Question 1.1

Briefly explain what GET and POST mean in HTTP? What are the other request methods in HTTP version 1.1?

The `GET` method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

The `POST` method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line.

`OPTIONS`, `HEAD`, `PUT`, `DELETE`, `TRACE` and `CONNECT`.

Reference: `http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`

Question 1.2

Describe what the program `simple_webserver.py` does couple of sentences.

1. main function establishes an HTTP server at localhost:80 listening at the HTTP socket
2. `do_GET()` and `do_POST()` determines how to response `GET()` and `POST()` request method respectively

Question 1.3

What does the code 200 in `self.send_response(200)` in that program mean? Name another very commonly used HTTP response code and explain its meaning.

200 OK
The request has succeeded.

The information returned with the response is dependent on the method used in the request, for example:
`GET` an entity corresponding to the requested resource is sent in the response;
`POST` an entity describing or containing the result of the action.

404 Not Found
The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

Reference: `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`

Question 1.4

Based on your observations in Step 1.2, explain briefly what the function `do_Get()` in `simple_webserver.py` does.

`info_messgage()` function analyses client values and server values as well as headers.

CLIENT VALUES
client address, command, path, real path, query, request version

SERVER VALUES
server version, sys version (i.e. python version), protocol version

HEADERS
accept, accept-encoding, accept-language, cache-control, connection, host, upgrade-insecure-requests and user-agent

`do_GET()` function primarily generates a simple HTML and displays the data retrieved by `info_messgage()`.

Question 1.5

Based on your observations in Step 1.3, explain briefly what the function `do_Post()` in `simple_webserver.py` does.

`do_POST()` displays client address, user-agent, path ('/post_form' in this case) and the form data 'FirstName', 'LastName' and their values ('Mickey' and 'Mouse' by default) submitted in `formexample.html`

Question 1.6

Comment on the differences between the client we wrote and a browser, based on your observations. Which one is suited for which role(s)?

CLIENT VALUES
client address, path, real path and query are apparently different.

SERVER VALUES
the same.

HEADERS
accept, accept-encoding and user-agent are different. Request generated by client does not have 'accept-language' and 'upgrade-insecure-requests' fields.

In my opinion, modern browsers such as Chrome, Firefox and Microsoft Edge are rich and sophisticated clients specialized in web pages demonstration and human computer interaction. For example, the layout engine built in web browser renders HTML and CSS to an interactive document and JavaScript interpreter enables browser-end programming. Images, audios and videos are also supported by browsers.

However, clients are thin, light-weight and customized to realize certain logic. Unlike browsers, applications in clients can also invoke underlying protocols not limited to HTTP and HTTPS.

As a result, clients are suited for machines and browsers are suited for humans.

Question 2.1

Demand management falls under the umbrella of smart grid. Why does the grid become smart when there is demand response? What makes the legacy grid not so smart and which new feature(s) changes this?

The grid becomes smart when there is demand response because electricity users (e.g. households, business, industry) can adjust their power load when the total demand in the whole grid changes. The grid benefits from this mechanism by reducing surges and smoothing fluctuations in electricity demand.

Big industrial sites have made agreements with power companies to reduce their demand at peak times. However, this service is infeasible to be expanded to individual clients due to the high cost of human intervention. Hence, **demand management automation** is necessary in smart grid.

Question 2.2

Discuss very briefly the similarities and differences between the transitions from the old telephony system to Internet and the legacy power network to the modern power grid.

Old telephony system commonly employs analog signal to transmit voice over copper loops with limited bandwidth and susceptible to distortion. Modern Internet has larger capacity and is born with error detection and correction mechanism. In fact, the old telephony system is fiercely challenged by Voice over IP technology. In spite of the advent of the smart gird, the main task of grid is to carry electricity. Hence, I think the legacy power grid will be attributed new feature such as demand management automation. Perhaps in the future, the smart grid will be a amalgamation of legacy power grid and Internet. For example, broadband over power lines is a method of power line communication that allows relatively high-speed digital data transmission over the public electric power distribution wiring.

