1. Питання 1

1.1. Метод оберненої функції

Генерування випадкової величини r за заданим законом розподілу F(x):

$r = F^{-1}(\varsigma)$.

Експоненціальний закон розподілу: $r = -1\frac{1}{\lambda} ln\varsigma$

Працює так: генеруємо випадкове число ς з рівномірного розподілу [0,1], потім застосовуємо до нього обернену функцію розподілу

Обмеження:

Потрібно вміти знаходити обернену функції

Не завжди існує аналітичний вираз для F⁻1

Для деяких розподілів обчислення F^{-1} може бути складним

Приклади:

```
# F(x) = 1 - e^(-x²/2σ²), x ≥ 0
def generate_rayleigh(sigma, size=1):
    u = np.random.uniform(0, 1, size)
    return sigma * np.sqrt(-2 * np.log(1 - u))
# F(x) = (1/π)arctan(x) + 1/2
def generate_cauchy(size=1):
    u = np.random.uniform(0, 1, size)
    return np.tan(np.pi * (u - 0.5))
```

1.2. табличний метод

$r = x_{i-1} + \frac{x_i - x_{i-1}}{a_i - a_{i-1}}(\varsigma - a_{i-1})$. $a_i = F(x_i)$. Емпіричний закон розподілу.

x_i - значення випадкової величини

a_i = F(x_i) - значення функції розподілу

ς - випадкове число з рівномірного розподілу [0,1]

Алгоритм методу:

a) Підготовчий етап:

- Створюємо таблицю зі значеннями x_i
- Обчислюємо відповідні значення функції розподілу a_i
- Впорядковуємо дані за зростанням

b) Генерація:

Генеруємо випадкове число ς ∈ [0,1]. Знаходимо інтервал [a_(i-1), a_i], якому належить ς. Застосовуємо формулу лінійної інтерполяції.

1.3. Спеціальні методи

Нормальний закон розподілу: $r = \sigma(\sum_{i=1}^{12}[\varsigma_i - 6]) + a$

Закон розподілу ерланга: $r = -\frac{1}{k\mu} ln(\prod_{i=1}^{k} \varsigma_i)$

2. Питання 2

Аналітичне дослідження властивостей мережі Петрі	Властивість	Спосіб дослідження	
		Матричний підхід	Дерево досяжності
	k-обмеженість	Не досліджується	Необхідна і достатня
	зберігання	Необхідна і достатня умова	Необхідна і достатня
	досяжність	Тільки необхідна умова	Послідовність переходів залишається невідомою
	активність	Не досліджується	Не досліджується

Властивість	Визначення
k-обмеженість	Якщо кількість маркерів в будь-якій позиції мережі Петрі не перевищує k маркерів, то мережа являється k – обмеженою.
досяжність	Досяжністю мережі Петрі називається множина досяжних маркірувань.
збереження	Якщо в мережі Петрі неможливе виникнення і знищення ресурсів, то мережа володіє властивістю збереження.
активність	Якщо з будь-якого досяжного початкового стану можливий перехід в будь-який інший досяжний стан, то мережа Петрі володіє властивістю активності.

2.1. Збереження (консервативність)

Збереження (консервативність) – Якщо існує вектор w, компоненти якого цілі додатні числа, такий що $w^T \cdot M = w^T \cdot M^\bullet$ для будь − якого досяжного з початкового маркування, то мережа Петрі володіє властивістю збереження.

Властивість консервативності мережі Петрі означає, що загальна кількість маркерів (фішок) у мережі залишається незмінною під час її функціонування. Це важлива характеристика, яка має кілька аспектів:

Практичне значення:

- Дозволяє моделювати системи з обмеженими ресурсами
- Гарантує, що ресурси не створюються і не знищуються в системі
- Допомагає виявляти помилки в моделюванні

$$w^T \cdot M = w^T \cdot M^\bullet, \forall M^\bullet (\text{досяжного})$$
$$w^T \cdot M = w^T \cdot (M + a \cdot v), \forall v$$
$$0 = w^T \cdot a \cdot v, \forall v$$
$$0 = w^T \cdot a$$
$$0^T = (w^T \cdot a)^T$$
$$0^T = a^T \cdot w$$

Твердження. Мережа Петрі володіє властивістю зберігання тоді і тільки тоді, коли існує вектор w, компоненти якого цілі додатні числа, такий, що $a^T \cdot w = 0, w_i \in Z_+$.

