# Homework 03

**Brown University**

**DATA 1010**

**Fall 2020**

## Problem 1

In this problem we'll explore the Lagrange multipliers method used to discover the extrema of a smooth function $f$ whose domain is a strict subset of $\mathbb{R}^n$.

(a) Write down conditions on the first derivative (the gradient) of $f$ and conditions on the second derivative (the Hessian) of $f$ that must be satisfied at any local extremum of $f$, and explain briefly why they are necessarily satisfied. (Your written solution may follow the presentation in DG, but it should be in your own words.) You may assume that the boundary of the domain of $f$ is the 0-level set of a differentiable function $g$ with nonvanishing gradient.

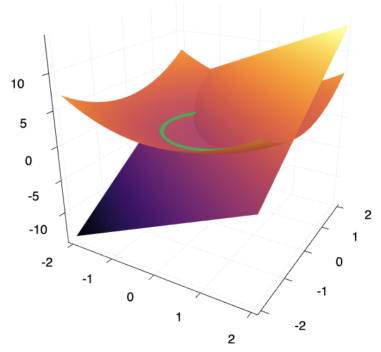
*Hint: The second-order condition at the boundary will require some new thoughts. You'll need to think about how the Hessian specifies which directions you can go from a critical point to increase/decrease the function.*

In the standard derivation, the multiplier $\lambda$ arises as an equation-solving convenience. However, it can be given an interpretation in the context of the constrained optimization problem.

(b) Using the example $f(\mathbf{x}) = 3x_1 + 4x_2$ and $g(\mathbf{x}) = x_1^2 + x_2^2 - 1$, find the extrema and the values of $\lambda$ at those points.

(c) At this point, find the gradient of both $f(x)$ and $g(x)$, are these two gradients linearly dependent? If so, what is the multiplier relating these two gradients? Describe what $\lambda$ represents. (Hint: think about replacing the constraint equation $g(x) = 0$ with $g(x) = c$ for some small value of $c$.)

Here's an interactive graph to help you visualize what's going on:



In [9]:

```
using Plots
plotlyjs()
f(x) = 3x[1] + 4x[2]
g(x) = x[1]^2 + x[2]^2 - 1
surface(-2:0.1:2, -2:0.1:2, (x₁,x₂) -> f([x₁,x₂]))
surface!(-2:0.1:2, -2:0.1:2, (x₁,x₂) -> g([x₁,x₂]))
θs = LinRange(0, 2π, 100)
path3d!(cos.(θs), sin.(θs), fill(0.1,length(θs)), linewidth = 8)
```

```
[ Info: Precompiling Plots [91a5bcdd-55d7-5caf-9e0b-520d859cae80]
[ @ Base loading.jl:1278
[ Info: Precompiling PlotlyJS [f0f68f2c-4968-5e81-91da-67840de0976a]
[ @ Base loading.jl:1278
[ Warning: `@get!(dict, key, default)` at /home/enminz/.julia/packages/WebIO/nTMDV/src/scope.jl:160 is deprecated, use `get!(()->default, dict, key)` instead.
[ @ Base deprecated.jl:204
[ Warning: `@get!(dict, key, default)` at /home/enminz/.julia/packages/WebIO/nTMDV/src/scope.jl:357 is deprecated, use `get!(()->default, dict, key)` instead.
[ @ Base deprecated.jl:204
```

**Unable to load WebIO. Please make sure WebIO works for your Jupyter client. For troubleshooting, please see [the WebIO/IJulia documentation (https://juliagizmos.github.io/WebIO.jl/latest/providers/ijulia/)](https://juliagizmos.github.io/WebIO.jl/latest/providers/ijulia/).**

Out[9]:

(a) The gradient of $f$ must be zero because the gradient shows the increase or decrease direction of a point $p$ in the domain $f$. If gradient is nonzero, then there must exist a direction where $f$ increases or decreases so that this point is not a local extremum. And the hessian of $f$ at point $p$ must also be positive definite or negative definite to be a local extremum. If $p$ is a critical point and hessian of $f$ at $p$ is indefinite, then $f$ is a saddle point that increases along some directions but decreases along others. If $f$ is bounded, then we only need to consider directions that are allowed. In this case, we can ignore part of the hessian matrix to decide whether the rest of the hessian matrix is postive definite of negative definite to find the local extremum.

