Lab 1

2D Plotting

The first three labs will help you become familiar with plotting and animations in Python. Later in the semester you will have labs that will ask you to plot and/or animate certain functions and it may be helpful to reference these labs.

Download the ACME Introduction to Matplotlib document from

acme.byu.edu/wp-content/uploads/2019/08/MatplotlibIntro19.pdf

Read the first 9 pages of the document and write/execute every piece of sample code from the yellow boxes in your `Sandbox` file. You can skip Problem 1, but you should do Problems 2, 3, 4, and 5 and turn these in. Stop reading at Visualizing 3D Surfaces.

Lab 2

3D Plotting

In the ACME Introduction to Matplotlib document

acme.byu.edu/wp-content/uploads/2019/08/MatplotlibIntro19.pdf

start reading on page 9 at Visualizing 3D Surfaces. Read through page 11 and write/execute every piece of sample code from the yellow boxes in your `Sandbox` file. Do Problem 6 and turn it in.

Now download the ACME 3D Plotting and Animations document from

www.acme.byu.edu/wp-content/uploads/2018/09/Animation.pdf

Do Problems 2 and 4 and turn these in. It will be helpful to read in the document a bit before each of these problems to help you do them (but feel free to ignore anything about animations until next week).

Lab 3

Animating Plots

In the ACME 3D Plotting and Animations document

www.acme.byu.edu/wp-content/uploads/2018/09/Animation.pdf

read everything about animation and write/execute every piece of sample code from the yellow boxes in your `Sandbox` file. Do Problems 1, 3, and 5 and turn these in.

Lab 4

The Zen of Python

In this lab you will learn how to execute Python code from the command line instead of using Colab. In the ACME Python Essentials document

(see foundations-of-applied-mathematics.github.io/)

follow the instructions in Section 1, Getting Started, to install Python 3.7 via Anaconda on your machine. Read Running Python and do Problem 1. (If your laptop is running Windows, it may be easier to do this entire lab on one of the department Linux machines.) Then read about IPython and write/execute all of the yellow code boxes in your `Sandbox` file. Now do the following problems.

> Using your code from Fall Semester Lab 20, Cracking the Vigenére Cipher, create a file `vigenere.py` that decodes an encoded message in a `.txt` file (see below for instructions on passing command line arguments to a Python program.) Get the text file `ciphertext.txt` from Learning Suite. Your program should print (not return) both the key and the message using two separate print statements.
>
> ```
> >>> python vigenere.py ciphertext.txt
> <key>
> <familiar plaintext>
> ```

To pass command line arguments to a Python program, use the `argparse` module. A minimal working example (saved in a python file `argparse-ex.py`):

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("n")
args = parser.parse_args()
print int(args.n)**2
```

When we run the command `python argparse-ex.py 17` in the command line it outputs `289` since $289 = 17^2$. If you are interested in passing more arguments into your Python

program, or other options, see the tutorial

docs.python.org/2/howto/argparse.html

To read the contents of a `.txt` file in Python, use the command

```
ctext = open('ciphertext.txt').read()
```

Read The Zen of Python: legacy.python.org/dev/peps/pep-0020/. Now familiarize yourself with the PEP 8 Style Guide for Python Code www.python.org/dev/peps/pep-0008/ (don't read the whole thing, it's very long). I suggest you read the first 5 sections (through Whitespace in Expressions and Statements) and also Naming Conventions: Function and Variable Names. Now go back to your first few Python Labs from Fall Semester and reflect on how they might be changed to conform to this style guide. Write a few sentences listing some practices you'll try to implement this semester to make your code more readable.

Lab 5

Solving Sudoku, Part I

In this lab we will begin to write a program to solve sudoku puzzles. This week we will write a program that will solve particularly easy sudoku puzzles; next week we will expand it to solve any sudoku.

A sudoku is a 9×9 grid containing digits in some cells, as below.

4			8	7			2	
	8					4		
		6	3			8		1
7			1				8	
6	1	2		9	8	7	3	4
				6			1	9
1	9	3	4	2	7	5		
8		7		1		3		2
	2				3			

The goal of the puzzle is to insert numbers into the empty squares in such a way that each digit from 1 to 9 occurs exactly once in:

- each row,
- each column, and
- each of the nine 3×3-subsquares of the puzzle.

In this lab, we will create some basic functions to solve simple sudoku puzzles. The puzzles that I supply you will consist of a string of 81 digits from 0 to 9, with 0 indicating an empty cell, and each group of nine digits representing a row in the sudoku puzzle. For instance, the sudoku puzzle above would be entered as

400870020080000400006300801700100080612098734000060019193427500807010302020003000

1. To begin, write a function `convert(string)` that converts a string of 81 digits into a one dimensional array of 81 numbers, with the first nine numbers representing the first row of the sudoku, the second nine representing the second row, etc.

2. Create a function `entry(a,b,grid)` that takes an array `grid` of the type you returned in problem 1, and returns the (a, b) entry of the Sudoku grid (where the rows and columns of the sudoku grid are labeled with integers from 1 to 9). For example, if `S = convert(string)`, where `string` is the string of length 81 listed above, then

```
>>> grid( 5, 3, S)
2
```

3. Create a function `ispossible(a,b,n,grid)` that returns `True` if n is a possible value for the (a, b) entry of `grid`, and `False` otherwise. To determine if n is a possible value, it should examine all of the nonzero digits in the same column, row, and 3×3 subsquare as (a, b); n is a possible value if and only if it matches none of these values. If the (a, b) entry already has a nonzero value, the function `ispossible` should return an answer as if the (a, b)-entry were zero; in other words it should tell you whether n could be placed in the (a, b)-entry if that entry were empty.

```
>>> ispossible( 5, 4, 5, S)
True
>>> ispossible( 7, 2, 4, S)
False
```

4. Create a function `computepossibilities(a,b,grid)` that returns a list of all possible n for which `ispossible(a,b,n,grid)` is `True`.

```
>>> computepossibilities( 5, 4, S)
[5]
>>> computepossibilities( 9, 3, S)
[4, 5]
```

5. Create a function `insert(a,b,n,grid)` that sets the (a, b) entry of the grid to the number n (where n is a digit from 1 to 9). If the (a, b) entry of the grid is already nonzero, or if n is not a possible value that can fit into the (a, b) entry of the grid, the function should do nothing and return a value FALSE, otherwise it should set the (a, b) entry of the grid to n and return a value of TRUE.

6. Create a function `singletons(grid)` that checks each of the entries of the grid, and if the entry is empty and has only one possible value, fills it in with the only possible value. If there are no such entries, the function should return FALSE, otherwise it should return TRUE.

7. Create a procedure `simplesolve(grid)` that runs `singletons(grid)` over and over until it returns a value of False.

At this point, the function `simplesolve(grid)` should be able to solve very simple sudoku puzzles. In addition to the puzzle above, here is a list of ten sudokus that it should be able to solve.