Question 3.1

Provide a working copy of the program you write according to the guidelines for full credit. Note that this is an open-ended question. Feel free to use your creativity!

Price simulation

We want to simulate price change over 24h based on 'avg_pricelist' returned by `import_pricedata()` function. Noticing that data in `GRAPH_5VIC1.csv` starts from 12/07/2014 12:35, we decide to **circularly shift** the python list in order to match price and the real time.

For instance, the 30min average price out of six points (12/07/2014 12:35, 12:40, 12:45, 12:50, 12:55) is used to simulate the real price in the time period from 12:30:01 to 13:00:00, the 30min average price out of six points (13/07/2014 0:05, 0:10, 0:15, 0:20, 0:25) is used to simulate the real price in the time period from 00:00:01 to 00:30:00. More examples are available in the following table.
This arithmetic is implemented by `get_price()` function in `webserver.py` and `import_pricedata()` function in `pricetempreader.py`.

Temperature measurement simulation

We want to simulate temperature change over 24h based on 'temperature' returned by `import_tempdata()` function. Noticing that data in `IDV60901.94868.json` was sorted by their time in descending order (latest

entries listed first), we modify `import_tempdata()` function slightly, i.e. select data with 'sort_order' from 28 to 75 and output in reverse order.
This arithmetic is implemented by `get_temperature()` function in `webclient.py` and `import_tempdata()` function in `pricetempreader.py`.

Simulation result

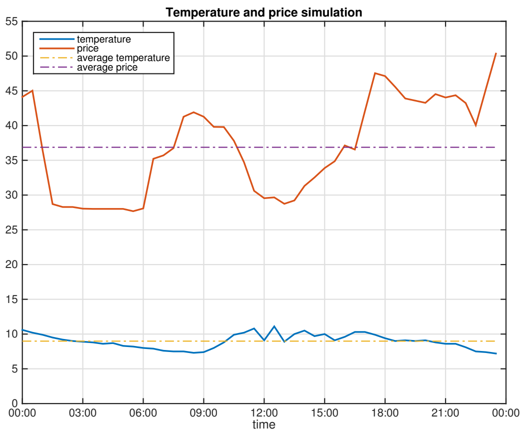

Figure 1: Price and temperature over 24h

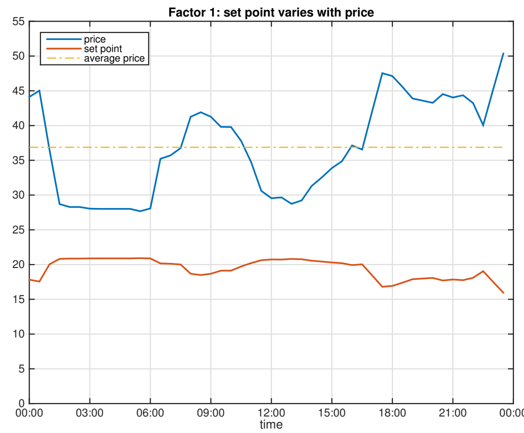

Figure 2: Set point varies with price

Set point decision strategy

Outside temperature influence

As can be clearly seen in Figure 1, temperature fluctuates around 8.98 degrees centigrade. This result is easy to understand, because 12th of July in Melbourne is during winter. Room temperature of 20 degrees is most comfortable for human, hence the house needs heating. The outside temperature will not be taken in to consideration when deciding set point.

avg_pricelist index	time in csv	real time
0	2014-07-12 12:35∼13:00	12:30:01∼13:00:00
1	2014-07-12 13:05∼13:30	13:00:01∼13:30:00
2	2014-07-12 13:35∼14:00	13:30:01∼14:00:00
…	…	…
22	07-12 23:35∼07-13 0:00	23:30:01∼00:00:00
23	2014-07-13 0:05∼0:30	00:00:01∼00:30:00
24	2014-07-13 0:35∼1:00	00:30:01∼01:00:00
…	…	…
46	2014-07-13 11:35∼12:00	11:30:01∼12:00:00

sort_order	local time in json	real time
75	2014-07-12 00:00:00	00:00:00∼00:29:59
74	2014-07-12 00:30:00	00:30:00∼00:59:59
…	…	…
29	2014-07-12 23:00:00	23:00:00∼23:29:59
28	2014-07-12 23:30:00	23:30:00∼23:59:59

