# AMATH 301 – Spring 2022
# Homework 4

Due: 11:59pm, February 3, 2023.

**Instructions for submitting:**

- Coding problems are submitted to Gradescope as a python script (`.py` file) or a Jupyter Notebook (`.ipynb` file) (warning, this may not work as well as a python script). You have **8** attempts (separate submissions to Gradescope) for the coding problems.
- Writeup problems are submitted to Gradescope as a single `.pdf` file that contains text and plots. Put the problems in order and label each writeup problem. When you submit, **you must indicate which problem is which on Gradescope**. Failure to identify pages on Gradescope will lead to a **5% grading penalty. All code you used for this part of the assignment should be included at the end of your .pdf file. Failure to do so results in a 25% penalty.**

## Coding problems

**Note that the coding portion is shorter than the writeup portion of the assignment this week. Please plan accordingly.** The coding portion is worth 10 points, with each answer worth 1 point.

1. A seminal breakthrough in HIV treatment was the use of multiple-drug combination therapy. The idea behind the therapy is that multiple drugs, introduced to the body at specific times, fight the virus off better than one drug alone. A key step in the development of multiple-drug therapy is determining the best time to take the next drug. One way this is done is to take the next drug when the previous drug is at its maximum in the body (and beginning to decrease from there). Drug absorption and elimination in the body is modeled using *compartmental models*.

   Consider a model whose solution is given by

   $$x(t) = \frac{11}{6}\left(e^{-t/12} - e^{-t}\right), \tag{1}$$

   where $x(t)$ is the amount of the drug in the body at time $t$. Assume that we want to administer the next drug when $x(t)$ is at its maximum. We call this time $t_{\max}$ in what follows. Recall that we have only talked about **minimizing** functions so you will have to restate the problem as a minimization problem.

(a) Find $t_{\max}$ and the maximum of the function $x(t)$ by taking the derivative (by hand) and using `scipy.optimize.fsolve` to solve the resulting equation (see an example of how to use this in the Homework 4 template), with an initial guess of $x_0 = 1.5$, to find the root of $x'(t)$. First save $x'(1.5)$ to the variable `A1` to make sure you have $x'(t)$ defined correctly. Then save $t_{\max}$ to the variable `A2` and the maximum of $x(t)$ to the variable `A3`.

**Note:** `scipy.optimize.fsolve` returns an array, make sure you get the single number out of that array for your answers here.

(b) Use `scipy.optimize.fminbound` with the interval $[0, 10]$ to find $t_{\max}$ and the maximum of $x(t)$. Create an array with 2 elements: $t_{\max}$ in the first component and the maximum in the second component. Save this array to the variable `A4`.

2. The function

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

is known as *Himmelblau's function* which is a function designed for testing performance of optimization algorithms (you can see more examples here). Writeup Problem 1 is about visualizing this function, so if you are having trouble seeing what's happening you could work on visualizing it. Himmelblau's function has multiple *local* minima. In this problem we will find one of those local minima.

(a) Create an anonymous function that computes $f$ using only one input, an array with two elements: $\begin{bmatrix} x & y \end{bmatrix}^T$. To make sure that your function is defined correctly, compute $f(3, 4)$ (using the array $[3, 4]$) and save the answer to the variable `A5`.

(b) Use `scipy.optimize.fmin` with an initial guess of $\begin{bmatrix} -3 & -2 \end{bmatrix}^T$ to find the argmin of $f$ (the array that minimizes $f$) and save it to the variable `A6`.

(c) Create an anonymous function (with only one input) that calculates the gradient $\nabla f(x, y)$. The gradient is given by the following formula

$$\nabla f(x, y) = \begin{pmatrix} 4x^3 - 42x + 4xy + 2y^2 - 14 \\ 4y^3 - 26y + 4xy + 2x^2 - 22 \end{pmatrix}. \tag{2}$$

Recall that the gradient of $f$ should be zero at a local minimum. Calculate $\nabla f(x, y)$ at the $x$ and $y$ values found in part (b) and save the result to the variable `A7`. Then save the 2-norm of `A7` to the variable `A8`. In other words, if you found $(x, y) = (x_1, y_1)$ above, save

$$\|\nabla f(x_1, y_1)\|_2$$

to the variable `A8`. The 2-norm is calculated in python via `np.linalg.norm`.

(d) Recall from class that we wrote the following code to perform one step of gradient descent (given on next page).

```
p = np.array([6, 4]) # Choose an initial guess
grad = gradf(p) # Find which direction to go
phi = lambda t: p - t*grad # Define the path
f_of_phi = lambda t: f(phi(t)) # Create a function of
                                # "heights along path"
tmin = scipy.optimize.fminbound(f_of_phi,0,1) # Find time it takes
                                                # to reach min height
p = phi(tmin); # Find the point on the path and update your guess
```

Adapt the code so that it performs up to 2000 iterations of gradient descent and stops if the 2-norm of the vector `grad` is less than a tolerance of $10^{-7}$.

(e) Now use your gradient descent code to find the argmin of the function $f(x, y)$. Use an initial guess of $(x, y) = (-3, -2)$. Save the final result to the variable `A9` and the number of iterations to the variable `A10`. The initial guess does not count as an iteration.

# Writeup problems

The writeup portion is worth 10 points, each question is worth 5 points.

