Sample Midterm solutions

1. a) Explain the difference between symbolic computation and numerical computation. Discuss one situation where symbolic computation gives structural insight that numerical computation cannot provide directly!

Machine precision floating-point numbers may not be exact (rounding errors), and accumulated small errors can lead to significant inaccuracies. Some computations require exact results. Symbolic computation allows for analytic solutions and deeper understanding of mathematical relationships. (source: 2nd lecture notes)

Example situation - Factoring an expression:

1. Symbolic: $x^2 + 3x + 2 = (x + 1)(x + 2) \rightarrow$ reveals roots and structure
2. Numerical: evaluating at points only gives approximate values, not the factorization

Thus, symbolic computation reveals structure (like roots, simplifications, identities) that numerical computation cannot directly provide.

b) Explain the difference between exact rational arithmetic and floating-point arithmetic. Why can mathematically equivalent formulas produce different numerical results in floating-point computation?

Floating-point arithmetic can lead to small but significant errors in calculations due to rounding errors. For example: $(1/49) * 49 = 0.999999$ with floating-point arithmetic, while exact rational arithmetic can calculate the exact value (1). (source: 2nd lecture notes)

c) In Python, every iterator is an iterable, but not every iterable is an iterator. Explain this statement and give one example involving generator expressions.

An iterable is an object you can loop over (e.g. a list), while an iterator is an object that actually produces values one at a time using next(), iterables must first return an iterator. For example, a generator expression like (x**2 for x in range(3)) is already an iterator, whereas a list like [x**2 for x in range(3)] is only an iterable and needs iter() before next() can be used. (next() can generate the next element of the iterable object - e.g. go to the next element of the list)

2. Consider the symbolic expression:

$$f(x, y) = \frac{(x + y)^4 - (x - y)^4}{x}.$$

Using SymPy, perform the following tasks:
a) Define the expression symbolically and simplify it.
b) Expand the numerator and determine whether the whole expression can be simplified to a polynomial.
c) Write a Python function that recursively traverses the expression tree and counts how many nodes of type Pow, Add, and Mul occur.
d) Determine the depth of the expression tree.
e) Substitute y = x into the expression before simplification and explain why the order of substitution and simplification matters.

a)

```
import sympy as sp
x, y = sp.symbols('x␣y')
f = ((x + y)**4 - (x - y)**4) / x
sp.simplify(f)
```

b)

```
num = sp.expand((x + y)**4 - (x - y)**4)
sp.expand(f)
```

c)

```
def count_nodes(expr):
    counts = {"Add": 0, "Mul": 0, "Pow": 0}

    if isinstance(expr, sp.Add):
        counts["Add"] += 1
    elif isinstance(expr, sp.Mul):
        counts["Mul"] += 1
    elif isinstance(expr, sp.Pow):
        counts["Pow"] += 1

    for arg in expr.args:
        sub_counts = count_nodes(arg)
        for key in counts:
            counts[key] += sub_counts[key]

    return counts
```

d)

```
def tree_depth(expr):
    if not expr.args:
        return 1
    return 1 + max(tree_depth(arg) for arg in expr.args)

tree_depth(f)
```

e)

```
f_sub = f.subs(y, x)
sp.simplify(f_sub)
```

The order matters because substitution acts on the current structure, and subs() always returns a new expression. Symbolic expressions are structured objects, so if you simplify first, structure may cancel terms differently before substitution; if you substitute first, you may expose cancellations (like division by x) earlier. This affects intermediate forms and sometimes whether simplification succeeds or is straightforward.

3. Solve the following tasks using SymPy.

a) Compute the first and second derivatives of

$$g(x) = x^3 \sin(x).$$

b) Compute the definite integral

$$\int_0^{\pi} x \sin(x)\, dx.$$

c) Compute the Taylor expansion of $\ln(1 + x)$ around $x = 0$ up to degree 6.

d) Solve the differential equation

$$y'' + 3y' - 4y = e^{2x}.$$

e) Verify symbolically that the obtained solution satisfies the equation.

a)

```
import sympy as sp

x = sp.symbols('x')
g = x**3 * sp.sin(x)

g1 = sp.diff(g, x)
g2 = sp.diff(g, x, 2)
```

b)

```
sp.integrate(x*sp.sin(x), (x, 0, sp.pi))
```

c)

```
sp.series(sp.log(1 + x), x, 0, 7)
```

d)

```
y = sp.Function('y')
eq = sp.Eq(y(x).diff(x, 2) + 3*y(x).diff(x) - 4*y(x), sp.exp
    (2*x))

sol = sp.dsolve(eq)
```

e)

```
sp.checkodesol(eq, sol)
```

4. Let

$$p(x) = x^5 - 2x^4 - x^3 + 2x^2, \qquad q(x) = x^4 - 1.$$

Using SymPy, perform the following tasks:

a) Factor both polynomials completely over the rational numbers.

b) Determine whether $p(x)$ is divisible by $q(x)$. Compute quotient and remainder.

c) Compute the greatest common divisor of $p(x)$ and $q(x)$.

d) Compute all roots of $q(x)$, including complex roots.

e) Rewrite $p(x)$ as a polynomial in powers of $(x - 1)$.

Defining the polynomials:

```
x = sp.symbols('x')

p = sp.Poly(x**5 - 2*x**4 - x**3 + 2*x**2, x)
q = sp.Poly(x**4 - 1, x)
```

a)

```
p_factored = sp.factor(p.as_expr())
q_factored = sp.factor(q.as_expr())
```

b)

```
quotient, remainder = sp.div(p, q)
```

c)

```
gcd_pq = sp.gcd(p, q)
```

d)

```
roots_q = sp.roots(q)
```

e)

```
y = sp.symbols('y')
p_shifted = sp.expand(p.as_expr().subs(x, y + 1))
```