```
004300209005009001070060043006002087190007400050083000600000105003508690042910300
040100050107003960520008000000000017000906800803050620090060543600080700250097100
600120384008459072000006005000264030070080006940003000310000050089700000502000190
497200000100400005000016098620300040300900000001072600002005870000600004530097061
005910308009403060027500100030000201000820007006007004000080000640150700890000420
100005007380900000600000480820001075040760020069002001005039004000020100000046352
009065430007000800600108020003090002501403960804000100030509007056080000070240090
000000657702400100350006000500020009210300500047109008008760090900502030030018206
503070190000006750047190600400038000950200300000010072000804001300001860086720005
060720908084003001700100065900008000071060000002010034000200706030049800215000090
```

- It may make your debugging easier if you write a function `printgrid(grid)` which prints the sudoku grid in a human readable form (i.e. as a 3×3 grid of 3×3 grids, with appropriate spacing). You do not need to write this function if you don't want to, and you will not be graded on it.
- All access to entries of the grid (either reading them or writing them) should be done via the function `entry(a,b,grid)` and `insert(a,b,n,grid)`. This will make the programs look more uniform, and allow for changing the data structure at a later time, without having to rewrite the program.
- Of the functions described above, the most crucial, and probably the longest, will be `ispossible(a,b,n,grid)`. In particular, determining the entries of the 3×3 subsquare containing the (a, b) entry can be tricky. One possibility is to set `r=(a-1)//3`, `s=(b-1)//3`, and then the 3×3 subsquare will consist of the `(3r+i,3s+j)` entries with $1 \leq i, j < 4$.
- If you have extra time, write a function that looks at each column to see if there is a number that can only be placed in a single spot in that column, and if so, places it there. You can do the same for rows and 3×3 subsquares as well. Add these functions into your `simplesolve(grid)` function; they will speed up the solution to more difficult sudokus that we will do next week.

Lab 6

Symbolic Python

Today we will learn the basics of SymPy (using Python to do algebra on symbolic expressions instead of numerical functions) using the ACME Introduction to SymPy text

www.acme.byu.edu/wp-content/uploads/2017/08/Sympy.pdf

Work through and turn in Problem 1 (writing a symbolic function), problem 3 (simplifying a symbolic function), problem 4 (solving symbolic equations), and problem 6 (differentiating symbolic equations). You will likely find it helpful to read most of the instructions surrounding these problems and execute the yellow-boxed code in your `Sandbox` file. Feel free to ignore the sections on linear algebra (you already know all of this) and multiple integrals (we'll get to this later).

Lab 7

Solving Sudoku, Part II

In this lab we will continue to modify our sudoku solving program from last week to solve *any* sudoku puzzles, not just easy ones. We will need some preliminary functions before writing the main routine.

1. Write a function `issolved(grid)` that checks to see if a sudoku grid is completely filled out (in other words, are all of the entries in the array nonzero). It should return `True` if all entries have been filled in, and `False` otherwise.

2. Write a function `issolvable(grid)` that checks to see if every entry that has not been filled out still has values that could possibly be put into it. In other words, check that for each entry in the grid that is still 0, `computepossibilities(a,b,grid)` returns a nonempty list.

We are now prepared to write a recursive function that will solve *any* solvable sudoku puzzle. Write a function `solve(grid)`, which takes as input a sudoku grid, and returns either a completely solved sudoku grid, or the value `False` if it fails to solve the grid. Here is some pseudocode, with a lengthier explanation below it (read all of this lab before starting to write your program).

```
def solve(grid):

  run simplesolve(grid)

  if grid is solved:
    return grid

  if grid is not solvable:
    return False

  find an entry (a,b) of the grid which is still 0

  possibilities = list of all possible entries for the (a,b) slot
```

```
for each possibility n:

  make a copy of grid, called newgrid
  insert n into the (a,b) entry of newgrid

  if newgrid is solvable:
    newgrid = solve(newgrid)
    if not(newgrid is False):
      return newgrid

return False
```

The following notes clarify some of the steps above.

(a) After running simplesolve and checking if the grid is solvable, we have a grid for which each unfilled entry has multiple possibilities (since if any entry had only one possibility, `simplesolve(grid)` would have filled it in). The function should find some entry of the grid which has not been filled out, and store the coordinates of the entry in the variables `a` and `b`. It should also compute the possible numbers that could fit into the (a, b) entry, and store than in the variable `possibilities`.

Note: If you want your program to run faster, don't just find an arbitrary entry that hasn't been filled out; instead find the one with the smallest number of possibilities. This will reduce the size of the next loop.

(b) We will now loop through the entries of `possibilities`, one at a time. For each entry:

(i) Make a copy of the sudoku grid, which we will call `newgrid` (be sure not to just use the command `newgrid=grid`, since this will only make a new reference to `grid`, not actually a new copy).

(ii) Use the `insert` command (from last week) to set the `(a,b)` entry of `newgrid` to equal the kth entry of `possibilities`.

(iii) Check if `newgrid` is solvable. If so, run the command `newgrid=solve(newgrid)`. Otherwise do nothing.

(iv) After running `newgrid=solve(newgrid)`, either `newgrid` will be a solved sudoku grid, or the value `False`. If `newgrid` is not `False`, then `return newgrid`. If `newgrid` is `False`, there is no need to do anything; this means that this possibility was not the correct entry; it does not lead to a solution.

There are two ways that we can exit this loop; we may have returned a solved sudoku (in which case, any code after the loop will never be reached), or we may have tried each possible value in the (a, b) entry, and found that none of them lead to a solution, in which case the code after the loop will be run.

At this point, the function `solve(grid)` should return a solved sudoku grid for any sudoku puzzle that you put in that has a valid solution. Here are some puzzles for you to try it on. Some of them may take your program a while to solve.

00300090002090406070005000301030508000608030005020907050001000808070301000900040098010000050100000006708900030020640000400080000570800600035026000000030100000204501070208030005000100600070060007000409042306040008000900300090080004000205060704090305040006000801010000000700007009030080900505006000020000000608040003000603080140003000000060080000000001000050090080000600070200000000102700503000040900000000,708000300000201000500000000040000026300080000000100090090600004000070500000000000708000300000601000500000000040000026300080000000100090090200004000070500000000000307040000000000918000000040000070000016000000025000000000038009000050002060000050070060000380000000000020062040000000000009170000000000035080400000100000090000400706000038000000000002006205000000000000917000000000004308050000010000009000004001020000000907001000000000430600800000050000200000705008000000600300900000000705000002000401000300000000010600400200050000000000090000370000080000600090000080000004109003000003000500000480070000000000620100000006002000050700008000000900000705000002000401000300000000010600400200050000000000090000370000090000800080000060080010000005000030000000400000605070890000200000300000200000109006700000000400000809000300000701000500000000070000026300090000000100040060200004000080500000000000708000300000601000400000000060000025300080000000100090090500002000070400000000000

Remarks:

- Unlike `simplesolve(grid)`, the function `solve(grid)` must return a value, not just change the entries of `grid`. This is because many (perhaps most) of the values that it inserts into `grid` will be guesses that are incorrect. If we actually changed the original grid with these guesses, we would have to keep track of them and be able to erase them.
- The logic of the recursive `solve(grid)` command does not actually require the function `simplesolve(grid)` at all (however, it will run faster with `simplesolve`). Try commenting out the `simplesolve(grid)` command at the beginning of the function, and see what happens. (This will work better if you programmed `solve(grid)` to look for the unfilled entry with the fewest possibilities).
- Be sure that you understand the logic of the program. Bonus: can your program be modified so that it could find multiple solutions to a badly formed sudoku puzzle? Try it on the following puzzle, which has two solutions.

407200000100400005000016098620300040300900000001072600002005870000600004530097061

Lab 8

Kepler's Laws

Download and unzip the file `kepler.zip` and open the file `Keplers_Laws.ipynb` in Jupyter. Follow the instructions in that Jupyter notebook.

Lab 9

Modular Exponentiation

In today's lab we will write a program that computes a huge power of an integer modulo n for some n. The big idea is to use the binary representation of the exponent to do this very quickly. For example,

$$73 = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 2^6 + 2^3 + 2^0 = 64 + 8 + 1,$$

so to compute, say, 6^{73} modulo 11, we can do the following procedure:

$$\begin{aligned}
6^1 &\equiv 6 \pmod{11} * \\
6^2 \equiv 36 &\equiv 3 \pmod{11} \\
6^4 \equiv (6^2)^2 &\equiv 3^2 \equiv 9 \pmod{11} \\
6^8 \equiv (6^4)^2 \equiv 9^2 &\equiv 81 \equiv 4 \pmod{11} * \\
6^{16} \equiv (6^8)^2 \equiv 4^2 &\equiv 16 \equiv 5 \pmod{11} \\
6^{32} \equiv (6^{16})^2 \equiv 5^2 &\equiv 25 \equiv 3 \pmod{11} \\
6^{64} \equiv (6^{32})^2 \equiv 3^2 &\equiv 9 \pmod{11} *
\end{aligned}$$

In each step, we square the result of the previous step and reduce modulo 11 so that we are never working with numbers that are too large. Then using the results from the $*$ equations above we find that

$$6^{73} \equiv 6^{64} \cdot 6^8 \cdot 6^1 \equiv 9 \cdot 4 \cdot 6 \equiv 9 \cdot 24 \equiv 9 \cdot 2 \equiv 18 \equiv 7 \pmod{11}.$$

Note that this method uses only 8 multiplications mod 11 to compute 6^{73} mod 11.

1. Write a function `power(a,b,n)` that will compute $\overline{a}^b$ in $\mathbb{Z}/n\mathbb{Z}$. Have it return the answer as an integer k with $0 \leq k < n$. It should work for very large values of a, b and n. It should **not** use the built in python function `pow(a,b,n)`.

   Use it to compute $\overline{2}^{1234567890}$ in $\mathbb{Z}/n\mathbb{Z}$ for $n = 1234567891$

   Note: To compute the binary representation of a number, you can either use the function you wrote last semester or use the Python built-in `bin(n).replace("0b","")`

2. Recall that the set

$$(\mathbb{Z}/n\mathbb{Z})^\times = \{\bar{a} : 1 \leq a < n : \gcd(a, n) = 1\}$$

is a group. (If you don't know what a group is, that's fine, you can still do this problem.) Define a function $\varphi : \mathbb{N} \to \mathbb{N}$ by the rule $\varphi(n) = |(\mathbb{Z}/n\mathbb{Z})^\times|$. This function is called the Euler phi function.

Write a function `eulerphi(n)` that determines the size of $\mathbb{Z}_n^\times$. Test your code by finding $\left|\mathbb{Z}_{11}^\times\right| = 10$, $\left|\mathbb{Z}_{56}^\times\right| = 24$, and $\left|\mathbb{Z}_{120}^\times\right| = 32$. What is $\varphi(p)$ when p is a prime?

3. Write a function `order(a,n)` that returns the order of a in the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$. (This is the smallest positive integer k such that $\bar{a}^k = \bar{1}$ in $\mathbb{Z}/n\mathbb{Z}$.) If a is not relatively prime to n, your function should raise the error "Number not relatively prime to modulus".