Factor 1: price

In order to save total expanse, we decide to tune down the set point when price is higher than the average price (AU$ 36.87) and slightly turn it up and try to store some thermal energy when the price is low.

$$\text{set point} = \begin{cases} 20 - 0.3 \times (\text{price} - \text{average price}) & \text{price} > \text{average price} \\ 20 + 0.1 \times (\text{average price} - \text{price}) & \text{price} < \text{average price} \end{cases} \tag{1}$$

How price impacts on set point is shown in in Figure 2.

Factor 2: daily routine

It is believed that most people are sleeping in bed from 00:00 to 06:00. It is not necessary to keep the room warm during this period. We decide to gradually cool down and then warm up the house during these 6 hours. How daily routine impacts on set point is shown in in Figure 3. Sharp transition is avoided.

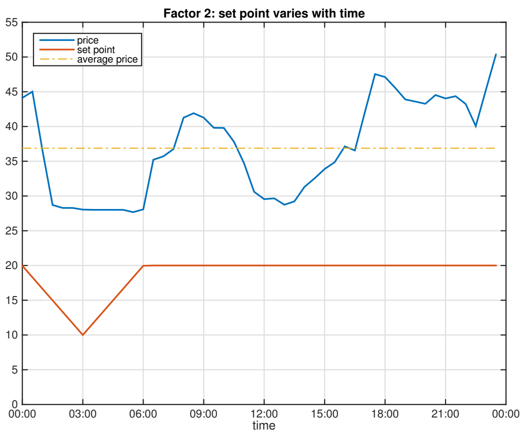

Figure 3: Set point varies with price

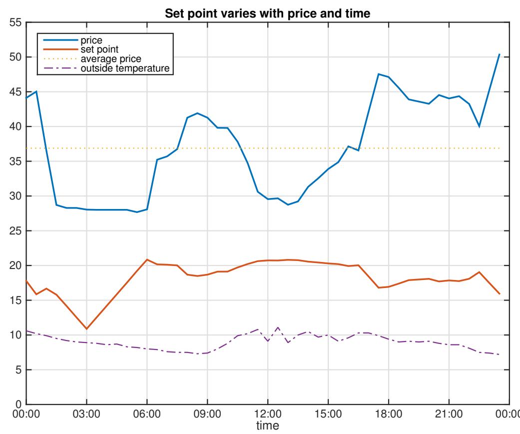

Figure 4: Set point varies with price and time

Comprehensive influences

Comprehensive influences are demonstrated in Figure 4. Apparently, the set point changes from 10.88 to 20.83 degrees.

Client information collection

For demonstration purpose, we write client information such as **user name**, **temperature**, **set point** and current **timestamp** into a `.csv` file. In practical implementation, relational database management system (RDBMS) such as MySQL will be preferable.

`client-information.csv`

user name, temperature, set point, timestamp

```
angli,8.6,17.75,05 Oct 2015 21:47:41
angli,8.6,17.75,05 Oct 2015 21:47:44
qingyun,8.6,17.75,05 Oct 2015 21:47:45
angli,8.6,17.75,05 Oct 2015 21:47:47
qingyun,8.6,17.75,05 Oct 2015 21:47:48
angli,8.6,17.75,05 Oct 2015 21:47:50
```

```
qingyun,8.6,17.75,05 Oct 2015 21:47:51
angli,8.6,17.75,05 Oct 2015 21:47:53
qingyun,8.6,17.75,05 Oct 2015 21:47:54
angli,8.6,17.75,05 Oct 2015 21:47:56
qingyun,8.6,17.75,05 Oct 2015 21:47:57
angli,8.6,17.75,05 Oct 2015 21:47:59
qingyun,8.6,17.75,05 Oct 2015 21:48:00
angli,8.6,17.75,05 Oct 2015 21:48:02
qingyun,8.6,17.75,05 Oct 2015 21:48:03
```

