

lib

April 9, 2025

```
[51]: def floatfmt(v, prec, exp):  
        return f"{v/10**(exp):0=1.{prec}f}{LibFormatter.exp10(exp) if exp != 0 else_  
        ↪ '}'"  
  
def prec_ceil(v, prec=0):  
    return np.true_divide(np.ceil(v * 10**prec), 10**prec)  
  
def prec_floor(v, prec=0):  
    return np.true_divide(np.floor(v * 10**prec), 10**prec)
```

```
[57]: class LibFormatter:  
        OutputType = 'text'  
  
        @classmethod  
        def exp10(self, exp):  
            if LibFormatter.OutputType == 'latex':  
                return f' \\cdot 10^{{{exp}}}'  
            elif LibFormatter.OutputType == 'text':  
                return f'e{exp}'  
            else:  
                raise ValueError(f"Unsupported OutputType: '{LibFormatter.  
        ↪OutputType}'")  
  
        @classmethod  
        def pm(self):  
            if LibFormatter.OutputType == 'latex':  
                return f'\\pm'  
            elif LibFormatter.OutputType == 'text':  
                return f'±'  
            else:  
                raise ValueError(f"Unsupported OutputType: '{LibFormatter.  
        ↪OutputType}'")  
  
        @classmethod  
        def sigma(self):  
            if LibFormatter.OutputType == 'latex':  
                return f'\\sigma'
```

```

        elif LibFormatter.OutputType == 'text':
            return f' '
        else:
            raise ValueError(f"Unsupported OutputType: '{LibFormatter.
↪OutputType}'")

```

```

[2]: import math
import numpy as np

class ValErr:
    val: float = 0
    err: float = 0
    err_set = False

    def __init__(self, val, err=0):
        self.val = val
        if err != 0:
            self.err_set = True
            self.err = err

    def getTuple(self):
        return (self.val, self.err)

    def setErr(self, err_value):
        self.err_set = True
        self.err = err_value

    @classmethod
    def fromMeasurements(self, measurements):
        return ValErr(np.mean(measurements), (1 / math.sqrt(len(measurements)))
↪* np.std(measurements, ddof=1))

    @classmethod
    def fromTuple(self, tup):
        return ValErr(tup[0], tup[1])

    @classmethod
    def fromFit(self, popt, pcov, i):
        return ValErr(popt[i], np.sqrt(pcov[i][i]))

    @classmethod
    def fromFitAll(self, popt, pcov):
        for i in range(0, len(popt)):
            yield ValErr(popt[i], np.sqrt(pcov[i][i]))

    @classmethod
    def fromValPerc(self, v, perc):

```

```

    return ValErr(v, v * perc/100)

def strfmt(self, prec=2):
    if self.err != 0:
        return fr"{self.val:.{prec}e} {LibFormatter.pm()} {self.err:.{prec}e}"
    else:
        return f"{self.val:.{prec}e}"

def strfmtf(self, prec, exp, name = ""):
    prefix = ""
    if name != "":
        prefix = f"{name} = "

    if self.err != 0:
        return prefix + fr"{floatfmt(self.val, prec, exp)} {LibFormatter.
pm()} {floatfmt(self.err, prec, exp)}"
    else:
        return prefix + f"{floatfmt(self.val, prec, exp)}"

def strfmtf2(self, prec, exp, name = ""):
    prefix = ""
    if name != "":
        prefix = f"{name} = "

    if self.err != 0:
        return prefix + fr"{f'(' if exp != 0 else ''}{self.val/10**(exp):
0=1.{prec}f} {LibFormatter.pm()} {self.err/10**(exp):0=1.
{prec}f}{f')' if exp != 0 else ''}"
    else:
        return prefix + f"{floatfmt(self.val, prec, exp)}"

def strltx(self, prec=2):
    if self.err != 0:
        return fr"{self.val:.{prec}e} \pm {self.err:.{prec}e}"
    else:
        return f"{self.val}"

def relerr(self):
    return self.err / self.val

def sigmadiff(self, other):
    return np.abs(self.val - other.val) / np.sqrt(self.err**2 + other.
err**2)

def sigmadiff_fmt(self, other, prec=2):
    return f"{prec_ceil(self.sigmadiff(other), prec)}{LibFormatter.sigma()}"

```

```

def pow(self, p):
    return ValErr(self.val**2, 2 * self.val * self.err)

def __repr__(self):
    return f"ValErr({self.val}, {self.err})"

def __radd__(self, other):
    return self.__add__(other)

def __add__(self, other):
    if isinstance(other, self.__class__):
        return ValErr(self.val + other.val, math.sqrt(self.err**2 + other.
↪err**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(self.val + other, self.err)
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
↪__class__}' and '{type(other)}'")

def __rsub__(self, other):
    if isinstance(other, self.__class__):
        return ValErr(other.val - self.val, math.sqrt(other.err**2 + self.
↪err**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(other - self.val, self.err)
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
↪__class__}' and '{type(other)}'")

def __sub__(self, other):
    if isinstance(other, self.__class__):
        return ValErr(self.val - other.val, math.sqrt(self.err**2 + other.
↪err**2))
    elif isinstance(other, float) or isinstance(other, int):
        return ValErr(self.val - other, self.err)
    else:
        raise TypeError(f"unsupported operand type(s) for +: '{self.
↪__class__}' and '{type(other)}'")

def __rmul__(self, other):
    return self.__mul__(other)

def __mul__(self, other):
    if isinstance(other, self.__class__):

```

```

        return ValErr(self.val * other.val, math.sqrt((other.val * self.
↪err)**2 + (self.val * other.err)**2))
        elif isinstance(other, float) or isinstance(other, int):
            return ValErr(self.val * other, self.err * np.abs(other))
        else:
            raise TypeError(f"unsupported operand type(s) for +: '{self.
↪__class__}' and '{type(other)}'")

    def __rtruediv__(self, other):
        if isinstance(other, self.__class__):
            return ValErr(other.val / self.val, math.sqrt((other.err / self.
↪val)**2 + (other.val * self.err / self.val**2)**2))
        elif isinstance(other, float) or isinstance(other, int):
            return ValErr(other / self.val, np.abs(other / self.val**2) * self.
↪err)
        else:
            raise TypeError(f"unsupported operand type(s) for +: '{self.
↪__class__}' and '{type(other)}'")

    def __truediv__(self, other):
        if isinstance(other, self.__class__):
            return ValErr(self.val / other.val, math.sqrt((self.err / other.
↪val)**2 + (self.val * other.err / other.val**2)**2))
        elif isinstance(other, float) or isinstance(other, int):
            return ValErr(self.val / other, self.err / other)
        else:
            raise TypeError(f"unsupported operand type(s) for +: '{self.
↪__class__}' and '{type(other)}'")

```

```

[54]: def spacearound(dat, add):
        return np.linspace(dat[0] - add, dat[len(dat)-1] + add)

```

```

[55]: def div_with_err(a, a_err, b, b_err):
        err = (1 / b) * np.sqrt(a_err**2 + (a * b_err / b)**2)
        return (a / b, err)

```

```

[56]: def print_all(*args):
        for e in args:
            print(e)

```

```

[ ]:

```