2.2. S – інваріант мережі Петрі

Розв'язки рівняння $a^T \cdot w = 0$, де w – невідомий вектор розміру $|P| \times 1$, називають S-інваріантом мережі Петрі.

S-інваріант, або інваріант стану, дозволяє досліджувати консервативність системи. Консервативність означає, що існує зважена сума маркувань позицій мережі Петрі, яка для будь-якого досяжного маркування залишається незмінною. Рівняння, які формулюються і розв'язуються в термінах цілих чисел, називають діофантовими.

2.3. Циклічність

Якщо існує послідовність запусків переходів, така що мережа повертається в початкове маркірування, то функціонування мережі Петрі є циклічним.

Формальне визначення:

- Мережа є циклічною, якщо з будь-якого досяжного маркування можна повернутися до початкового маркування M_0
- Для кожного маркування M, досяжного з M_0, існує послідовність переходів, що повертає систему в M_0

$M^{\bullet} = M$

$M + a \cdot v = M, \exists v$

$0 = a \cdot v, \exists v$

Функціонування мережі Петрі є циклічним тоді і тільки тоді, коли існує вектор v, компоненти якого цілі невід'ємні числа, такий, що $a \cdot v = 0, v_i \in Z_+$.

2.4. T – інваріант мережі Петрі

Розв'язки рівняння $a \cdot v = 0$, де v – невідомий вектор розміру $|P| \times 1$, називають T-інваріантом мережі Петрі.

T-інваріант, або інваріант функціонування, означає досяжність початкового маркування. Цей інваріант є важливим для дослідження циклічності процесів функціонування.

Циклічність означає існування такої послідовності запусків переходів, що мережа Петрі повертається в початкове маркірування. Наявність T-інваріантів гарантує циклічність функціонування системи.

2.5. Досяжність

Існування невід'ємного цілого вектора запуску переходів, що задовольняє рівнянню M'= M + a · v, є тільки необхідною, але не достатньою умовою.

2.6. Активність

Рівень активності	Перехід T має рівень активності A, якщо
0	він ніколи не може бути запущений
1	існує маркірування (досяжне з початкового) , яке дозволяє запуск цього переходу T
2	для довільного цілого числа n існує послідовність запусків переходів, в якій перехід T присутній принаймні n раз
3	існує нескінченна послідовність запусків переходів, в якій перехід T присутній необмежено багато разів
4	якщо для довільного маркірування M, що є досяжним з початкового маркірування, існує послідовність запусків переходів, яка призводить до маркірування, що дозволяє запуск переходу T

2.7. Дерево досяжності

Дерево досяжності представляє множину досяжних маркувань мережі Петрі. Дерево досяжності розпочинається з початкового маркування, а закінчується термінальним або дублюючим маркуванням.

Термінальним маркуванням називається маркування, в якому жоден з переходів мережі Петрі не запускається.

Дублюючим маркіруванням називається маркірування, що раніше зустрічалося в дереві досяжності.

Символ ω в позиції Mj маркування M з'являється тоді, коли на шляху до маркування M спостерігається маркування M′, в якому всі значення, крім j-ого, не перевищують значення маркування M, а j-е значення є меншим. Одного разу з'явившись, символ ω уже не змінюється і не зникає в дереві досяжності: додавання або віднімання від нескінченності є нескінченність.