Appendix

simulation_and_design.m

```
clear;
close all;

load data.mat;
% temperature and price

hour = 0:47;
hour = floor(hour/2);

minute = zeros(1, 48);
minute(2:2:48) = 30;

sdate = datenum(2015, 10, 5, hour, minute, 0);

temperature_average = mean(temperature);
price_average = mean(price);

set_point1 = ones(1, 48) * 20;
for index=1:48
    if price(index) > price_average
        set_point1(index) = set_point1(index) - 0.3 * (price(index) - price_average);
    else
        set_point1(index) = set_point1(index) - 0.1 * (price(index) - price_average);
    end
end

set_point2 = ones(1, 48) * 20;
for index=1:48
    if index<=13
        set_point2(index) = set_point2(index) - (10 - 1.66 * abs(index - 7));
    end
end

set_point = ones(1, 48) * 20;
for index=1:48
    if price(index) > price_average
        set_point(index) = set_point(index) - 0.3 * (price(index) - price_average);
    end
    if price(index) < price_average
        set_point(index) = set_point(index) - 0.1 * (price(index) - price_average);
    end
    if index<=13
        set_point(index) = set_point(index) - (10 - 1.66 * abs(index - 7));
    end
end

plot(sdate, temperature, sdate, price, 'linewidth', 1.5);
hold on;
plot([sdate(1) sdate(48)], [temperature_average temperature_average], '-.');
plot([sdate(1) sdate(48)], [price_average price_average], '-.');
datetick('x', 'HH:MM');
legend('temperature', 'price', 'average temperature', 'average price', 'location', 'northwest');
title('Temperature and price simulation');
xlabel('time');
ylim([0 55]);
grid on;
```

```
figure;
plot(sdate, price, sdate, set_point1, 'linewidth', 1.5);
hold on;
plot([sdate(1) sdate(48)], [price_average price_average], '-.');
datetick('x', 'HH:MM');
title('Factor 1: set point varies with price');
xlabel('time');
legend('price', 'set point', 'average price', 'location', 'northwest');
ylim([0 55]);
grid on;

figure;
plot(sdate, price, sdate, set_point2, 'linewidth', 1.5);
hold on;
plot([sdate(1) sdate(48)], [price_average price_average], '-.');
datetick('x', 'HH:MM');
title('Factor 2: set point varies with time');
xlabel('time');
legend('price', 'set point', 'average price', 'location', 'northwest');
ylim([0 55]);
grid on;

figure;
plot(sdate, price, sdate, set_point, 'linewidth', 1.5);
hold on;
plot([sdate(1) sdate(48)], [price_average price_average], ':');
plot(sdate, temperature, '-.');
datetick('x', 'HH:MM');
title('Set point varies with price and time');
xlabel('time');
legend('price', 'set point', 'average price', 'outside temperature', 'location', 'northwest');
ylim([0 55]);
grid on;
```

