching the beginning point of the leg. Extension only
                                                  # happens for legs wqhich are not of type None, meshing well
                                                  # with the fact that fly has stopped moving.
        else:
            fly_legs_pts.append(None)


# discretize current leg, for horse and fly, and for each point in the discretization
# render the frame. If a fly crosses a site, update the fly_points_so_far list
horse_leg_disc = discretize_leg(horse_leg_pts)    # list of points
fly_legs_disc  = map(discretize_leg, fly_legs_pts) # list of list of points


# Each iteration of the following loop tacks on a new frame to ims
# this outer level for loop is just proceeding through each position
# in the discretized horse legs. This is the motion which coordinates
# the flys motions
for k in range(len(horse_leg_disc['points'])):
    current_horse_posn = horse_leg_disc['points'][k]
    current_fly_posns  = []  # updated in the for loop below.
    for j in range(len(fly_legs_disc)):
        if fly_legs_disc[j] != None:
            current_fly_posns.append(fly_legs_disc[j]['points'][k])

            if fly_legs_disc[j]['legtype'] == 'gsg' and k==9: # yay, we just hit a site!
                fly_points_so_far[j].append(fly_legs_disc[j]['points'][k])
                fly_sites_so_far[j].append(fly_legs_disc[j]['points'][k])
        else:
            current_fly_posns.append(None)
    objs = []
    # Plot sites as black circles
    for site in sites:
        circle = Circle((site[0], site[1]), 0.01, \
                        facecolor = 'k'    , \
                        edgecolor = 'black'    , \
                        linewidth=1.0)
        sitepatch = ax.add_patch(circle)
        objs.append(sitepatch)


    # Plot trajectory of flies
    assert(len(fly_points_so_far) == number_of_flies)
    for ptraj, i in zip(fly_points_so_far, range(number_of_flies)):
        print current_fly_posns[i]
        if current_fly_posns[i] is None:
            xfs = [pt[0] for pt in ptraj]
            yfs = [pt[1] for pt in ptraj]

        else:
            xfs = [pt[0] for pt in ptraj] + [current_fly_posns[i][0]]
            yfs = [pt[1] for pt in ptraj] + [current_fly_posns[i][1]]

        flyline, = ax.plot(xfs, yfs, '-', linewidth=2.5, alpha=0.30, color=colors[i])
        flyloc   = Circle((xfs[-1], yfs[-1]), 0.01, facecolor = colors[i], alpha=0.7)
        flypatch = ax.add_patch(flyloc)
        objs.append(flypatch)
        objs.append(flyline)


    # Plot currently covered sites as colored circles
    for sitelist, i in zip(fly_sites_so_far, range(number_of_flies)):
      for site in sitelist:
            circle = Circle((site[0], site[1]), 0.015, \
                            facecolor = colors[i]    , \
                            edgecolor = 'black'     , \
                            linewidth=1.0)
```

```
                    sitepatch = ax.add_patch(circle)
                    objs.append(sitepatch)

        # Plot trajectory of horse
        xhs = [pt[0] for pt in horse_points_so_far] + [current_horse_posn[0]]
        yhs = [pt[1] for pt in horse_points_so_far] + [current_horse_posn[1]]
        horseline, = ax.plot(xhs,yhs,'-',linewidth=5.0, markersize=6, alpha=1.00, color='#D13131')
       horseloc  = Circle((current_horse_posn[0], current_horse_posn[1]), 0.01, facecolor = '#D13131', alpha=1.00)
        horsepatch = ax.add_patch(horseloc)
        objs.append(horsepatch)
        objs.append(horseline)

        debug(Fore.CYAN + "Appending to ims "+ Style.RESET_ALL)
        ims.append(objs) # [::-1] means reverse the list
    ◇
```

Fragment referenced in 84a.

7.8.5 This function just places a fixed number of points along each segment of a leg for both the horse and flies. This causes the horse and flies to move slowly along short segments and faster on longer segments. To make the animation more uniform, you might actually want to place the discretization points along the leg segments by a scheme of dividing the length of the longest segment by the shortest segment, and then taking a multiple of the answer's floor. For multiple flies, I would then take the greatest common divisor of these numbers and place points every so often along them.

But this scheme will take more time ti implement and not worth the extra work for now.

⟨ *Define discretization function for a leg of the horse or fly tour* 88 ⟩ ≡

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

        if pts == None:
            return None
        else:
            numpts = len(pts)

            if numpts == 2:    # horse leg or fly-leg of type gg
                k  = 19
                legtype = 'gg'
            elif numpts == 3: # fly leg of type gsg
                k  = 10
                legtype = 'gsg'

            pts = map(np.asarray, pts)
            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 {'points': subleg_pts,
                    'legtype'  : legtype}
    ◇
```

Fragment referenced in 86b.

7.8.6 It is important that I save the the animation to disk before I render the animation into a live window. If I do `plt.show()` first, then after closing the live-window the animation does not seem to get saved to disk and everything just hangs.

Not sure if this is only an Ubuntu thing. For the purposes of experiments, you just need to comment the `plt.show()` line, at the end of the block.