(b)

In [6]:
```
using SymPy, ForwardDiff
f(x, y) = 3*x + 4*y
g(x, y) = x^2 + y^2-1
L(x, y, l) = f(x,y) - l*g(x,y)
@vars x y l
dx = diff(L(x,y,l), x)
dy = diff(L(x,y,l), y)
dl = diff(L(x,y,l), l)
cp = solve([dx,dy,dl], [x, y, l])
```

Out[6]:
```
2-element Array{Tuple{Sym,Sym,Sym},1}:
 (-3/5, -4/5, -5/2)
 (3/5, 4/5, 5/2)
```

In [49]:
```
using ForwardDiff: hessian
cp1 = [-3/5, -4/5, -5/2]
cp2 = [3/5, 4/5, 5/2]
H1 = hessian(L, cp1)
H2 = hessian(L, cp2)
H1, H2
```

Out[49]:
```
([5.0 0.0 1.2; 0.0 5.0 1.6; 1.2 1.6 0.0], [-5.0 0.0 -1.2; 0.0 -5.0 -1.6; -1.2 -1.6 0.0])
```

(c) For point $[-\frac{3}{5}, -\frac{4}{5}]$, the gradient of f is $[3, 4]$, the gradient of g is $[-\frac{6}{5}, -\frac{8}{5}]$; for point $[\frac{3}{5}, \frac{4}{5}]$, the gradient of f is $[3, 4]$, the gradient of g is $[\frac{6}{5}, \frac{8}{5}]$. At both points, the gradient of f and g are linearly dependent. $\lambda$ represents the coefficients that we need to multiply to the gradient of g to get the gradient of f. $\lambda$ means how much the local extrema chagnes if we change c in the constraint g(x) = c.

In [ ]:
```
1
```

## Problem 2

We would like to examine the details of matrix differentiation in this problem. Let's say that $\mathbf{x} = [x_1, x_2, x_3]$ and

$$A = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

(a) Given an $\mathbb{R}^3$-valued function $\mathbf{a}$ of a vector $\mathbf{x}$ in $\mathbb{R}^3$, recall that the derivative of $\mathbf{a}$ with respect to $\mathbf{x}$ is

$$\frac{\partial \mathbf{a}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial a_1}{\partial x_1} & \frac{\partial a_1}{\partial x_2} & \frac{\partial a_1}{\partial x_3} \\ \frac{\partial a_2}{\partial x_1} & \frac{\partial a_2}{\partial x_2} & \frac{\partial a_2}{\partial x_3} \\ \frac{\partial a_3}{\partial x_1} & \frac{\partial a_3}{\partial x_2} & \frac{\partial a_3}{\partial x_3} \end{pmatrix}$$

where $\mathbf{a} = [a_1, a_2, a_3]$, $\mathbf{x} = [x_1, x_2, x_3]$.

Using this definition, find $\frac{\partial (A\mathbf{x})}{\partial \mathbf{x}}$ and show that it is equal to the matrix $A$.

(b) Similarly, the derivative of a real-valued function with respect to a vector in $\mathbb{R}^3$ is

$$\frac{\partial a}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial a}{\partial x_1} & \frac{\partial a}{\partial x_2} & \frac{\partial a}{\partial x_3} \end{bmatrix},$$

where $\mathbf{x} = (x_1, x_2, x_3)$

Using this definition, find $\mathbf{x}' A \mathbf{x}$, then find $\frac{\partial \mathbf{x}' A \mathbf{x}}{\partial \mathbf{x}}$, and show that it is equal to $\mathbf{x}'(A + A')$.