webserver.py

```python
# -*- coding: utf-8 -*-

import time
import csv
import cgi
import urlparse
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
# import helper data reading functions

from pricetempreader import import_pricedata
from pricetempreader import get_time_index

import json

# Main parameters
HOST_NAME = 'localhost'
PORT_NUMBER = 8080

class MyHandler(BaseHTTPRequestHandler):
    ''' HTTP request handler class extending BaseHTTPRequestHandler '''

    def myparse_getrequest(self):
        ''' GET request: parse the path and extract query '''
        query_string = urlparse.urlparse(self.path).query
        querydict=urlparse.parse_qs(query_string)

        # return path components in a list (ordered)
        # and query variables&values in a dictionary (unordered)
        return querydict

    def myparse_postrequest(self):
        ''' POST request: parse the form data posted '''
        form = cgi.FieldStorage(
            fp = self.rfile,
            headers = self.headers,
            environ = {
                'REQUEST_METHOD': 'POST',
                'CONTENT_TYPE': self.headers['Content-Type'],
            }
        )
        postdict = {}
        for field in form.keys():
            postdict[field] = form.getvalue(field)

        return postdict

    # respond to a GET request
    def do_GET(self):
        ''' responds to a GET request '''

        #send response
        self.send_response(200)
        self.end_headers()

        # querydict=self.myparse_getrequest()

        # replace this part with application logic ----------------
        # send back parsed request content for debugging
        # self.wfile.write(querydict)
        ####------------------------------------------------------
```

```
        return

    # respond to a POST request
    def do_POST(self):

        # Begin the response
        self.send_response(200)
        self.end_headers()

        postdict=self.myparse_postrequest()

        username = postdict['username']
        temperature = postdict['temperature']
        set_point = float(postdict['set_point'])
        localtime = postdict['localtime']

        if username and temperature and set_point and localtime :
            writer = csv.writer(file('client-information.csv', 'a+'))
            row = [username, temperature, set_point, localtime]
            writer.writerow(row)

        price = get_price()

        data = json.dumps({"price": price})

        # replace this part with application logic ----------------
        # send back parsed post content for debugging
        self.wfile.write(data)
        ####-------------------------------------------------------

        return

def get_price():
    time_index = get_time_index()
    # divide 24*60 minutes into 48 slots (30 min / slot)
    # get_time_index() is defined in 'pricetempreader.py'

    # We want to simulate price change over 24h based on 'avg_pricelist' returned by
        import_pricedata() function
    # In 'GRAPH_5VIC1.csv', data start from 12:35.

    # csv   csv slot        time_index   real slot
    # 0     12:35~13:00     25           12:30:01~13:00:00
    # 1     13:05~13:30     26           13:00:01~13:30:00
    # 2     13:35~14:00     27           13:30:01~14:00:00
    # ...   ...
    # 22    23:35~0:00      47           23:30:01~00:00:00
    # 23    0:05~0:30       0            00:00:01~00:30:00
    # 24    0:35~1:00       1            00:30:01~01:00:00
    # ...   ...             ...
    # 46    11:35~12:00     23           11:30:01~12:00:00
    # 47    12:05~12:30     24           12:00:01~12:30:00

    # for instance
    # the price in 12:30:01~13:00:00 corresponds to avg_pricelist[0]
    # the price in 12:00:01~12:30:00 corresponds to avg_pricelist[47]

    index = (time_index + 23) % 48

    avg_pricelist = import_pricedata()
```

```
    price = avg_pricelist[index]

    price = round(price, 2)     # round and keep 2 decimals

    return price


################################################
##              Main

httpd = HTTPServer((HOST_NAME, PORT_NUMBER), MyHandler)
print time.asctime(), "Server Starts - %s:%s" % (HOST_NAME, PORT_NUMBER)
try:
    httpd.serve_forever()
except KeyboardInterrupt:
    pass
httpd.server_close()
print time.asctime(), "Server Stops - %s:%s" % (HOST_NAME, PORT_NUMBER)
```