⟨ *Write animation of tour to disk and display in live window* 89 ⟩ ≡

```
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)

print "Mike testing, mike testing"
#debug(Fore.MAGENTA + "\nStarted writing animation to disk"+ Style.RESET_ALL)
#ani.save(animation_file_name_prefix+'.avi', dpi=250)
#debug(Fore.MAGENTA + "\nFinished writing animation to disk"+ Style.RESET_ALL)

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

Fragment referenced in 84a.

$\boxed{\text{Experiment}}$: Decrease in Tour Length with a Larger Number of Drones

7.9.1 The point of using multiple drones is to get a further decrease in the tour-length of the horse. Being a super-difficult NP-hard problem, answering this question might seem intractable.

However, as a proxy we study the drop in the tour length of the horse as the number of drones increases for the different heuristics. This might give us some kind of insight into the rate of decrease for the original problem. My hope is that the decrease in tour length is linear (doubling the number of drones should decrease the tour-length by half) Different algorithms might give different rates of decrease and it would be a good idea to compare them on this axis too. Another point of interest would be if the rate of the fall of the tour length depends somehow on the speed ratio φ. All this looks like a job for the design of experiments which I will study later exactly for this purpose of divising good computational experiments.

7.9.2 Thus far, only one algorithm has been implemented i.e. the earliest capture heuristic. Whatever experimental setup, I do for this experiment will be equally valid for other algorithms and for experiments in other chapters.

7.9.3 How should the experiment be run? Well, all experiments go into a dedicated folder. Each file has a dedicated main function, that coordinates the input and output required for the experiment. All experimental output data will be stored as YAML files, for the sake of later plotting. The resulting data-files will be written as run folders in the same `expts` directory. The directory will be stamped with the time of the experiments. You might want to keep a `clean.sh` script to remove unnecessary folders and data-files. But the nature of the script will be revealed as I do the coding, not right now....

7.9.4 A point to note is that there will be no interative canvas involved here. Everything will be dictated at the terminal. Further, many of the parameters under which the experiment must be run will be dictated by command-line arguments. Examples of parameters relevant to this particular experiment are

- The speed-ratio φ.
- The number N of points chosen.
- The initial position of the horse and flies (what I have been terming `inithorseposn` in the codes).
- The number k of flies to be used in each run of the algorithm.
- Other considerations that will come up during the run of the algorithm, that I will keep documenting above this line.

There are lots of factors and lots of responses, indeed some consideration will have to be given to the design of experiments for the comparison of various algorithms and giving algorithmic recommendations.

7.9.5 Here is the layout of the experiment file. It will be good to have a sophisticated parser to read command-line input as a python code especially for lists when trying to specify the number of drones used as a list as a list comprehension.

```
"../src/expts/expt_decrease_in_tour_length_with_larger_number_of_drones.py" 91a≡
    import sys, os
    import numpy as np
    import matplotlib.pyplot as plt
    from matplotlib import rc
    from colorama import Fore, Back, Style
    import argparse
    sys.path.append('../lib')
    import problem_one_horse_multiple_flies as ohmf
    import utils_graphics
    import utils_algo
    import scipy

    def expt(number_of_sites, scheme, inithorseposn, phi):
        ⟨Set up a run data-struture for one horse multiple flies 91b⟩
        ⟨Generate a point set of size N 91c⟩
        ⟨Set up an list of ks which we will loop through and measure the corresponding tour-length 91d⟩
        ⟨Run the selected algorithms for the various ks and generate plots of the tours 92a⟩
        ⟨Plot lengths of tours against the number of drones 92b⟩

    if __name__ == "__main__":
        expt(number_of_sites=10, scheme='uniform', inithorseposn=(0.2,0.5), phi=3)
    ◇
```

7.9.6 This just sets up an empty data-structure whose members will be populated in the succeeding chunks. The `MultipleFliesInput` data-structure stores the list of sites and the initial horse position.

⟨*Set up a run data-struture for one horse multiple flies* 91b⟩ ≡
```
    run = ohmf.MultipleFliesInput()
    ◇
```
Fragment referenced in 91a.

7.9.7 It makes sense to run an experiment against both uniform and non-uniform sets of sites.

⟨*Generate a point set of size N* 91c⟩ ≡
```
    if scheme == 'nonuniform':
        run.sites = utils_algo.bunch_of_non_uniform_random_points(number_of_sites)
    elif scheme == 'uniform' :
        run.sites = scipy.rand(number_of_sites,2).tolist()
    else:
        print "scheme not recognized"
        sys.exit()

    run.inithorseposn = inithorseposn
    print run.sites
    ◇
```
Fragment referenced in 91a.

7.9.8 It will be sufficient to test the decrease in the tour-length for values of ks upto a small number say 20. Maybe for going to bigger and bigger data-sets it will make sense to increase the number of *k*s.

⟨*Set up an list of ks which we will loop through and measure the corresponding tour-length* 91d⟩ ≡
```
    #ks = range(1,20,2)
    ks = [2,4]
    ◇
```
Fragment referenced in 91a.

7.9.9 This is the main loop for genereating data. For different numbers of drones, we get the corresponding tour-lengths. These will be plotted in a succeeding chunk.

⟨ *Run the selected algorithms for the various ks and generate plots of the tours* 92a ⟩ ≡
```
tour_lengths = []
for k in ks:
    tour = run.getTour(ohmf.algo_greedy_earliest_capture, phi, number_of_flies=k)
    print tour
    #tour_lengths.append(pass)
```
◇
Fragment referenced in 91a.
Uses: getTour 74c.

7.9.10

⟨ *Plot lengths of tours against the number of drones* 92b ⟩ ≡
```
fig, ax = plt.subplots()
```
◇
Fragment referenced in 91a.

Chapter Index of Fragments

Chapter Index of Identifiers

Appendices

Appendix A

Index of Files

"../main.py" Defined by 12.
"../src/expts/expt_decrease_in_tour_length_with_larger_number_of_drones.py" Defined by 91a.
"../src/lib/problem_classic_horsefly.py" Defined by 20a.
"../src/lib/problem_one_horse_multiple_flies.py" Defined by 69.
"../src/lib/utils_algo.py" Defined by 17ab, 18abcdef, 19c.
"../src/lib/utils_graphics.py" Defined by 14, 15abc, 16d.

Appendix B

Man-page for `main.py`

Bucketlist of TODOS