1940223_2022-03-22 (linear diophantine equations)

March 28, 2022

AIM: Creating a function to find the general solution of a given linear Diophantine equation.

1 Computationally efficient code

[10]:
```
def gcd(s, t):
    r = s % t
    if r == 0: return t
    return gcd(t, r)

from sympy import Symbol
def linDiophantine_computationallyEfficientCode(a, b, c):
    d = gcd(a, b)
    #====================================
# PARTICULAR SOLUTION
#------------
# If d = gcd(a, b) does not divide b...
    if  c % d != 0:
        print("No integer solutions!")
        return
#------------
# If d = gcd(a, b) divides b...
    x0 = 0
    while True: # Solution is guaranteed
        tmp = a*x0 - c
        if tmp % b == 0: break
        x0 = x0 + 1
    """
    NOTE:
    ax + by = c => ax - c = -by => b | (ax - c)
    Hence, we iterate for x = 0, 1, 2...
    If gcd(a, b) | c, solution is guaranteed, even positive solution.
    """

    y0 = -(a*x0 - c)/b
#====================================
# GENERAL SOLUTION
#------------
```

```
    t = Symbol('t')
    x = x0 + (b/d)*t
    y = y0 - (a/d)*t
#====================================
# RETURN VALUE
#------------
    return {'x': x, 'y': y}
```

[11]:
```
def inputInteger(prompt):
    while True:
        try: return int(input(prompt))
        except: print("Please enter integer values!")
#====================================
print("For ax + by = c...")
a = inputInteger("a = ")
b = inputInteger("b = ")
c = inputInteger("c = ")
#====================================
print(linDiophantine_computationallyEfficientCode(a, b, c))
```

```
For ax + by = c…
a = 18
b = 42
c = 30
{'x': 7.0*t + 4, 'y': -3.0*t - 1.0}
```

2 Simulating handwritten method

Computationally, the following method is far less efficient and far more complicated that the above code. However, it was an interesting challenge to simulate the method we use to solve linear Diophantine equations manually… Plus, I didn't think about the above (undisputably better) code myself, instead coming up originally with the clunky, unnecessarily complicated option :'(

2.1 Helper functions

[5]:
```
# Finding the GCD of two numbers
def gcd(s, t):
    r = s % t
    if r == 0: return t
    return gcd(t, r)
#====================================

"""
NOTES ON THE FOLLOWING FUNCTIONS
------------------------
The following functions are used to calculate the particular solution
of the linear Diophantine equation ax + by = c. The aim of these functions is to
```

```python
obtain an expression for gcd(a, b) in terms of ak + bh. Hence, you get
d = ak + bh; where d = gcd(a, b)
=> d*(c/d) = c = a(k*c/d) + b(h*c/d)
Hence, you can obtain a particular solution for x and y as
x = k*c/d, y = h*c/d

I have used s and t (instead of a and b) to visually separate
the following functions from the context of the linear Diophantine equation form,
since these functions can be used in a variety of contexts in different ways.
For example, they can be used for solving the linear congruence ax   b(mod n),
where we get the linear Diophantine equation ax + ny = b.
"""

# Euclidean algorithm simulator
def euclidalg(s, t, alias):
# 'alias' allows for customised variable naming in the expressions
    q, r, Q, R = 1, 1, ['', ''], [alias[0], alias[1]]

    """
    NOTES
    This function returns the quotients and remainders of the Euclidean algorithm process.
    The expression making algorithm considers every dividend as a previous remainder.
    Hence, we include s and t at the start of the remainders list to enable it.
    We don't include the values, only the character, so we can factorize it easily.
    """

    while r > 0:
        q, r = s // t, s % t
        s, t = t, r
        Q.append(q)
        R.append(r)
    return (Q, R)

# Finds the expression for each remainder in the Euclidean algorithm
def euclidalgRemainderExpressions(s, t, alias):
    (Q, R) = euclidalg(s, t, alias)
    E = []
    i = len(R) - 2
    while i > 1:
        E.insert(0, [str(R[i]), str(R[i-2])+'-'+str(Q[i])+"*"+str(R[i-1])])
# Each element is a tuple containing the remainder and its expression.
        i = i - 1
    return E
```

```python
# Returning the remainders as well, for reference.
# They are reversed to match the expression orders.

# Obtain gcd(s, t) as sk + th
from sympy import simplify, Symbol
def gcdExpression(s, t, alias):
    E = euclidalgRemainderExpressions(s, t, alias)
    for i in range(1, len(E)): # Skipping the first element
        E[i][1] = E[i][1][:-len(E[i-1][0])] + "(" + E[i-1][1] + ")"

    """
    NOTES
    E[i-1][0] denotes the previous remainder.
    The current remainder's expression contains this.
    The code was designed so this previous remainder appears at the end.
    The length of this remainder is given by len(E[i-1][0]).
    To exclude this remainder, we do E[i][1][:-len(E[i-1][0])].
    """

# Obtaining the expression for the GCD
    try: E[-1][1] = E[-3][1] + E[-1][1][len(E[-3][0]):]
    except: pass

    """
    NOTES
    Based on the structure of Euclidean algorithm for s and t...
    s = q_1*t + r_1
    t = q_2*r_1 + r_2
    r_1 = q_3*r_2 + r_3
    ...
    r_(n-3) = q_(n-1)*r_(n-2) + r_(n-1)
    r_(n-2) = q_(n)*r_(n-1) + r_n
    r_(n-1) = q_(n+1)*r_n + 0

    Hence, we get the last non-zero remainder r_n (also the GCD of the s and t)␣
↪as
    r_n = r_(n-2) - q_(n+1)•r_(n-1)
    Now, note that in our loop, we only replace the tail remainder with its␣
↪expression.
    But for the GCD's expression, we need to replace the head remainder as well.
    The head remainder of the GCD's expression is E[-3][1]
    Hence, we do E[-1][1] = E[-3][1] + E[-1][1][len(E[-3][0]):].

    We use a try except block because sometimes this substitution is not␣
↪necessary.
    """
```

```
# Obtaining the expression for gcd(s, t) in the form sk + th
    expression = E[-1][1]
    return simplify(expression)
# Last expression equals gcd(s, t)
```