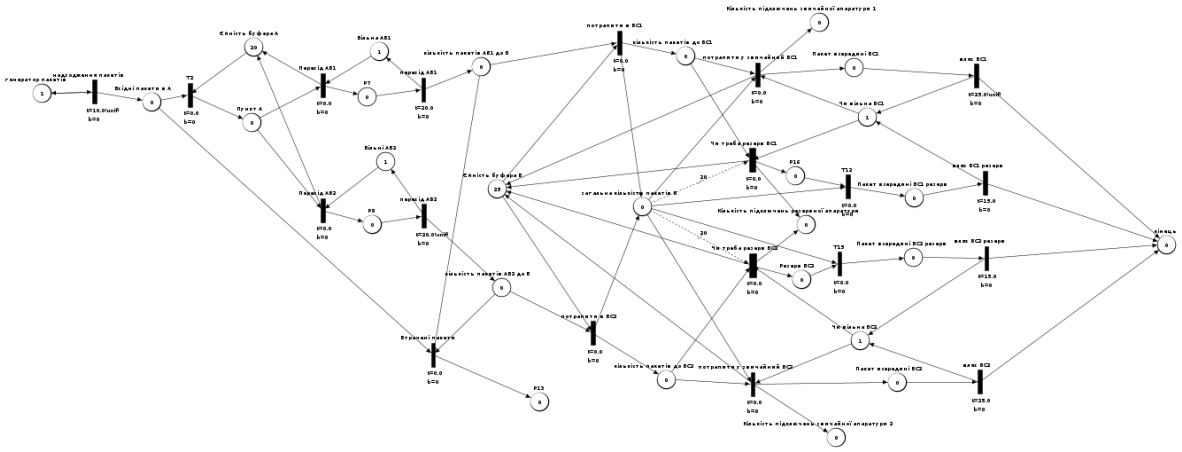
Приклад дослідження дерева досяжності:

```
import attr
from collections.abc import Sequence
import numpy as np
import numpy.typing as npt

@attr.frozen
class Arc:
      place_index: int
      transition_index: int
      weight: int = attr.field(default=1)

@attr.mutable
class TreeNode:
      state: npt.NDArray[np.int64]
      substates: list['TreeNode'] = attr.field(factory=list)

def recurse_tree(
      transition_count: int,
      input_arcs: Sequence[Arc],
      output_arcs: Sequence[Arc],
      tree_node: TreeNode,
      all_states: list[npt.NDArray[np.int64]]
):
      for transiton_index in range(transition_count):
      def filter_by_transiton_index(arc: Arc):
            return arc.transition_index == transiton_index

      transiton_index_input_arcs  =  list(filter(filter_by_transiton_index,
input_arcs))
```

```
      if    all(map(lambda    arc:    tree_node.state[arc.place_index]    >=
arc.weight, transiton_index_input_arcs)):

            new_state = tree_node.state.copy()

            for arc in transiton_index_input_arcs:
                  new_state[arc.place_index] -= arc.weight

            for arc in filter(filter_by_transiton_index, output_arcs):
                  new_state[arc.place_index] += arc.weight

            if all(map(lambda state: not np.array_equal(new_state, state),
all_states)):
                  all_states.append(new_state)
                  new_tree_node = TreeNode(new_state)
                  tree_node.substates.append(new_tree_node)
                  recurse_tree(transition_count,  input_arcs,  output_arcs,
new_tree_node, all_states)

TreeNodeTuple = tuple[npt.NDArray[np.int64], list['TreeNodeTuple']]

def tree_node_to_tuple(tree_node: TreeNode) -> TreeNodeTuple:
      tree_node_tuple: TreeNodeTuple = (tree_node.state, [])
      for substate in tree_node.substates:
      tree_node_tuple[1].append(tree_node_to_tuple(substate))
      return tree_node_tuple

from pprint import pprint
transition_count = 3
root_node = TreeNode(np.array([3, 3, 0, 3]))
input_arcs: list[Arc] = [Arc(0, 0), Arc(1, 0), Arc(2, 1), Arc(2, 2, 2),
Arc(3, 2)]
output_arcs = [Arc(0, 1), Arc(1, 2), Arc(2, 0), Arc(3, 1, 2)]
all_states: list[npt.NDArray[np.int64]] = [root_node.state]
recurse_tree(transition_count,    input_arcs,    output_arcs,    root_node,
all_states)
root_node_tuple = tree_node_to_tuple(root_node)
pprint(root_node_tuple)
(array([3, 3, 0, 3]),
 [(array([2, 2, 1, 3]),
   [(array([1, 1, 2, 3]),
     [(array([0, 0, 3, 3]),
       [(array([1, 0, 2, 5]),
         [(array([2, 0, 1, 7]), [(array([3, 0, 0, 9]), [])]),
          (array([1, 1, 0, 4]),
           [(array([0, 0, 1, 4]), [(array([1, 0, 0, 6]), [])])])]),
        (array([0, 1, 1, 2]), [])]),
      (array([2, 1, 1, 5]), [(array([3, 1, 0, 7]), [])]),
```

```
      (array([1, 2, 0, 2]), [])]),
    (array([3, 2, 0, 5]), [])]))
```