(a) Ax is $[x_1 + 2x_2, 2x_1 + x_2, x_2 + x_3]$ $\frac{\partial(A\mathbf{x})}{\partial \mathbf{x}}$ is to get the partial differentialtion of each equation in Ax, which is

$$\begin{bmatrix} 1 & 2 & 0 \\ 2 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

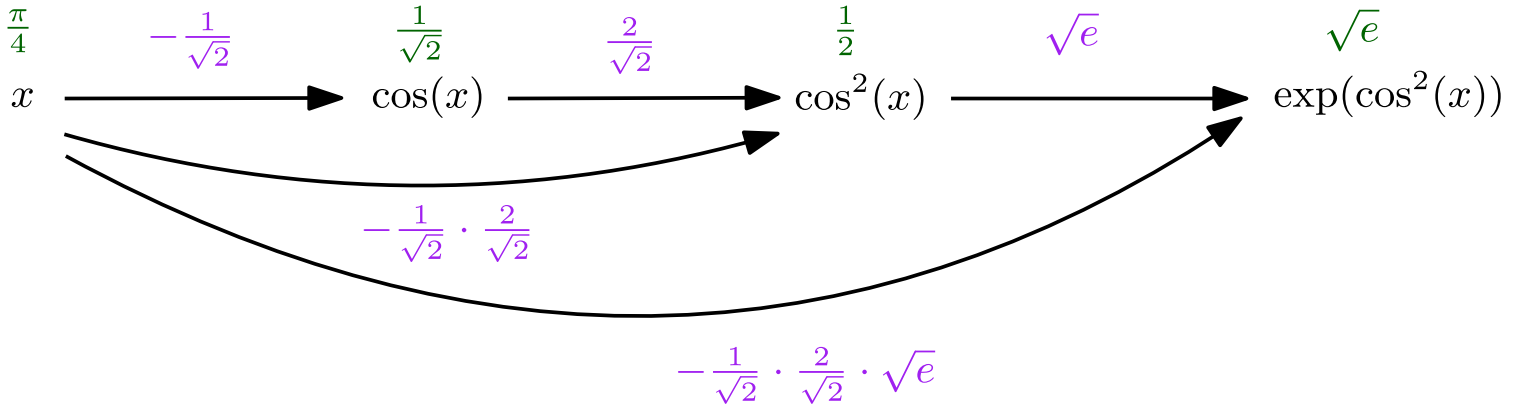
(b) $x'Ax = x_1^2 + x_2^2 + x_3^2 + 4x_1x_2 + x_2x_3$ Differentiate x'Ax with respect to x, we get $[2x_1 + 4x_2, 2x_2 + 4x_1 + x_3, 2x_3 + x_2]$ If we calculate A+A', we get

$$\begin{bmatrix} 2 & 4 & 0 \\ 4 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}$$

, then $x'(A + A'$ is same as the matrix above, so they are equal.

## Problem 3

In class, we differentiated the function $x \mapsto \exp(\cos^2(x))$ at the point $x = \pi/4$ "autodiff-style", meaning that we substituted as soon as possible in our function evaluations and derivative calculations, so that we could have nothing but functions and numbers the whole way through. In other words, we avoided having to represent any symbolic expressions in the computation. This is what it looks like drawn out in a diagram, with derivative values in purple and function values in green:



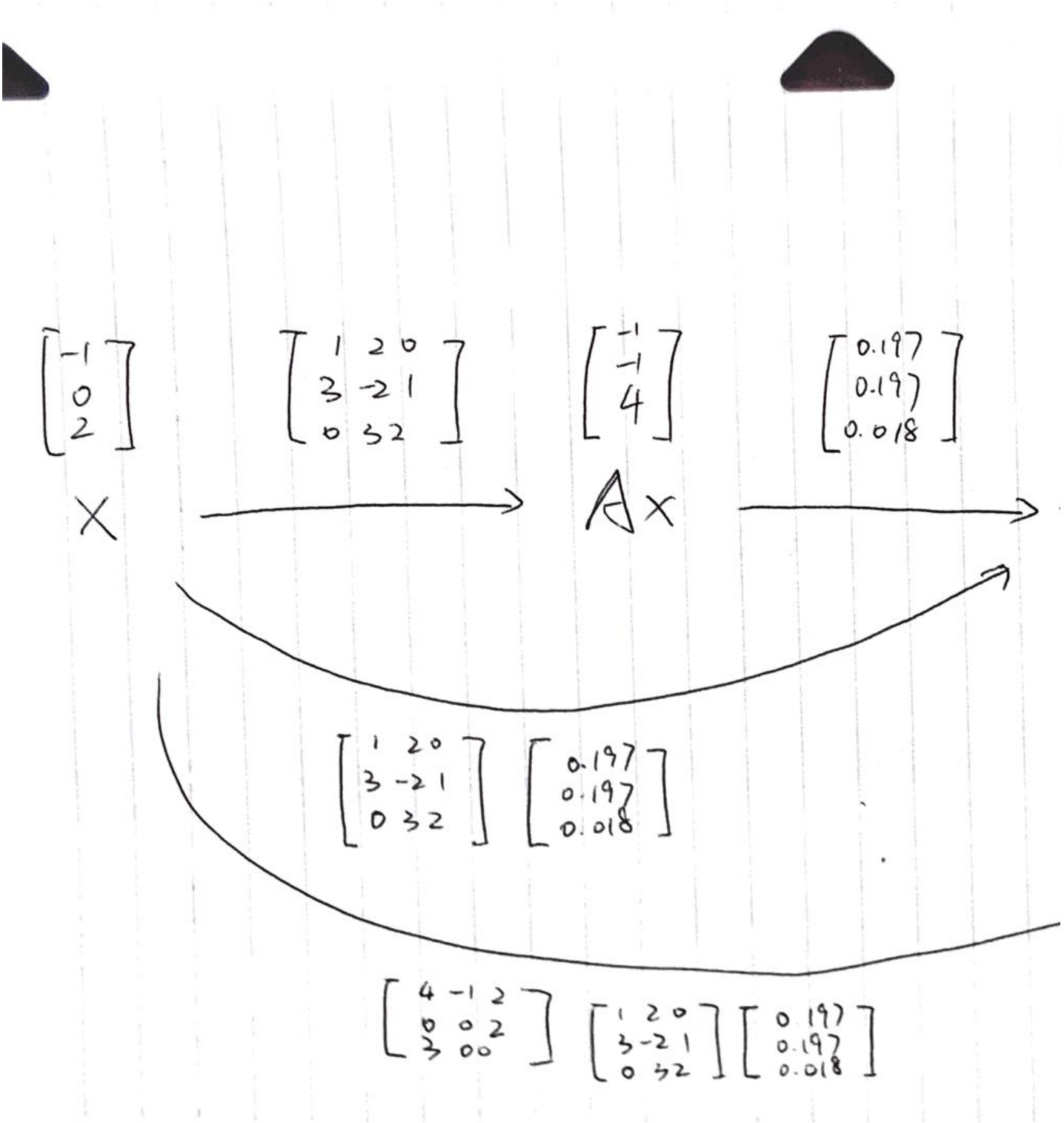

In this problem, we're going to do the same thing but with matrix derivatives in place of the single-variable derivatives.

Consider the function $f(\mathbf{x}) = [\sigma(x_1), \sigma(x_2), \sigma(x_3)]$, where $\sigma(x) = \frac{1}{1+\exp(-x)}$. Let $A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & -2 & 1 \\ 0 & 3 & 2 \end{bmatrix}$ and $B = \begin{bmatrix} 4 & -1 & 2 \\ 0 & 0 & 2 \\ 3 & 0 & 0 \end{bmatrix}$. Differentiate the function $\mathbf{x} \mapsto Bf(A\mathbf{x})$ with respect to $\mathbf{x}$ at the point $\mathbf{x} = [-1, 0, 2]$, using a diagram as similar as possible to the one above. Actually, it should be exactly the same, mutatis mutandis (https://en.wikipedia.org/wiki/Mutatis_mutandis).

Notes:

1. The function we're differentiating here is a composition of the functions "multiply by $B$", f, and "multiply by $A$".
2. To make life easier, feel free to take the equation $\frac{d\sigma}{dx} = \frac{e^{-x}}{(1+e^{-x})^2}$ as given.
3. This function is not only related to neural networks, it **is** a neural network. Differentiating (specifically using this autodifferentiation technique) is how we train neural networks.
4. Also, feel free to evaluate each expression numerically or exactly, as you prefer.