```
>>> order(4,11)
5
>>> order(33,56)
6
>>> order(7,120)
4
```

4. Write a function `gen(n)` that determines if $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic and returns a generator g if it is, or `False` otherwise. (It is cyclic if there is an integer g, called a generator, such that the order of $\bar{g}$ in $(\mathbb{Z}/n\mathbb{Z})^\times$ is the size of $(\mathbb{Z}/n\mathbb{Z})^\times$.) For each number n between 2 and 100, use your `gen` function to determine whether $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, and print your results in the following format:

2 1

3 2

4 3

5 2

6 5

7 3

8 False

⋮

where each line is of the form `n g`, where `g` is a generator of $(\mathbb{Z}/n\mathbb{Z})^\times$ if $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic, and just `n False` if $(\mathbb{Z}/n\mathbb{Z})^\times$ is not cyclic.

Lab 10

Visualization of multivariate functions

1. It has often been said that a picture is worth 1,000 words. Make a mesh plot to help you visualize the following functions:

   a) $f(x,y) = \dfrac{x^2 - y^2}{x^2 + y^2}$

   b) $f(x,y) = xy^2 - x^3$ (monkey saddle)

   c) $f(x,y) = \dfrac{2x^2 + 3xy + 4y^2}{3x^2 + 5y^2}$

2. Repeat exercise 1, but this time make contour plots of the functions.

3. Python can also take partial derivatives using sympy. Search the documentation for how to do this and take partial derivatives for x and y for each of the functions in exercise 1.

4. The following function represents the continuous topography of a mountain range, giving the elevation $h(x,y)$ at each point (x,y):

$$h(x,y) = \left(5000 - \frac{x^2+y^2+xy}{200} + \frac{25(x+y)}{2}\right) e^{-\left|\frac{x^2+y^2}{1000000} - \frac{3(x+y)}{2000} + \frac{7}{10}\right|}.$$

   Here is some Python-friendly code you can copy and paste (you'll need to modify `exp` and `abs` depending on how you imported numpy):

```
( 5000-0.005*(x**2+y**2+x*y)+12.5*(x+y) )
* exp( -abs(0.000001*(x**2+y**2)-0.0015*(x+y)+0.7) )
```

   Make a mesh plot to help you visualize this function. For this whole problem, keep the x and y values in the box

$$0 \le x, y \le 1600.$$

   A mosquito wants to fly from the starting point $A = (200, 200, h(200,200))$ to the ending point $B = (1400, 1400, h(1400,1400))$. It will do this by first flying straight

up to the point $(200, 200, H)$, then, maintaining a constant elevation of H, it will fly around any obstacles in its way until it reaches $(1400, 1400, H)$, at which point it will fly straight down to B. What is the minimum value of H so that this is possible? Keep in mind that the x, y values of the mosquito's position must always satisfy the condition $0 \leq x, y \leq 1600$. (You may assume that the mosquito takes up no physical space, so can fit through gaps of arbitrarily small size.)

Hint: what is the maximum value of $h(0, y)$ (or $h(x, 0)$)?

Lab 11

Maximal Paths

In this lab, we will write programs to find paths through arrays of numbers that maximize certain sums. As an example, we present the triangle below:

								19												
							41		30											
						28		**39**		62										
					96		42		**43**		19									
				56		31		09		**14**		22								
			81		87		72		25		**44**		98							
		08		40		15		08		49		**98**		27						
	69		02		23		79		26		34		**67**		41					
72		57		06		29		63		39		57		**58**		76				
64	11		39		82		97		61		83		**82**		53		97			

Our goal is to move downward from the top entry of the triangle to the bottom, choosing to go left or right as we go from each row to the next, and maximize the sum of the numbers on our path. For instance, the path going 19-41-39-43-14-44-98-67-58-82 (indicated in boldface above) has a sum of 505. It is not, however the path having the highest sum.

Triangles like the one above with 10, 15, 20, 25, 30, 100, and 200 rows are available online, at

math.byu.edu/∼nick/triangles.txt

This file consists of n rows of positive numbers, each two digits, with a space between the numbers. The maximum path sums for the first four triangles are 581, 1060, 1437, 1918, respectively.

1. Write a function `all_sums(string)` that will take as input a string representing a triangle of at most 10 rows and return a list of all the sums obtained by valid paths (sorted in increasing order, without repetition).

   The first few lines of your function should be:

   ```
   lines = string.split('\n')
   str_grid = [line.split(' ') for line in lines]
   grid = [[int(entry) for entry in line] for line in str_grid]
   ```

2. Write a function `max_path(string)` that will take as input a string representing any of the triangles given, and return the largest sum of a valid path. For small triangles, you can do this using the brute-force method of problem 1. But in order for this function to run efficiently for larger triangles, you will need to find the maximum path size more cleverly. *Hint:* consider the second row up from the bottom in the triangle above. For each entry in that row the maximum path size from that entry to the bottom row can be determined easily; you either move left or right. Now replace each entry in that row with the maximum path size from that entry to the bottom row:

$$72+64 \quad 57+39 \quad 06+82 \quad 29+97 \quad 63+97 \quad 39+83 \quad 57+83 \quad 58+82 \quad 76+97$$

You can now ignore the bottom row and think of this as a triangle with one fewer row. Now iterate.

3. Write a function `max_path_rect(string)` to find the path with largest sum for a rectangular array of numbers, where you wish to go from the upper left to the lower right corner, and you can only move right or down. Test your program on the rectangles of various sizes found at

math.byu.edu/∼nick/rectangles.txt

The maximum path sums for the first two rectangles are 1401 and 2904, respectively.

47	54	58	48	66	30	06	06	33	76	22	35	77	22	01	26
56	31	22	31	74	98	39	67	21	58	71	90	74	80	63	85
36	27	70	06	50	58	62	88	58	13	25	80	13	36	14	51
40	**89**	**57**	**75**	**76**	55	78	71	35	20	64	23	35	60	82	10
62	21	95	58	**69**	**53**	81	63	52	75	82	65	58	66	49	47
79	82	29	03	37	**87**	15	93	48	50	49	91	92	18	39	02
35	35	77	10	30	**43**	45	46	85	42	25	80	10	55	71	01
75	19	63	56	10	**82**	56	19	48	58	48	20	83	93	89	50
18	01	95	10	14	**69**	**53**	40	28	34	41	24	26	60	23	59
13	73	56	07	49	05	**99**	27	16	11	79	53	48	47	44	45
61	17	40	63	15	49	**92**	06	68	11	21	68	55	65	97	08
56	54	41	75	02	74	**32**	70	85	39	83	51	21	29	89	17
78	05	89	45	66	19	**75**	53	55	54	84	57	18	19	80	85
68	74	54	41	95	79	**74**	**76**	30	83	88	16	52	54	90	75
71	34	83	38	91	81	34	**66**	39	68	38	74	46	47	42	94
63	62	33	37	77	52	08	**42**	**22**	**65**	**36**	**80**	**17**	**54**	**25**	**98**

The path above:

47-56-36-40-89-57-75-76-69-53-87-43-82-69-53-99-
-92-32-75-74-76-66-42-22-65-36-80-17-54-25-98

has sum 1885.

Lab 12

One-dimensional Optimization and Gradient Descent

See math.byu.edu/∼nick/gradient-descent.

Lab 13

The RSA Cryptosystem

In this lab we will implement the RSA public key cryptosystem. So that we are all using the same alphabet, and so that our messages can include both uppercase and lowercase letters and some punctuation, run the following code.