3. Питання 3



4. Питання 4

```
from typing import List, Optional, Literal, Union
import random
from random import choices
import math
import sys

def generate_exponential(time_mean: float) -> float:
        a = 0
        while a == 0:
        a = random.random()
        a = -time_mean * math.log(a)
        return a

def generate_uniform(time_min: float, time_max: float) -> float:
        a = 0
        while a == 0:
        a = random.random()
        a = time_min + a * (time_max - time_min)
        return a

def generate_normal(time_mean: float, time_deviation: float) -> float:
        a = time_mean + time_deviation * random.gauss()
        return a

def generate_erlang(time_mean: float, k: int) -> float:
```

```
        lambda_: float = k / time_mean
        sum_random_values: float = sum(math.log(random.random()) for _ in range(k))
        return (-1 / lambda_) * sum_random_values

class Customer:
        def __init__(self, enter_time: float):
        self.enter_time: float = enter_time

class Route:
        def __init__(self, element: 'Element', p: Optional[float] = None):
        self.element: 'Element' = element
        self.p: Optional[float] = p

class Element:
        next_id: int = 0

        def __init__(self, name_of_element: str = 'anonymous', delay: float = 1.0):
        self.name: str = name_of_element
        self.tnext: float = 0.0
        self.delay_mean: float = delay
        self.distribution: str = 'exp'
        self.tcurr: float = self.tnext
        self.state: int = 0
        self.id_: int = Element.next_id
        Element.next_id += 1
        self.name = 'element' + str(self.id_)
        self.delay_dev: float = 0.0
        self.quantity: int = 0
        self.route_choice: Optional[Literal['probability', 'priority']] = None
        self.routes: Optional[List[Route]] = None
        self.customer: Optional[Customer] = None
        self.customer_presence_time: float = 0

        def get_delay(self) -> float:
        delay: float = self.delay_mean
        if self.distribution.lower() == 'exp':
        delay = generate_exponential(self.delay_mean)
        elif self.distribution.lower() == 'generate_normal':
        delay = generate_normal(self.delay_mean, self.delay_dev)
        elif self.distribution.lower() == 'generate_uniform':
        delay = generate_uniform(self.delay_mean, self.delay_dev)
        return delay

        def get_next_element(self) -> Optional['Element']:
        if self.route_choice is None:
        return None

        if self.route_choice == 'probability':
```

```
        return choices([route.element for route in self.routes],
                [route.p for route in self.routes])[0]
        elif self.route_choice == 'priority':
        routes_by_queue = sorted(self.routes, key=lambda r: r.element.queue)
        return routes_by_queue[0].element

        raise ValueError('Make sure that either "probability" or "priority" is set as the route
choice')

        def in_act(self, customer: Optional[Customer]) -> None:
        pass

        def out_act(self) -> None:
        self.quantity += 1
        if self.customer:
        self.customer_presence_time += (self.tcurr - self.customer.enter_time)

        def set_tcurr(self, tcurr: float) -> None:
        self.tcurr = tcurr

        def print_result(self) -> None:
        print(f'{self.name} quantity = {self.quantity}')

        def print_info(self) -> None:
        print(f'{self.name} state = {self.state} quantity = {self.quantity} tnext = {self.tnext}')

        def do_statistic(self, delta: float) -> None:
        pass

class Create(Element):
        def __init__(self, delay: float):
        super().__init__(delay=delay)
        self.tnext = 0.0
        self.failures: int = 0

        def out_act(self) -> None:
        super().out_act()
        self.tnext = self.tcurr + self.get_delay()
        customer = Customer(self.tcurr)
        route1, route2 = self.routes
        if route1.element.queue + route2.element.queue == route1.element.max_queue +
route2.element.max_queue:
        self.failures += 1
        else:
        if route1.element.queue == route2.element.queue:
                next_element = random.choice([route1.element, route2.element])
        else:
```

```
                next_element = route1.element if route1.element.queue <
route2.element.queue else route2.element
        next_element.in_act(customer)

class Process(Element):
        def __init__(self, delay: float, num_devices: int = 1):
        super().__init__(delay=delay)
        self.devices: List[Element] = [Element(delay=delay) for _ in range(num_devices)]
        for device in self.devices:
        device.tnext = float('inf')
        self.queue: int = 0
        self.max_queue: int = sys.maxsize
        self.mean_queue: float = 0.0
        self.tnext = float('inf')
        self.failure: int = 0
        self.average_load: float = 0
        self.aver_customers: float = 0

        def in_act(self, customer: Optional[Customer] = None) -> None:
        system_busy = True
        for device in self.devices:
        if device.state == 0:
                system_busy = False
                device.state = 1
                device.tnext = self.tcurr + self.get_delay()
                device.customer = customer
                break
        if system_busy:
        if self.queue < self.max_queue:
                self.queue += 1
        else:
        self.tnext = min(device.tnext for device in self.devices)

        def out_act(self) -> None:
        for device in self.devices:
        if self.tcurr >= device.tnext:
                super().out_act()
                device.out_act()
                device.tnext = float('inf')
                device.state = 0

                if self.queue > 0:
                self.queue -= 1
                device.state = 1
                device.tnext = self.tcurr + self.get_delay()
                self.tnext = min(device.tnext for device in self.devices)

                next_route = super().get_next_element()
```

```
                if next_route is not None:
                next_route.in_act(None)

        def is_available(self) -> bool:
        return any(device.state == 0 for device in self.devices)

        def print_info(self) -> None:
        for device in self.devices:
        device.print_info()
        print(f'failure = {self.failure}, queue = {self.queue}')

        def set_tcurr(self, tcurr: float) -> None:
        self.tcurr = tcurr
        for device in self.devices:
        device.tcurr = tcurr

        def do_statistic(self, delta: float) -> None:
        self.mean_queue += self.queue * delta
        self.average_load += delta * self.devices[0].state
        self.aver_customers += delta * (self.devices[0].state + self.queue)

class Model:
        def __init__(self, elements: List[Union[Create, Process]]):
        self.list: List[Union[Create, Process]] = elements
        self.tnext: float = 0.0
        self.event: int = 0
        self.tcurr: float = self.tnext
        self.queue_changes: int = 0

        def simulate(self, time: float) -> None:
        while self.tcurr < time:
        self.tnext = float('inf')
        for index, e in enumerate(self.list):
                if e.tnext < self.tnext:
                self.tnext = e.tnext
                self.event = index

        for e in self.list:
                e.do_statistic(self.tnext - self.tcurr)
        self.tcurr = self.tnext
        for e in self.list:
                e.set_tcurr(self.tcurr)
        self.list[self.event].out_act()
        for e in self.list:
                if e.tnext == self.tcurr:
                e.out_act()
        self.check_queue_change()
        self.print_result()
```

```
        def check_queue_change(self) -> None:
        if self.list[1].queue - self.list[2].queue >= 2:
        self.list[1].queue -= 1
        self.list[2].queue += 1
        self.queue_changes += 1
        elif self.list[2].queue - self.list[1].queue >= 2:
        self.list[2].queue -= 1
        self.list[1].queue += 1
        self.queue_changes += 1

        def print_info(self) -> None:
        for e in self.list:
        e.print_info()

        def print_result(self) -> None:
        print('\n-------------RESULTS-------------')
        for e in self.list:
        e.print_result()
        if isinstance(e, Process):
                print(f'\taverage load = {e.average_load / self.tcurr}')
        print()

        print(f'lost customers percentage = {self.list[0].failures / self.list[0].quantity}')
        print(f'average customer time in bank = '
        f'{(self.list[1].devices[0].customer_presence_time +
self.list[2].devices[0].customer_presence_time) / (self.list[1].quantity + self.list[2].quantity)}')

# Example usage
creator = Create(2.5)
processor1 = Process(1.5)
processor2 = Process(1.5)

creator.route_choice = 'priority'
creator.routes = [Route(processor1), Route(processor2)]

processor1.max_queue = 3
processor2.max_queue = 3

creator.name = 'Clients arrival'
processor1.name = 'Cashier 1'
processor2.name = 'Cashier 2'

creator.distribution = 'exp'
processor1.distribution = 'exp'
processor2.distribution = 'exp'

elements = [creator, processor1, processor2]
```

```
model = Model(elements)
model.simulate(1000)
```