webclient.py

```python
# -*- coding: utf-8 -*-

import requests

from pricetempreader import import_tempdata
from pricetempreader import get_time_index

import time

USERNAME = 'angli'
# username

def get_temperature(time_index):

    temperature = import_tempdata()
    # import_tempdata() has been slightly modified

    temperature = temperature[time_index]

    return temperature

def get_set_point(time_index, price):
    #  arithmetic within get_set_point() function is explained in detail in the workshop report.

    set_point = 20
    price_average = 36.87

    if price > price_average :
        set_point = set_point - 0.3 * (price - price_average)
    else:
        set_point = set_point - 0.1 * (price - price_average)

    if time_index <= 12 :
        set_point = set_point - (10 - 1.66 * abs(index - 6))

    set_point = round(set_point, 2)

    return set_point


#################################################
##              Main

set_point = 0
# initial set point

print USERNAME
# print username

while True:
    time_index = get_time_index()
    # divide 24*60 minutes into 48 slots (30 min / slot)
    # get_time_index() is defined in 'pricetempreader.py'

    temperature = get_temperature(time_index)

    localtime = time.strftime('%d %b %Y %X', time.localtime(time.time()))
    # local time represented in string like '04 Oct 2015 19:51:42'

    payload = {'username': USERNAME, 'temperature': temperature, 'set_point': set_point, '
```

```
        localtime': localtime}
    # parameters posted to server

    r = requests.post("http://localhost:8080/client_api", data=payload)
    # post data and receive response
    r = r.json()
    # json decode
    price = r['price']

    set_point = get_set_point(time_index, price)
    #  arithmetic within get_set_point() function is explained in detail in the workshop report.

    print 'Temperature: ', temperature
    print 'Price: ', price
    print 'Set point: ', set_point

    time.sleep(3)
```

pricetempreader.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Jul 13 13:42:46 2014

@author: alpcan
"""

import csv
import numpy as np
import json

import datetime

def import_pricedata():
    ''' Imports wholesale electricity price from the AEMO file
        'GRAPH_5VIC1.csv' which should be in the same folder.
        returns the 30 min average prices as an array.

        This array should be aligned with the temperature data!
    '''

    filename='GRAPH_5VIC1.csv'
    prices=[]
    with open(filename,'rb') as f:
        content=csv.reader(f)
        content.next() # skip first row
        for row in content:
            prices.append(float(row[3])) # retail price

    # convert to numpy array
    pricearray=np.array(prices)

    # calculate moving average of prices
    # to get 30mins out of 5 min data
    avg_prices=np.zeros(48)
    for i in range(48):
        avg_prices[i]=np.mean(pricearray[i*6:(i+1)*6])

    # back to list from numpy array for convenience
    avg_pricelist=avg_prices.tolist()

    return avg_pricelist


# imports RRP data from AEMO file
def import_tempdata():
    ''' Imports air temperature data from BOM file for Melbourne
        'IDV60901.94868.json' which should be in the same folder.
        returns the 30 min temperatures as an array.

        This array should be aligned with the AEMO price data!
    '''

    filename='IDV60901.94868.json'
    with open(filename,'rb') as f:
        content=json.load(f)

    # only interested in air temp
    subset=content['observations']['data']
    tempset=[item['air_temp'] for item in subset]
```

```
    # We want to simulate temperature change over 24h based on the data stored in 'IDV60901
        .94868.json'
    # We select the dataset whose 'sort_oder' is in the range from 75 to 28

    # sort order     local time              real time
    # 75             2014-07-12 00:00:00     00:00:00~00:29:59
    # 74             2014-07-12 00:30:00     00:30:00~00:59:59
    # ...            ...
    # 29             2014-07-12 23:00:00     23:00:00~23:29:59
    # 28             2014-07-12 23:30:00     23:30:00~23:59:59

    # sort_order=28:75 AEMO price data
    temperature=tempset[28:75+1]

    # output in reverse order
    temperature=temperature[::-1]

    # Eventually, we get 48 temperatures on 12th July, 2014 which are enough to simulate the
        temperature change over 24h.

    return temperature


# divide 24*60 minutes into 48 slots (30 min / slot)
def get_time_index():
    now = datetime.datetime.now()
    # the time stamp of now

    midnight = now.replace(hour=0, minute=0, second=0, microsecond=0)
    # the time stamp of 00:00:00 on the same day

    minutes = (now - midnight).seconds / 60
    # how many minutes since midnight

    time_index = minutes / 30
    # divide 24*60 minutes into 48 slots (30 min / slot)

    return time_index


def print_price_list():
    avg_pricelist = import_pricedata()

    price_list = [0] * 48

    for i in range(48):
        index = (i + 23) % 48
        price_list[i] = avg_pricelist[index]
    print price_list
```