5. Here's some code to help get you started:


X
Ax

f(Ax)
0.269
0.269
0.982
0.596
0.214
0.625
Bf(Ax)
2.771
1.964
0.80

In [1]:
```
A = [1 2 0; 3 -2 1; 0 3 2]
B = [4 -1 2; 0 0 2; 3 0 0]
x = [-1, 0, 2]
# σ(x) = 1/(1 + 1/sympy.exp(-x)) # exact approach
# σ(x) = 1/(1 + 1/exp(-x))  # numerical approach
f(x) = 1/(1+exp(-x))
df(x) = exp(-x) / (1+exp(-x))^2
println(x)
println(A*x)
println(f.(A*x))
println(B*f.(A*x))
println(df.(A*x))
println(A*df.(A*x))
println(B*A*df.(A*x))
```

```
[-1, 0, 2]
[-1, -1, 4]
[0.2689414213699951, 0.2689414213699951, 0.9820137900379085]
[2.770851844185802, 1.964027580075817, 0.8068242641099853]
[0.19661193324148185, 0.19661193324148185, 0.017662706213291114]
[0.5898357997244456, 0.21427463945477296, 0.6251612121510278]
[3.395390983745065, 1.2503224243020556, 1.7695073991733365]
```

## Problem 4

Given a square matrix $A$, its matrix exponential $\exp(A)$ is defined to be $I + A + \frac{1}{2}A^2 + \frac{1}{6}A^3 + \cdots$. In other languages, the function `exp` exponentiates a matrix entry-by-entry, while the matrix exponential is a different function, like `expm`. In Julia, `exp` computes the matrix exponential, while `exp.` is the component-wise version:

In [42]:
```
A = [1 2; 3 4]
println(exp(A))
println(exp.(A))
factorial(2)
```

```
[51.968956198705044 74.73656456700328; 112.10484685050491 164.07380304920997]
[2.718281828459045 7.38905609893065; 20.085536923187668 54.598150033144236]
```

Out[42]: 2

(a) Numerically investigate the claim that the derivative of $\exp(tA)$ with respect to $t \in \mathbb{R}$ is $A\exp(tA)$. Or should it be $\exp(tA)A$?

Note 1: the derivative of a matix-valued function of a real variable is defined entry-by-entry.

Note 2: "Numerically investigate" just means "make up a few small examples and see if the results you get for those examples make sense". You can use difference quotients to approximate derivatives. Even though that method loses a lot of precision, it's fine because you're just doing an approximate check anyway.

In [43]:
```
using LinearAlgebra
h = 0.001
df(t) = (exp((t+h)*A) - exp(t*A))/ h
t = 5
println(df(t))
println(A* exp(t*A))
println(exp(t*A)*A)
```

```
[5.960362825743408e11 8.686794354302979e11; 1.3030191531453247e12 1.8990554357196655e12]
[5.944366788168833e11 8.66348129552458e11; 1.299522194328687e12 1.8939588731455706e12]
[5.944366788168833e11 8.663481295524578e11; 1.299522194328687e12 1.8939588731455706e12]
```

we can see that these 3 values are nearly the same, so derivative of $\exp(tA)$ is $t \in \mathbb{R}$ is $A\exp(tA)$ and is also $\exp(tA)A$

(b) Numerically investigate the claim that if $A$ is diagonalizable as $VDV^{-1}$, then the matrix exponential can be calcluated as $V\exp.(D)V^{-1}$. In other words, the idea is that you can matrix exponentiate by diagonalizing and applying the exponential function entry-wise to the diagonal matrix.

*Solution.*

In [75]:
```
D = Diagonal(eigvals(A))
V = eigvecs(A)
println(exp(A))
println(V*(exp(D))*inv(V))
```

```
[51.968956198705044 74.73656456700328; 112.10484685050491 164.07380304920997]
[51.96895619870499 74.73656456700319; 112.10484685050479 164.0738030492098]
```

## Problem 5

In this problem, we're going to look at a couple virtues of having the `Int8` and `Int64` data types "wrap around" from $2^n - 1$ to $-2^n$ (for $n = 8$ and $n = 64$, respectively).

(a) Use ordinary, by-hand arithmetic to add the binary numbers `00011011` and `00000101`.

*Hint: this is exactly like decimal addition, with place-value and carrying and all that, but with 2 in place of 10.*

00010000

(b) Now apply the same, by-hand algorithm to add the Int8 numbers `00000011` and `11111001`. Does the algorithm yield the correct result, despite the second number being negative?

11111001 Because the the first number is 3 and second number is -7 and the adding result is 4, we have the correct result. There will be incorrect results if overflow happens but not in this case.