1. Recall Himmelblau's function defined in Coding Problem 2. In this problem you will create plots of this function to aid in visualization. **Turn in the plots and the code used to make the plots.**

   (a) First we will create a surface plot using `plot_surface`.

   i. Define the anonymous function $f(x, y)$ as a function of two variables.

   ii. Use the `meshgrid` function to create a mesh with 40 equally spaced points from $-7$ to 7 in the $x$ direction and 40 equally spaced points from $-7$ to 7 in the y direction.

   iii. Use `plot_surface` to plot $f(x, y)$ versus $x$ and $y$.

   iv. Notice that the plot does not look good with default settings. You should

   - Change the coloring of the plot by adding the option `cmap = 'viridis'` (or if you have another colormap you prefer, you can use that) to the `plot_surface` command.
   - Add a *color bar* that indicates what values the colors correspond to.
   - Set the viewing angle of the plot using `ax.view_init(30,30)`.
   - Label your $x$, $y$, and $f(x, y)$ axes.
   - Add a descriptive title to your plot.

   (b) Next you will create a *contour plot* of the function. This plot should not be in the same figure as above: it should be on its own.

   i. Define a meshgrid with 100 linearly spaced points between $-7$ and 7 along the $x$-axis and 100 linearly spaced points between $-7$ and 7 along the $y$-axis.

ii. Use the `contour` command to plot the contours of $f(x, y)$ vs $x$ and $y$. Notice that the contours are not very helpful here so we are going to change the scale to make them look better. Use `contour(..., np.logspace(-1, 3, 22))` where the `...` is filled in with what you would usually put in `contourf`. This makes the contours logarithmically spaced, making it easier to see the minima.

iii. Change the colormap to `'viridis'` as was done in part (a).

iv. Label your $x$ and $y$ axes.

v. Add a colorbar.

vi. Add a descriptive title to your plot. You may want to title the plot after adding in the stuff in part (c).

(c) Next you will use `scipy.optimize.fmin` repeatedly to find all of the local minima (the multiple local minimums) of $f$. Use the plots you have created and appropriate initial guesses (based on what you see in the plots you have created) to find all 4 local minima. Plot each of the 4 local minima you found with yellow stars (`'y*'`) on top of the contourplot you created in part (b). You should use a markersize of 15 to make the markers visible.

(d) Now we will add visual markers to the contourplot figure to show where the minima are, according to the gradient-descent algorithm used in the coding section of the homework. **You should not copy and paste the results that your code returns**. Instead, copy and paste your **code** (not the values that your code gives) and rerun it here, or save your data from that file and load it in here using the `np.savetxt` and `np.genfromtxt` commands. Plot the local minimum you found with gradient descent in Coding Problem 2 (e) as a green square. Set the markersize to 5. Create a legend entry for this point with the label "Grad descent."

2. This problem is about comparing the two different ways to do gradient descent. You should re-familiarize yourself with Coding Problem 2 and the details therein before moving on. For both methods, you will record the time it takes to complete the algorithm, the number of steps it takes to complete the algorithm, and whether or not the algorithm *converged.* For the purposes of this problem, we will say that **the method converged if it stops iterating before reaching the maximum number of steps we have provided.** In other words, if the stopping condition was ever satisfied, the method converged.

   Instead of calculating `tmin` with `fminbound` for every step of gradient descent, you can use a fixed value of `t`, which we will call `tstep`, to calculate the next guess in gradient descent. You do this by replacing the lines of code

```
phi = lambda t: p - t*grad # Define the path
f_of_phi = lambda t: f(phi(t)) # Create a function of "heights along path"
tmin = scipy.optimize.fminbound(f_of_phi,0,1) # Find the time it takes
                                              # to reach min height
p = phi(tmin) # Find the point on the path and update your guess
```

with the single line of code

```
p = p - tstep*grad;
```

where the variable `tstep` must be defined previously in the code.

In each of the following solves using gradient descent, use an initial guess of $\begin{bmatrix}2 & 3\end{bmatrix}^T$, a tolerance **on the norm of the gradient** of $10^{-9}$ and a maximum of 8000 iterations to try to find the vector/array $\begin{bmatrix}x & y\end{bmatrix}^T$ that minimizes the function $f(x, y)$.

**You do not need to answer these questions individually, you only need to turn in the table created in part (e) and the comments in part (f), along with the code you used for this problem.**

(a) Use gradient descent with `fminbound` (as you did on Coding Problem 2, **except now use a tolerance of** $10^{-9}$). Time how long it takes for your algorithm to run to completion using `time.time`. Further, save the number of iterations required to complete the algorithm.

(b) Use gradient descent with `tstep = 0.01` (described above). As in (a), time how long it takes to complete the algorithm and record the number of steps.

(c) Repeat part (b) with `tstep = 0.02`.

(d) Repeat part (b) with `tstep = 0.025`.

(e) Record your results in the following table

| | Number Iterations | Time | Converged (Yes/No) |
|---|---|---|---|
| `tstep=0.01` | | | |
| `tstep=0.02` | | | |
| `tstep=0.025` | | | |
| fminbound | | | |

(f) Comment on the results in the table. Your comments should discuss at least the following things.

- Did the Gradient Descent method always converge? For which settings/step sizes (using fminbound vs. a fixed time step) did it not converge? In the cases for which it did not converge, explain why it did not converge using what we discussed/showed in class.
- For which settings (using fminbound vs. a fixed time step and different step sizes) did the method converge the quickest?
- Which converged in the fewest number of iterations?
- Is your answer to the above two questions (fastest and fewest iterations) the same? If so, why? If not, why? What slows down the algorithm?
- **Optional:** Can you find the optimum `tstep` to make this algorithm converge the fastest for this problem? Hint: you can write a function that takes in `tstep` as a variable, outputs the time, and then minimize that function!