Demonstrations of the above functions...

[6]:
```
s, t = 12, 7

print("\ngcd(s, t):")
print(gcd(s, t))

print("\neuclidalg(s, t):")
print(euclidalg(s, t, ['s', 't']))

print("\neuclidalgRemainderExpressions(s, t):")
print(euclidalgRemainderExpressions(s, t, ['s', 't']))

print("\ngcdExpression(s, t)")
print(gcdExpression(s, t, ['s', 't']))
```

```
gcd(s, t):
1

euclidalg(s, t):
(['', '', 1, 1, 2, 2], ['s', 't', 5, 2, 1, 0])

euclidalgRemainderExpressions(s, t):
[['5', 's-1*t'], ['2', 't-1*5'], ['1', '5-2*2']]

gcdExpression(s, t)
3*s - 5*t
```

2.2 Main function

[1]:
```
# Particular and general solution of a linear Diophantine equation ax + by = c
def linDiophantine_handWrittenMethodSimulation(a, b, c):
    d = gcd(a, b)
#====================================
# PARTICULAR SOLUTION
#------------
# If d = gcd(a, b) does not divide b...
    if  c % d != 0:
        print("No integer solutions!")
        return
#------------
# If d = gcd(a, b) divides b...
```

```
    expression = gcdExpression(a, b, ['a', 'b'])
    """
    NOTES
    The above expression is gcd(a, b) = ak + bh = c.
    Hence, to obtain the solution for ax + by = c, we do
    a(k*c/d) + b(h*c/d) = d * (c/d)
    """
    expression = expression*c/d
#------------
# Isolating the coefficient
    _a, _b = Symbol('a'), Symbol('b')
    x0 = int(expression.subs({_a:1, _b:0}))
    y0 = int(expression.subs({_a:0, _b:1}))
#====================================
# GENERAL SOLUTION
#------------
    t = Symbol('t')
    x = x0 + (b/d)*t
    y = y0 - (a/d)*t
#====================================
# RETURN VALUE
#------------
    return {'x': x, 'y': y}
```

2.3 Application

[8]:
```
def inputInteger(prompt):
    while True:
        try: return int(input(prompt))
        except: print("Please enter integer values!")
#====================================
print("For ax + by = c...")
a = inputInteger("a = ")
b = inputInteger("b = ")
c = inputInteger("c = ")
#====================================
print(linDiophantine_handWrittenMethodSimulation(a, b, c))
```

```
For ax + by = c…
a = 18
b = 42
c = 30
{'x': 7.0*t - 10, 'y': 5 - 3.0*t}
```