(c) Julia, like many languages, includes *bitshift* operators. As the name suggests, these operators shift a value's underlying bits over a specified number of positions:

In [4]:
```
twentyone = parse(Int8, "21")
bitstring(twentyone)
```

Out[4]: `"00010101"`

In [5]:
```
bitstring(twentyone >>> 1)
```

Out[5]: "00001010"

In [6]:
```
bitstring(twentyone >>> -1)
```

Out[6]: "00101010"

We can use the bitshift operator to calculate the midpoint between two integers `lo` and `hi`:

In [63]:
```
lo = 2
hi = 14
(lo + hi) >>> 1
println(lo+hi)
println(bitstring(lo))
println(bitstring(hi))
```

```
16
0000000000000000000000000000000000000000000000000000000000000010
0000000000000000000000000000000000000000000000000000000000001110
```

(d) Explain why the formula `(lo + hi) >>> 1` gives the midpoint between `lo` and `hi`.

In this case, both lo and hi are positive numbers. Move the bits to the right means that we decrease the number to half of it. Therefore, the above expression acutually calculates $\frac{lo+hi}{2}$ which generates the average of two numbers, which is the midpoint.

(e) Show that if `n` is equal to `2^62`, this formula gives the correct midpoint between `n` and `2n`, even though the intermediate value `2n` is actually *negative* in the Int64 system. Show also that the more straightforward formula `(n + 2n) ÷ 2` gives a mathematically incorrect result in this overflow context.

In [77]:
```
n = 2^62
println((n+2n) >>> 1)
println(bitstring(2n))
println((n+2n) ÷ 2)
```

```
6917529027641081856
1000000000000000000000000000000000000000000000000000000000000000
-2305843009213693952
```

2n is a negative number(smallest) in int64 so that n+2n is also negative and the divide operator divide it by 2 which still remain an incorrect result. On the other hand, since the shift operation moves the bit so that the negative number becomes the positive number because a zero is added to the leftmost bit, we still get a midpoint of n and 2n.

## Problem 6

Consider the following PRNG (which was actually widely used in the early 1970s): we begin with an odd positive integer $a_1$ less than $2^{31}$ and for all $n \geq 2$, we define $a_n$ to be the remainder when dividing $65539a_{n-1}$ by $2^{31}$.

Use Julia to calculate $9a_{3n+1} - 6a_{3n+2} + a_{3n+3}$ for the first $10^6$ values of $n$, and show that there are only $15$ unique values in the resulting list (!). Based on this observation, describe what you would see if you plotted many points of the form $(a_{3n+1}, a_{3n+2}, a_{3n+3})$ in three-dimensional space.

In [8]:
```
seed = 9
A = [seed]
for i = 1:3*10^6 + 2
    push!(A, mod(65539*A[end], 2^31))
end
length(Set([sum([9,-6,1].*A[3n+1:3n+3]) for n=0:10^6]))
# the last expression should return 15
```

Out[8]: 15

**This problem is purely optional:**

---

## Bonus Problem

We have learned the representation of numbers with the `Int64` and `Float64` format. We will look a bit more into how one can implement calculations at the machine level.

(a) Calculate the sum of the Int8 value with bitstring $00110101$ and the Int8 value with bitstring $b = 00011011$.

(b) Descibe an algorithm for multiplying two `Int8` numbers. *Hint: You will want to multiply digit by digit, similar to hand-multiplication in base 10.*

(c) Now we want to add the two `Int8` numbers $a = 01110010$ and $b = 00101011$, first, please convert these numbers to decimal numbers and calculate the correct sum.

(d) Using `Julia` and the builtin `Int8()` function, calculate the sum described in (c). What do you find? Can you provide an explaination for this behavior?

Now we will consider two `Float64` numbers. We would like to look at one way of implementing addition of `Float64` numbers, described below. For the sake of simplicity, we shall assume that we are only adding positive numbers and they do not exceed $2^{1024}$. Remember that for every `Float64` number, we have an exponent value $e$ that ranges from $0$ to $2^{11} - 1$, and a mantissa value $f$ that ranges from $0$ to $2^{52} - 1$.