```
>>> def str_to_int(s):
>>>   return int(''.join(str(ord(c)-32).rjust(2,'0') for c in s))
>>> def int_to_str(n):
>>>   return '' if n==0 else int_to_str(n//100)+chr((n%100)+32)
```

To check that everything is working:

```
>>> str_to_int('"Hello, world?"')
2406976767912008779827668 3102
>>> int_to_str(2406976767912008779827668 3102)
'"Hello, world?"'
```

We will need the following fact from group theory: for any positive integer N, let $\varphi(N)$ denote the size of the group $(\mathbb{Z}/N\mathbb{Z})^\times$, as in Lab 9. If a is relatively prime to N then

$$a^{\varphi(N)} \equiv 1 \pmod{N}.$$

(If G is a finite group of order m then $a^m = e$ for all $a \in G$.) Hence, if $k \equiv 1 \pmod{\varphi(N)}$, we can write $k = 1 + \ell\varphi(N)$, and we have

$$a^k = a^1(a^{\varphi(N)})^\ell \equiv a \pmod{N}.$$

In order to create an RSA key pair, we will find an integer N that is a product of two large primes p and q. For this value of $N = pq$, we know that $\varphi(N) = (p-1)(q-1)$. We will choose an integer e that is relatively prime to $\varphi(N)$ (usually the prime 1234577 will work). Then we compute d such that $de \equiv 1 \pmod{\varphi(N)}$. The public key is the pair

$$[e, N]$$

and the private key (which must be kept secret) is the pair

$$[d, N].$$

To encrypt a message to a person, you first look up their public key $[e, N]$. Convert your message to an integer M using `str_to_int`. Then compute

$$C \equiv M^e \pmod{N} \qquad \text{with } 0 \leq C < N.$$

This number is the encrypted message that is sent.

To decrypt a message C sent to you that is encrypted as above with your public key $[e, N]$, you would compute

$$P \equiv C^d \pmod{N} \qquad \text{with } 0 \leq P < N.$$

Since $C \equiv M^e \pmod{N}$, we see that $P \equiv (M^e)^d \equiv M^{de} \equiv M \pmod{N}$, since $de \equiv 1 \pmod{\phi(N)}$. Hence, you would retrieve the original message. Note that you should be the only one who knows your private key, so you should be the only one able to read the message.

If two people, Alice and Bob, each have a public/private key pair (say Alice's is $[e_A, N_A]$ and $[d_A, N_A]$ and Bob's is $[e_B, N_B]$ and $[d_B, N_B]$), then Alice can send Bob an encrypted and signed message as follows.

Let M be a number smaller than N representing the message. Then Alice encrypts it using Bob's public key $[e_B, N_B]$ to get

$$C \equiv M^{e_B} \pmod{N_B} \qquad \text{with } 0 \leq C < N_B.$$

Alice will then take the encrypted text C and run it though her decryption key (which is something that only she can do) to get a signature:

$$S \equiv C^{d_A} \pmod{N_A} \qquad \text{with } 0 \leq S < N_A.$$

She then sends Bob the pair of numbers (C, S). Using his decryption key (which only he can do), Bob can decrypt C to obtain and read M. Using Alice's encryption key (which is publicly known) on S will result in a number $V \equiv S^{e_A} \pmod{N_A}$. If $V \equiv C \pmod{N_A}$, this verifies that the message comes from Alice. If $V \not\equiv C \pmod{N_A}$, then the message is not from Alice.

Your assignment:

1. Create three functions:

   (a) A function `create_key(p,q)` that takes two large primes `p` and `q`, and creates a public key $[e, N]$ and a private key $[d, N]$ where $N = pq$ and e is the smallest number larger than 1234576 that is relatively prime to $\phi(N) = (p-1)(q-1)$. The decryption exponent is unique modulo $\varphi(N)$, so d should be between 0 and $\varphi(N)$.

   (b) A function `encrypt(message,e_key)` that takes a string and a public encryption key, and returns an integer C that is the RSA encrypted message.

   (c) A function `decrypt(C,d_key)` that takes an integer and a private decryption key, and returns the decrypted message as a string.

2. A list of large primes (over 100 digits) has been placed online at

   math.byu.edu/~doud/RSA/largeprimes

   Using the first two primes from this list, create a public/private key pair. Encrypt the message "I have finished part 2 of the assignment!" using the public key. You should probably check that the private key, together with your `decrypt` function, correctly decrypts the message.

3. You have been sent an encrypted message:

   math.byu.edu/~doud/RSA/keypair

   Note that this page contains both your public (encryption) key, EKEY, and your private (decryption) key, DKEY. It also contains a number C, which is the encrypted message that has been sent to you, encrypted with EKEY.

   Decrypt the message!

4. A message has been encrypted using a key created in exactly the way specified in Problem 2; by choosing two primes from the list of large primes, and creating a key. The public key used (and the number C representing the encrypted message) are here:

   math.byu.edu/~doud/RSA/publickey

   Write a function `RSA_crack(primes,C,e_key)` which takes a list of primes from which `e_key` was generated (you may assume that N is a product of two distinct primes from this list), an encrypted message `C`, and a public encryption key `e_key`, and returns the decrypted message.

   Use your function to find the private (decryption) key used above and decrypt the message.

5. Write a function `verify_signature(C,S,e_key)` which takes as input an encrypted message C, a signature S, and the encryption key of the purported sender. Both C and S will be integers. The function should return True if the signature is valid, and False otherwise.

   As before, your RSA key is found at

   math.byu.edu/~doud/RSA/keypair

   Two people, each claiming to be Alice, have sent you a message encrypted with your public key. Both messages (and their signatures) are contained in the web page

   math.byu.edu/~doud/RSA/SignedMessages

   Assuming that Alice keeps her private key secret, and using the list of public keys at

   math.byu.edu/~doud/RSA/PublicKeyDatabase

decrypt both messages and check the signatures to determine which (if any) of the two messages is really from Alice. Can you tell who the other one is from?

Lab 14

Riemann sums

In this lab we will study Riemann sums in higher dimensions.

1. Write a function `riemann_sum_2D` which takes 7 parameters `f, xMin, xMax, yMin, YMax, N`, and `method` and returns the Riemann sum

$$\sum_{j=1}^{N}\sum_{i=1}^{N} f(x_i^*, y_j^*)\Delta x\Delta y$$

   where $\Delta x = (\texttt{xMax} - \texttt{xMin})/\texttt{N}$, $\Delta y = (\texttt{yMax} - \texttt{yMin})/\texttt{N}$ and `method` determines whether we are using the lower left, upper right, or midpoints of the partition. (The options for the method should be `left, right, mid`).

2. Using your function from problem 1, use the midpoint method to calculate the Riemann sums for $N = 10$ and $N = 20$ for the following functions and domains:

   a) $f(x, y) = x\sin(xy)$ on the rectangle $[0, \pi] \times [0, \pi]$.

   b) $f(x, y) = y^2 e^{-x-y}$ on the rectangle $[0, 1] \times [0, 1]$.

   c) $f(x, y) = x^3y^2 + xy$ on the rectangle $[0, 1] \times [1, 2]$.

3. Consider the integral of $f(x, y) = x\sin(x + y)$ on the rectangle $[0, \pi/6] \times [0, \pi/3]$. First calculate the value of this integral analytically. Then make a plot that shows the error of the midpoint Riemann integral approximation as N ranges from 1 to 100.

4. Write a function `riemann_sum_3D` which takes 9 parameters `f, xMin, xMax, yMin, YMax, zMin, zMax, N`, and `method` and returns the Riemann sum

$$\sum_{k=1}^{N}\sum_{j=1}^{N}\sum_{i=1}^{N} f(x_i^*, y_j^*)\Delta x\Delta y\Delta z,$$

   where $\Delta x = (\texttt{xMax} - \texttt{xMin})/\texttt{N}$, $\Delta y = (\texttt{yMax} - \texttt{yMin})/\texttt{N}$, $\Delta z = (\texttt{zMax} - \texttt{zMin})/\texttt{N}$ and `method` determines whether we are using the lower left, upper right, or midpoint of the partition. (*Hint:* you can copy much of the code you wrote for Problem 1.)

5. Using your function from problem 4, use the midpoint method to calculate the Riemann sums for $N = 10$ and $N = 20$ for the following function and domain:

$$f(x, y, z) = xy + z^2 \text{ on the rectangle } [0, 2] \times [0, 1] \times [0, 3].$$

Lab 15

Solving the Rubik's cube, Part I

The set of all possible "moves" of the Rubik's cube is a good example of a group. Each possible move can be constructed as a product of moves consisting of rotating a single face of the Rubik's cube through a 90° rotation, either clockwise or counterclockwise. We will use the following notation for these moves:

One face of the Rubik's cube will be specified as the "Up" (top) face, and another as the "Front" face. These specifications will fix a "Down" (bottom) face (the face opposite the top face), a "Back" face (opposite the front face), and a "Right" and "Left" face (on the right or left as you look at the front face, with the top face on the top). We will use the following letters to denote the different faces:

U	Top
F	Front
R	Right
L	Left
B	Back
D	Down

Representing a turn of a face will be indicated by giving the letter of the face that we wish to turn, in upper case for a clockwise turn (as we look directly at the face), and in lower case for a counterclockwise turn. The turns will be listed in order from left to right in the order in which we perform them. For instance, the sequence `RFrf` would correspond to the following:

1. Turn the right face 90 degrees clockwise.
2. Turn the front face 90 degrees clockwise.
3. Turn the right face 90 degrees counterclockwise.
4. Turn the front face 90 degrees counterclockwise.

To keep the programming easier, we will not allow ourselves to turn the entire cube (in particular the squares in the center of each face will never change position). All moves that we allow will be moves of individual faces.

We will number the smaller faces of the Rubik's cube as indicated on the handout at

https://math.byu.edu/~doud/Math495R/Rubik-handout.pdf

With this numbering, you should check that turning the right face 90 degrees clockwise (the move denoted by `R`) would move the small face in spot 2 to spot 18, the face in spot 18 to spot 23, the face in spot 23 to spot 7, and the face in spot 7 to spot 2. The move `R` will actually involve five such 4-cycles: we have that as a permutation of the faces,

$$\texttt{R} = (2, 18, 23, 7)(3, 10, 22, 15)(6, 14, 19, 11)(26, 37, 46, 35)(30, 34, 42, 38)$$

We will define a variable `cube` that keeps track of the position of each sub-face as we manipulate the cube. When the cube is solved, `cube` will be a list of length 49, with the ith entry equal to i (note that there are only 48 subfaces that can move, but in order to keep the numbering simple, we will leave the zero entry always equal to 0).

1. Find the cycle structures for the moves `R`, `L`, `U`, `D`, `F`, and `B`. As an example, the cycle representing `R` would be

   `[[2,18,23,7],[3,10,22,15],[6,14,19,11],[26,37,46,35],[30,34,42,38]]`.

   Store these cycle structures in variables named `Right`, `Left`, `Up`, `Down`, `Front`, and `Back`. I suggest you crowdsource this part of the assignment.

2. Write a function `apply(move,cube)` that takes a list representing a movement of the cube (written as a product of cycles as in problem 1) and a list representing the cube (as described above), and performs the indicated move. It should actually change the values of the list `cube`, so that it does not actually need to return a value.

3. Write a function `execute(moves,cube)` that takes a string containing moves of the cube, and a position of the cube, and executes those moves on the cube.

   The function should apply all of the indicated moves to the cube, actually changing the values of the list. For instance, for a capital letter `R`, it should apply the permutation `Right` that you found above. For a lowercase `r`, it should apply the permutation `Right` three times (since a 90° counterclockwise move is the same as three 90° clockwise moves). After applying all moves specified in the string, the function should then return the sequence of moves that it made. If `moves` contains characters other than "`FBRLUDfbrlud`", these characters should be ignored, and not included in the returned variable.

   After this function is written and debugged, you should not change `cube` other than through this function (except to debug your code).

   In order to test the `execute` function, use it to find the orders of the following sequences of moves: `'RF'` should have order 105 (i.e., applying the move `RF` 105 times should return the cube to its initial position, but fewer than 105 times should not), `Ur` should have order 63, `rdL` should have order 180, and `RDFLBUrdflbu` should have order 360. Note that this last sequence of moves uses all twelve possible moves, so it should be a good test for your function.

4. Write functions: `align25(cube)`, `align26(cube)`, `align27(cube)`, `align28(cube)`, that put the subfaces numbered 25, 26, 27, and 28 in their correct positions. Each one should perform its task without changing the positions of the other subfaces in the set $\{25, 26, 27, 28\}$. Each function should return the sequence of moves that it makes.

   Here is my suggestion for how to write these functions.

   For `align25`, create a list of 24 sequences of moves. Entry 0 of this list would be the sequence of moves necessary to move subface 25 from position 25 to position 25; entry 1 of this list would be the sequence of moves necessary to move subface 25 from position 26 to position 25, and so on. Find the current position of subface 25, lookup the correct sequence of moves in the list, and execute the moves.

   For `align25`, here is what my function looks like:

```
def align25(cube):
    i=cube.index(25)
    move=['25','u26','uu27','U28','BLU29','RB30','uRB31','lb32',...][i-25]
    return execute(move,cube)
```

   In order to help me keep track of which entry of the list is which, I have included the starting position of subface 25 (so, for instance, "`Bru41`" would indicate the sequence of moves needed to move the subface in position 41 to position 25). If the `execute` function is correctly written, these extra digits will be ignored when the move is executed.

   The functions for `align26`, `align27`, `align28` can be constructed similarly.

5. Create four functions, `align1(cube)`, `align2(cube)`, `align3(cube)`, and `align4(cube)`, that return sequences of moves that put subfaces 1 through 4 in their correct positions without disturbing subfaces 25 to 28. These functions should work similarly to the ones written in part 4.

6. Create a function `solvetop(cube)` that has a scrambled cube as input, and returns a sequence of moves that solves the top level of the cube. The easiest way to do this would be to run each of the functions `align25(cube)`, `align26(cube)`, `align27(cube)`, `align28(cube)`, `align1(cube)`, `align2(cube)`, `align3(cube)`, and `align4(cube)` in the indicated order; put together their output into a single string, and return that string. Done efficiently, this can be a one or two line function.

   You should test your `solvetop` program using some of the 100 scrambled cubes found:
   math.byu.edu/∼doud/Math495R/Cubes.txt
   If your routines work properly, subfaces 1–12 and 24–32 should all be put into the correct positions by `solvetop(cube)`.
   If you need to see a graphical simulation of a cube, you can find one online at
   math.byu.edu/∼doud/RubiksCube/
   Enter a sequence of moves into the textbox at the top of the page, click the button labeled "Turn," and the program will execute the specified moves.

Lab 16

Solving the Rubik's cube, Part II

In this lab, we will continue programming our solution to the Rubik's cube. We will also construct several functions to help us debug our program.

Before beginning this lab, you should be sure that your functions from Lab 15 work correctly. They will be needed for this lab.

We will do the following in this lab: putting subfaces 33–40 in the correct position, and getting the edge faces 45–48 all on the bottom face. In the next (and final) Rubik's cube lab, we will complete the solution of the cube.

1. As a preliminary, we will create a function which scrambles a Rubik's cube. The function should have the following syntax: `scramble(cube,n=50)`. As input, it will take a vector representing a state of the Rubik's cube, and an (optional) integer n. It will then execute n random moves on the cube to scramble it. It should move the elements of `cube`, and return only the list of moves that it made.

Note that there are 12 possible moves: `"RLUDFBrludfb"`. To choose one at random use "`import random`" at the beginning of your program, and then `random.randint(0,11)` will choose a random integer from 0 to 11 (inclusive).

2. We now create a function that will translate sequences of moves designed to move subfaces on the front of the cube so that they move subfaces on other sides of the cube.

For instance, if we have a sequence of moves that takes the contents of the bottom front edge (location 43) and moves it to the right front edge (location 35), we want to be able to adjust it to move location 42 to 34 (on the right side), location 41 to 33 (on the back), or location 44 to 36, on the left.

To do this, hold up your cube handout, with the "Up" face on top, and the "Front" face facing you. Now rotate the cube 90° counterclockwise so that the "Left" face faces you. The face on the right hand side is now the face labeled "Front", so if you were to do an `R` move without reference to the labels on the cube, it would actually be a `F` move. Similarly, `L` would actually perform `B`, `F` would actually perform `L`, and `B` would actually perform `R`. The moves `T` and `D` would have the same effects as if the cube had not been rotated.

Note that if we do a sequence of moves that transfer the contents of location 43 to location 35, then after the rotation it would rotate the contents of location 44 to location 36.

Construct a function `toprotate(moves)` that will translate a series of moves by 90°, as above: in other words, it will make the following substitutions:

$$\texttt{R} \to \texttt{F}, \quad \texttt{F} \to \texttt{L}, \quad \texttt{L} \to \texttt{B}, \quad \texttt{B} \to \texttt{R}$$

$$\texttt{r} \to \texttt{f}, \quad \texttt{f} \to \texttt{l}, \quad \texttt{l} \to \texttt{b}, \quad \texttt{b} \to \texttt{r}$$

Probably the easiest way to do this is with the `translate` method on strings. To illustrate this method, run the following lines of code:

```
word="Convert abcde to numbers."
translation=str.maketrans("abcde","12345")
print(word.translate(translation))
```

Notice that each a has been turned into a 1, each b into a 2, and so on. Use this technique to make the translation function described above. Once you have `toprotate` working properly, the command `toprotate("RFLBrflb")` should return "`FLBRflbr`".

3. Create a function `edgesfrombottom(cube)` that will (assuming that the top face of the cube is solved) look at which faces are in locations 41 to 44, and if any face numbered between 33 and 40 is in one of these four spots will place it in its proper location. It should exit when all of the subfaces in locations 41 to 44 have numbers higher than 40. It should initialize a variable `result=""`, and use this variable to record and return any moves that it executes.

The first step to placing a subface in its proper place is to rotate it to the proper side of the cube. Note that all the edges on the front of the cube have numbers congruent to 3 mod 4; all the ones on the right have numbers congruent to 2 mod 4, all the ones on the back have numbers congruent to 1 mod 4, and all the ones on the left have numbers congruent to 0 mod 4. So, if we find a face numbered 33–40 in one of the spots 41–44, it is on the correct side of the cube if its number is congruent mod 4 to the position that it is in (i.e. if `(i-cube[i])%4)==0`. If it is in the wrong position, we need to use a `d`, `dd`, or `D` move to put it on the correct side.

Once we have it on the correct side, we need to move it to the correct position in the middle layer of the cube.

To do this, note that the sequence of moves `"drDRDFdf"` will move the contents of face 43 (the bottom edge of the front) to face 35 (the right edge of the front) without disturbing the top face or any of faces 33–40 except for 35 and 38. The sequence of moves `"DLdldfDF"` will move the contents of face 43 to face 39 (the left edge of the front) under similar restrictions.

These moves adjust the "Front" face of the cube. If we apply `toprotate` to them, they will have the desired effect on the "Left" face of the cube; namely moving the contents of position 44 to either position 36 or 40. Applying `toprotate` twice will affect the "Back" face of the cube, and applying `toprotate` three times will affect the "Right" face of the cube. (Try to find a formula for how many times you need to apply `toprotate`, instead of using multiple if statements.)

After checking each of locations 41 to 44, if no moves were executed (i.e., if `result` is still empty), the function should return the empty string. If any moves were executed, then we need to run the code again (since we may have disturbed a location that we had previously checked). Do this by returning `result+edgesfrombottom(cube)`.

In writing the code for this lab, the functions on the next page that test your code may be useful. If `testall(10000)` yields a positive result, then your code has worked correctly on 10000 scrambled cubes. These functions can be downloaded at

math.byu.edu/~doud/Math495R/Lab16Debug.py

After you finish this lab, you should go to Lab 20 and start working on the next function.

To test (and debug) the function `edgesfrombottom`, you can use the following function:

```
def test(n):
   flag=True
   for i in range(n):                          #Tests your function on n cubes
      cube=list(range(49))                     #Initializes a solved cube
      t=scramble(cube)                         #Randomizes the cube
      t=solvetop(cube)                         #Solves the top face of the cube
      t=edgesfrombottom(cube)                  #Moves any side edges off of the bottom
      for position in range(41,45):            #Looks in locations 41 to 44
         if cube[position]<41:                 #If any side edges are there
            flag=False                         #Report a problem
            print("Your function has a problem")
   if flag:
      print("Your function seems to be fine.")
```

To test and debug the function `edgesfromsides`, use the following function.

```
def test2(n):
   flag=True
   for i in range(n):                          #Tests your function on n cubes
      cube=list(range(49))                     #Initializes a solved cube
      t=scramble(cube)                         #Randomizes the cube
      t=solvetop(cube)                         #Solves the top face of the cube
      t=edgesfrombottom(cube)                  #Moves any side edges off of the bottom
      t=edgesfromsides(cube)                   #Corrects the position of side edges
      if cube[33:41]!=list(range(33,41)):      #Checks the side edges
            flag=False
            print("Your function has a problem")
            print(cube[33:41])
   if flag:
      print("Your functions seem to be fine.")
```

The following function tests all of the functions that we have written so far.