Say we have now a new data type `intinf` , the rules are:

- Every digit is either 0 or 1, there is a "decimal point" and an unlimited number of digits on both sides of the "decimal point".
- If the $n^{th}$ digit above the "decimal point" is a $1$, it represents $2^{n-1}$
- If the $n^{th}$ digit below the "decimal point" is a $1$, it represents $2^{-n}$
  If you are familiar with radix points in binary numbers, this is exactly that. For example, $110.01_{[intinf]} = 1 * 2^2 + 1 * 2^1 + 1 * 2^{-2} = 6.25$.

(e) What is $100.101_{[intinf]}$? What about $10.001_{[intinf]}$? What is $100.101_{[intinf]} + 10.001_{[intinf]}$?

(f) Given the $e$ and $f$ of a `Float64` number, please represent that number in `intinf` format. You will need to use the symbols `>>` and `<<` . `a >> x` means to move the digits of `a` right by `x` spaces while holding the "decimal point" still. E.g. `1.0 >> 2 = 0.01` , `0.101 << 2 = 10.1`

(g) To add two `Float64` numbers together, we will first convert them into `intinf` numbers, then add them together, and finally convert the sum back into `Float64` . With this process in mind, please write down the specific steps of adding two `Float64` numbers represented by $a = (e_a, f_a)$ and $b = (e_b, f_b)$. Your final answer should be another `Float64` number. Please give explaination to all your procedures.
You are not required to use mathematical representations from start to finish, feel free to explain in words where necessary.

Bonus: How would you implement multiplication of two `Float64` numbers? Again, please give sufficient reasoning and/or steps of calculation where necessary. (This is not required and will not affect your grade.)

In [2]:
```
parse(Int8, "00110101", base=2) + parse(Int8, "00011011", base=2)
```

Out[2]: 80

(b) Multipling bitstring a to b actually shifts b to the left by $n - 1$ digits if there is a 1 on the $nth$ digit of a, and add up all the shfited b's together to get the product of a * b.

(c) a is 114 and b is 43 and a + b is 157

(d) we find that the two numbers add up to -99, which is incorrect. It happens because the bitstring add up to the leftmost postion which changes the sign bit so that the number becomes negative. It is an overflow problem as Int8 can only represent from -128 to 127 but 157 is out of range.

In [4]:
```
parse(Int8, "01110010", base=2) + parse(Int8, "00101011", base=2)
```

Out[4]:
```
-99
```

(e) $100.101_{[intif]} = 2^2 + 2^{-1} + 2^{-3} = 4.625$
$10.001_{[intinf]} = 2 + 2^{-3} = 2.125$
$100.101_{[intinf]} + 10.001_{[intinf]} = 2^2 + 2 + 2^{-1} + 2^{-2} = 6.75$

(f) I convert the Float64 to intinf number using the following function which moves the bits of mantissa according to the bits in exponent with pivot at the decimal point.

In [41]:
```
function convert(e)
    float_e = [parse(Int, s) for s in e]
    exp = float_e[2:12]
    m = 0
    for i in 1:length(exp)
        if Int(exp[i]) == 1
            m += 2^(length(exp)-i)
        end
    end
    m = m - 1023
    if m < 0
        decimal_left =  [0]
        decimal_right = vcat([0 for i = 1:(abs(m)-1)], vcat([1], float_e[13:64]))
    elseif m ≤ 52
        decimal_left = vcat([1], float_e[13:m-1])
        decimal_right = float_e[13+m:64]
    else
        decimal_left = vcat([1], vcat(float_e[13:64], [0 for i = 1:m-51]))
        decimal_right = [0]
    end
    str_left = [string(i) for i in decimal_left]
    str_right = [string(i) for i in decimal_right]
    inf_str = join(vcat(str_left, '.', str_right))
    println(inf_str)
end
println(bitstring(1.0))
convert(bitstring(1.0))
```

```
0011111111110000000000000000000000000000000000000000000000000000
1.0000000000000000000000000000000000000000000000000000
```

(g) Firstly, we need to convert the Float64 to intinf using function 'convert' in (f); secondly, we add the intinf the same way as we add the Integers. Lastly, we convert the sum back to Float64. With $a = (e_a, f_a)$ and $b = (e_b, f_b)$, $a_{[intinf]} = (1 + f_a) << e_a$ and $b_{[intinf]} = (1 + f_b) << e_b$. Then we add $a_{[intinf]}$ and $b_{[intinf]}$ to get the sum. And we transfer both of them back to Float64.

In [ ]:
```

```