```
def testall(n):
    flag=True
    for i in range(n):                          #Tests your function on n cubes
        cube=list(range(49))                    #Initializes a solved cube
        t=scramble(cube)                        #Randomizes the cube
        t=solvetop(cube)                        #Solves the top face of the cube
        t=edgesfrombottom(cube)                 #Moves any side edges off of the bottom
        t=edgesfromsides(cube)                  #Puts side edges in correct spots
        t=flipbottomedges(cube)                 #Flips bottom edges to the bottom
        if cube[1:13]!=list(range(1,13)):       #Tests top face
            flag=False
        if cube[25:41]!=list(range(25,41)):     #Tests second layer
            flag=False
        if min(cube[45:])<45:                   #Tests edges on bottom face
            flag=False
    if flag:
        print("Your code seems to be working")
```

Lab 17

Monte Carlo Integration

See math.byu.edu/∼nick/montecarlo.

Lab 18

Visualizing Vector Fields

In this lab you will create visualizations to help you understand vector fields. Before beginning the lab, please review how to create quiver plots in python.

problemsolvingwithpython.com/06-Plotting-with-Matplotlib/
06.15-Quiver-and-Stream-Plots/

A quiver plot is a plot that shows vectors as arrows and is especially useful when talking about vector fields.

1. Plot the vector field $\mathbf{F}(x, y) = (y^2 - 2xy)\mathbf{i} + (3xy - 6x^2)\mathbf{j}$.

2. Let $\mathbf{F}(\mathbf{x}) = (r^2 - 2r)\mathbf{x}$, where $\mathbf{x} = \langle x, y \rangle$ and $r = |\mathbf{x}|$. You may have to vary your domain to see what is happening. First plot for $-1 \leq x, y \leq 1$ with a step size of 0.1. Then plot again for $-5 \leq x, y \leq 5$ with a step size of 0.5.

3. Plot the gradient vector field of f together with a contour map of f for:

   a) $f(x, y) = \ln(1 + x^2 + 2y^2)$

   b) $f(x, y) = \cos x - 2 \sin y$

4. Of course there are also vector fields in higher dimensions. See
   matplotlib.org/3.1.1/gallery/mplot3d/quiver3d.html

   Plot the following vector fields:

   a) $\mathbf{F}(x, y, z) = \mathbf{i} + 2\mathbf{j} + 3\mathbf{k}$

   b) $\mathbf{F}(x, y, z) = \mathbf{i} + 2\mathbf{j} + z\mathbf{k}$

   c) $\mathbf{F}(x, y, z) = x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$

   d) $\mathbf{F}(x, y, z) = \sin y\mathbf{i} + (x \cos y + \cos z)\mathbf{j} - y \sin z\mathbf{k}$

Lab 19

Solving the Rubik's cube, Part III

In this lab we will solve the remaining parts of the Rubik's cube.

1. Create a function `edgesfromsides(cube)`. It should look at the subfaces in positions 33 to 36, and if one of them is incorrect, it should move this face to the bottom edge, and then run `edgesfrombottom(cube)` to place the edge in its correct position. Once all four positions from 33 to 36 have been dealt with, the second layer of the cube should be solved.

   To move a face from a side edge to the bottom, use the basic move "`drDRDFdf`" (for position 35), with `toprotate` applied the appropriate number of times to make it apply to the other three positions.

   At this point, you have code that solves the top two layers of the Rubik's cube.

2. We will now work on putting the subfaces numbered 45 to 48 on the bottom face of the cube. Many instruction manuals for solving the cube call this "creating a yellow cross", since they typically have the bottom face be yellow, and when all four bottom edges are on the bottom, they form the shape of a cross.

   In order to do this, we will create a function called `flipbottomedges(cube)`. It should change the entries of cube, and return the sequence of moves that it executes.

   It turns out that there are only four possibilities for how the bottom faces can be positioned.

   (1) None of them are on the bottom. In this case, the sequence of moves "`FLDldfBRDrdRDrdb`" will flip them all to be on the bottom.

   (2) Two of them are on the bottom, opposite from each other. In this case, after rotating the bottom so that the two correct ones are in position 46 and 48, the sequence of moves "`FLDldf`" will leave all four bottom faces on the bottom.

   (3) Two of them are on the bottom, adjacent to each other. In this case, after rotating the bottom so that the two correct ones are in positions 45 and 48, the sequence of moves "`RDFdfr`" will leave all four bottom faces on the bottom.

   (4) All four of the bottom edges are already on the bottom; in this case, nothing needs to be done.

Count the number of subfaces in position 45–48 that are numbered above 45–48. If this number is 0, we are in case 1 above. If the number is 4, we are in case 4 above. Otherwise, we are in either case 2 or case 3, and we should perform a sequence of `D` moves until location 48 contains a bottom face and location 47 does not. Then, depending on which of positions 45 and 46 contain bottom faces, we perform the appropriate sequence of moves.

3. We will now write a function `positionbottomedges(cube)` that will put the bottom edges in the correct positions. It should change `cube`, and return the sequence of moves that it executes.

   At this point, all four bottom edges are on the bottom of the cube, but they are probably not positioned correctly. It will always be possible to rotate the bottom face in such a way that either 2 or 4 of the bottom faces are positioned correctly. To do this, count the number of subfaces in positions 45–48 that are numbered correctly. As long as that number is less than two, execute a `D` move, and then repeat.

   If all four faces are positioned correctly, we are finished with this step. If only two faces are positioned correctly, we have more to do.

   If the two faces that are positioned correctly are opposite each other, they are either in positions 46 and 48 or positions 45 and 47. In the first case execute the sequence of moves: "`FDfDFDDfRDrDRDDrD`". In the second case, `toprotate` this sequence of moves once, and execute the result. This will put all four edge pieces in the correct positions.

   If the two faces that are positioned correctly are adjacent to each other, if they are in positions 45 and 48, execute the sequence of moves "`LDlDLDDlD`. If they are positioned differently, `toprotate` this sequence of commands the correct number of times to have the desired effect. This will put all four edge pieces in the correct positions. If the correct ones are in positions 45 and 46 a single `toprotate` should suffice; if they are in position 46 and 47, two `toprotate`s will be needed, if they are in positions 47 and 48, three `toprotate`s will be needed.

4. We will now write a function `positionbottomcorners(cube)` that will put the bottom corners in the correct positions. It should change `cube`, and return the sequence of moves that it executes. We will identify the bottom corner pieces by looking at positions 21, 22, 23, and 24. Note that if the face in one of these positions is congruent modulo 4 to its position, (i.e. if `(cube[i]-i)%4==0`) then it is in the correct position, but might be twisted incorrectly.

   To position the bottom corners, we have a basic move, "`rDLdRDld`" that fixes the corner containing position 24, and moves the other three corners in a clockwise pattern (twisting them as they move). If the cube has been solved to this point, it should be in one of the states below:

   (1) If none of the corners are in the correct position, execute the basic move. One will now be in the correct position.

(2) If one of the corners is correct, use toprotate on the basic move above the correct number of times so that it leaves the correct corner fixed. Then applying the toprotated basic move either once or twice will put all the corners in the correct position.

(3) It should never happen that two or three of the corners are in the correct position.

(4) If all four corners are in the correct position, we are done with this step.

5. We are now ready for the final step. This step will rotate each corner cube into the correct orientation.

   Write a function called `rotatebottomcorners(cube)`. It should change the cube, and return the sequence of moves that it makes.

   The function should consist of a loop that does the following four times.

   (1) While the face in position 21 is is numbered less than 21, execute the following move "`buBUbuBU`". (You may have to do this twice.)

   (2) Execute the move `D`.

6. Finally, we put together all of the functions that we have produced so far, to get a single function that solves the Rubik's cube. Your function might look somthing like the one below:

```
def solvecube(cube):
    result=""
    result+=solvetop(cube)
    result+=edgesfrombottom(cube)
    result+=edgesfromsides(cube)
    result+=flipbottomedges(cube)
    result+=positionbottomedges(cube)
    result+=positionbottomcorners(cube)
    result+=rotatebottomcorners(cube)
    return result
```

Once you have written this function, use the functions at the website

math.byu.edu/∼doud/Math495R/Lab16Debug.py

to test it on a large number of cubes.

Lab 20

Machine Learning with Nim, Part I

In this Lab, we will program a computer to learn to play the game of Nim. Note that we will *not* program the computer to play perfectly–indeed, we will begin with a computer that plays randomly (and probably loses most of the time). We will then train the computer to recognize winning moves, until the computer plays perfectly.

Rules of Nim

A game of Nim begins with three piles of objects; one having 7 objects, the other having 5 objects, and the last having 3 objects. Two players take turns removing objects from the piles. A turn consists of removing any positive number of objects from one of the piles. The winner is the last player that is able to make a valid move (i.e. the player who removes the last objects from the last nonempty pile).

There are many variations of Nim on the internet; we could use different numbers of objects in each pile, or have a different number of piles. In some versions, the person who takes the last object loses. For our game, we will use the rules listed above.

Nim is a completely solved game: with perfect play, depending on the starting configuration, either the first player will always win, or the second player will always win. For our starting configuration, unless the first player plays poorly, the first player will always win.

A template for this lab is available at

math.byu.edu/∼doud/Math495R/Lab20template.py

1. The first function that we will write is `checkmove(position,pile,remove)`. The first input is a list `position` indicating the current position of the board. The second, `pile`, is an integer `1`, `2`, or `3`, and the third, `remove`, is a positive integer. The function should check whether taking `remove` objects from pile `pile` is a valid move, and return `True` or `False`. For example, if `position=[7,5,3]`, then `pile=3, remove=1` would be a valid move, but `pile=3, remove=5` would not (since there are not five objects on pile 3 to remove).

2. The next function is `player(n,position)`. The input to this function is an integer `n` indicating whether the human player is player one or player two, and a list `position`

indicating the position of the board prior to the player's move.

The function should ask the player "Player `n`, what move would you like to make?" (with "n" filled in with the playernumber) and accept an answer as a string. The string should be of the form `3 1` (two integers separated by a space). The first integer indicates which pile the player wishes to remove objects from, and the second indicates how many objects should be removed. The program should parse this string into two integers, and check whether it is a valid move for the indicated position.

If the move is not valid, the program should indicate this, and ask the player again for a valid move.

If the move is valid, the program should subtract the given number of objects from the correct component of `position`. Note that the program does not need to return a value; `position` is a vector that was passed to the function, so changing a component will change the original vector.

3. The second function that we need to write is `playgame(strategy)`. For the moment, we will not use the input variable `strategy`; it will be used to store the computer's strategy, once we have developed a strategy for the computer.

   This function should begin by asking "Would you like to play a game? (Y/N)". It should accept input in the form of a string. If the string is either "Y" or "y", it should proceed; if the input is "N" or "n" it should terminate, and other input should lead to the question being asked again.

   If the program proceeds, is should ask

   "Player 1: Computer (C) or Human (H)? (C/H) "

   and accept a string as an answer. It should store this answer in the variable `P1`. If the answer is neither "C" nor "H", it should ask the question again.

   It should then repeat the process for player 2, storing the response in the variable `P2`.

   If both players are computers, the program should ask "How many games?" and accept an integer as a response. If either player is human, this question should not be asked–a single game will be played.

   If more than one game is to be played (which should only happen with two computer players), the program should print

   "—Beginning Game 1—".

   The program should then initialize the board, setting the position to `[7,5,3]`. Beginning with player one, it should print the board position, in the format

   "The piles have 7, 5, 3 objects"

(where the numbers are taken from the vector indicating the position) and then call the function `player(n,position)` or `computer(n,position,moves,strategy)` (depending on whether player `n` is a human or computer) to allow the player to make a move. It should then change the player number, and repeat (printing "The piles have $*$, $*$, $*$ objects" each time), until the position becomes `[0,0,0]`.

Once the game is over, if both players are computers and multiple games have been requested, it should print

"—Beginning Game 2—"

and initialize and play a new game with two computer players. It should repeat this until the desired number of games have been played. Once all of the computer vs. computer games are over, it should print

"—Completed `n` games—"

where `n` is the number of games requested.

If either player is human, then once the game is over, to program should ask "Would you like to play again? (Y/N)", and depending on the answer, start over (asking if the players are human or computers), or terminate.

At this point, you should be able to play a two-person game (with no computer players) by running the program.

4. In order to have a computer play, we need to tell it how to play. We will create a vector called `strategy`, indexed by integers from 0 to 753. If the integer `n` is a valid board position for the game (i.e. the first digit is from 0 to 7, the second digit from 0 to 5, and the third digit from 0 to 3), then `strategy[n]` will itself be a vector containing all possible moves from the given position. Each move will be represented by a triple of numbers, representing the pile, number of objects to remove, and the quality of the move (to be described later). For instance, if the integer `n` is equal to 312, the board position will be `[3,1,2]`, and `strategy[312]` will be the vector

$$[[1, 3, 50], [1, 2, 50], [1, 1, 50], [2, 1, 50], [3, 2, 50], [3, 1, 50]].$$

For now, we just set the third number in each triple to 50 (it will be used to determine which moves are best, later).

Create a function `initialize()` that will create and return the desired vector `strategy`.

Lab 21

Machine Learning with Nim, Part II

In this part of the lab, we will train the computer to learn from its mistakes and become a perfect Nim player. We can only hope that it does not learn more than this.

1. We first teach the computer the rules of the game, so that it can play very badly. The function `computer(n,position,moves,strategy)` takes as input an integer `n` indicating the player number, a vector `position` giving the board configuration, a vector `moves` which we describe below, and the vector `strategy` that tells the computer all valid moves.

   The variable `moves` should be a vector of length two. The first component of this vector will consist of moves made so far by player one, and the second will consist of moves made so far by player 2. As the computer makes a move, it will add information describing that move to `moves[n-1]`.

   This routine should first convert `position` into a three-digit integer `pos`. It will then look at the possible moves in `strategy[pos]`. Set `mx` equal to the largest value of the third component of the possible moves, and then create a vector `bestmoves` containing the indices in `strategy[pos]` of all the possible moves whose third component matches that maximum value. As an example, if `position=[3,1,2]` then `pos=312` and we might have

   ```
   strategy[pos]=[[1,3,80],[1,2,20],[1,1,40],[2,1,80],[3,2,60],[3,1,80]]
   ```

   The maximum value of the third component is 80, and we should end up with the vector `bestmoves=[0,3,5]`, indicating that the 0, 3, and 5 entries of the vector achieve the maximum.

   Randomly choose an element `m` of `bestmoves` (this can be done, for instance, with the command `m=bestmoves[random.randint(0,len(bestmoves)-1)]`, and execute the indicated move (i.e. the move described in `strategy[pos][m]` by subtracting the appropriate number of objects from the appropriate pile). Print out "Computer player `n` takes `a` objects from pile `b`", and append the vector `[pos,m]` to the vector `moves[n]`.

   Finally, if the current value of `position` is equal to `[0,0,0]`, print "Computer player `n` wins" and exit the function. Note that no value needs to be returned.

Your function should now allow you to play a computer. The computer should, at this point, be selecting random moves from all possible moves (because every possible move has a "quality" of 50). You should be able to defeat this random computer easily.

2. Now copy your function `computer` into the function `computertrainer`. We will modify it slightly to make it more suitable to training the computer. To do this, we will want to do two things differently.

   First, eliminate the print statements. We will train the computer by having it play thousands of games against itself, and we don't want to see the output of those games.

   Second, when selecting the best moves, choose all the moves with a quality greater than `mx-90`. If we only select the best moves, then the computer will only ever try one winning move of all the possible moves; namely the first one that leads it to a victory. There could be other, even better moves available, and we want the computer to try them out. Note that the value 90 is arbitrary, after the next step you may want to experiment with it, to see how best to train the computer. Basically a value closer to 100 reserves judgement on whether a move is good or bad until later; a smaller value judges a move to be permanently bad as soon as there is one that is marginally better.

3. Our final function will be `traincomputer(strategy)`. This function will teach the computer which moves are the best, so that it can consistently win.

   We want the computer to play thousands of games against itself. Each time the computer wins, it should increase the quality of the moves that it made; each time it loses, it should decrease the quality. Eventually, the best moves will become apparent. Although this is the basic idea, we will make some modifications so that the best moves become apparent faster.

   The function `traincomputer(strategy)` will consist of a large loop, which will cause the computer to play against itself over and over, becoming a little bit better each time.

   The first thing to do inside the loop is to set the starting position, `position=[7,5,3]`, the player number `playernumber=2`, and initialize the list of moves that have been made, `moves=[[],[]]`. Note that the moves made are divided between moves made by player one and moves made by player two.

   Next, have the computer play a game against itself, using the `computertrainer` function. When the game ends, keep track of which player was the winner, and which was the loser.

   Now, examine the moves made by the winner. Each of these moves will consist of a pair, `[pos,move]`, where `pos` is the position before the move, and `mov` is the number of the move that was made from that position. For each of them increase the value of `strategy[pos][mov][2]` by 29 (but only up to a maximum of 100).

   Now examine the moves made by the loser. Each of these moves will consist of a pair, `[pos,move]`, where `pos` is the position before the move, and `mov` is the number of the move that was made from that position. For each of them decrease the value of `strategy[pos][mov][2]` by 11 (but only down to a minimum of 0).

As a further innovation, the last move made by the winner is a certain win; adjust the quality of that move to 1000 (we always want to make that move if we are in this position, regardless of how good the other moves are). The last move made by the loser leads immediately to a win by the other player, decrease the quality of that move to -1000 (since we would prefer to never make that move again, and don't need any more practice to know that it is a bad move).

With the parameters listed above, the function `traincomputer(strategy)` should produce a good strategy after about $15,000$ games have been played. Changing the parameters (the 90 in the previous part, and the 29 and 11 in this part) will adjust the speed at which the strategy approaches a good strategy. Feel free to adjust these parameters to experiment, and see if you can get a good strategy in less than $15,000$ games.