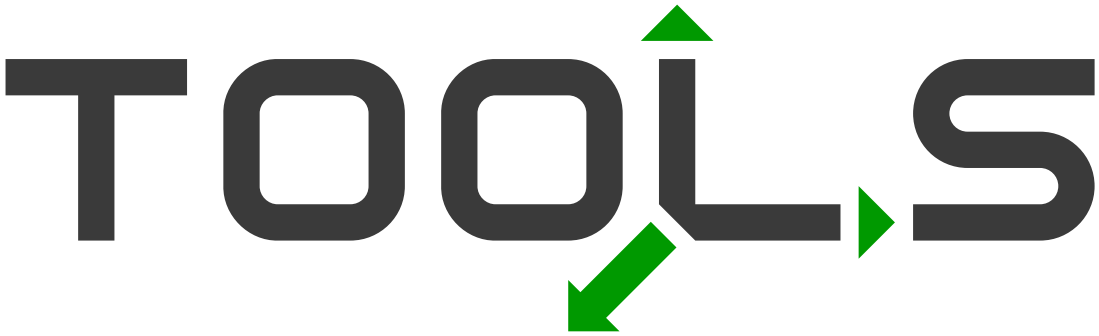
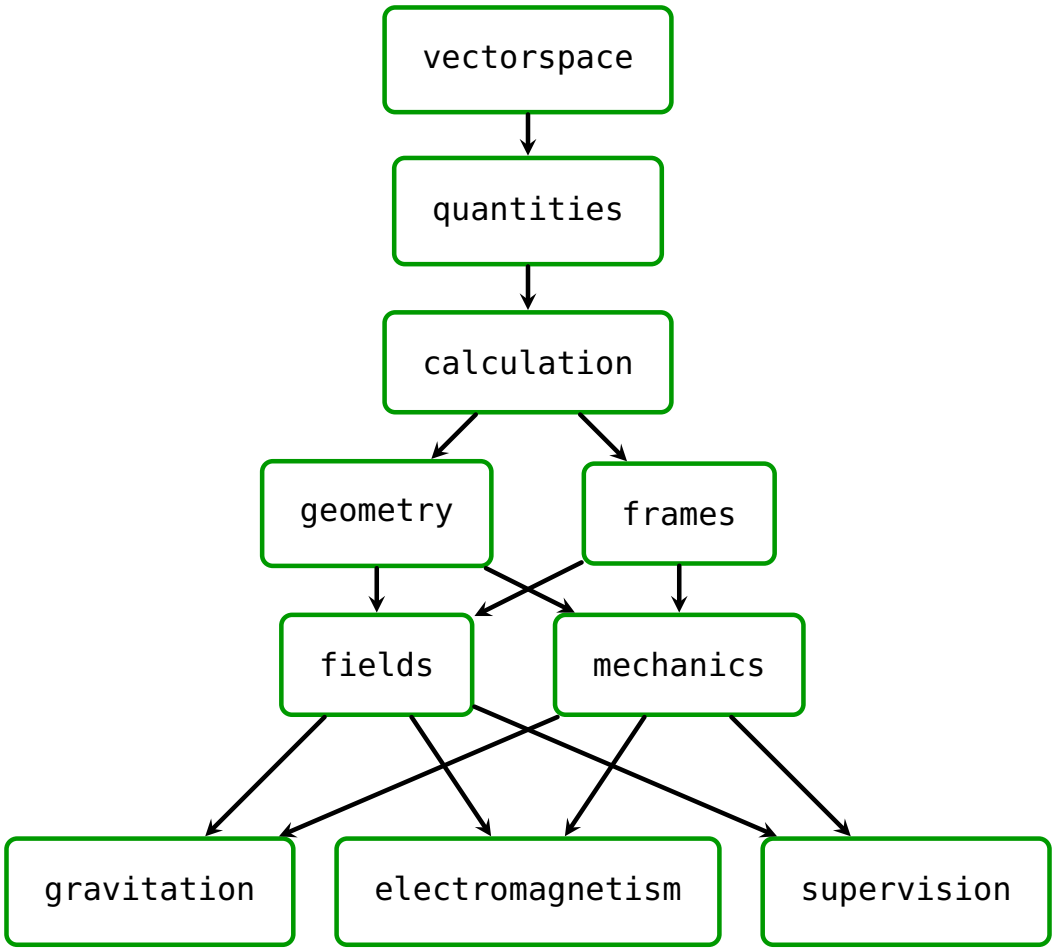

TOOLS

Python package for physics modeling.

Dunstan Becht



Contents

0 __init__

0.1 Storage

0.1.1 Save

__init__.py

```
def save(path, data):
    """Save the data to a file."""
    file = open(path, "w")
    file.write(__version__+"\n"+repr(data))
    file.close()
```

0.1.2 Load

__init__.py

```
def load(path):
    """Load the data from a file and return a string."""
    file = open(path, "r")
    version, content = file.read().splitlines()
    file.close()
    if version == __version__:
        return content
    raise ImportError("version conflict: requires version "+version)
```

0.2 Unit errors

__init__.py

```
class UnitError(ValueError):
    """Non homogenous operation or incorrect unit."""
```

0.3 Arguments

0.3.1 Management

__init__.py

```
def arguments(args):
    """Simplify the recognition of arguments for functions using *args."""
    if len(args) is 0:
        raise ValueError("no argument is given")
    if len(args) is 1 and isinstance(args[0], (list, tuple)):
        return args[0]
    return args
```

0.3.2 Types

__init__.py

```
def checkType(args, t=None):
    """Check the type of the items in 'args'."""
    if t is None: # the arguments must have the same type
        for i in range(1, len(args)):
            if type(args[0]) is not type(args[i]):
                raise TypeError("the arguments have different types")
        return type(args[0])
    else: # the arguments must have the type 't'
        for a in args:
            if not isinstance(a, t):
                message = "type "+str(type(a))+" invalid, "+str(t)+" requested"
                raise TypeError(message)
```

0.3.3 Sizes

__init__.py

```
def checkSize(args, s=None):
    """Check the size of the items in 'args'."""
    if s is None: # the arguments must be the same size
        for i in range(1, len(args)):
            if len(args[0]) is not len(args[i]):
                raise TypeError("the arguments have different sizes")
        return len(args[0])
    else: # the arguments must be of size 's'
        for a in args:
            if len(a) is not s:
                message = "size "+str(len(a))+" invalid, "+str(s)+" requested"
                raise ValueError(message)
```

0.3.4 Units

__init__.py

```
def checkUnit(args, u=None):
    """Check the unit of the items in 'args'."""
    if u is None: # the arguments must have the same unit
        for i in range(1, len(args)):
            if args[0].unit != args[i].unit:
                raise UnitError("the arguments have different units")
        return args[0].unit
    else: # the arguments must have the unit 'u'
        for a in args:
            if a.unit != u:
                message = "unit "+str(a.unit)+" invalid, "+str(u)+" requested"
                raise UnitError(message)
```

1 vectorspace

1.1 Vectors

$$\begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \longleftrightarrow \texttt{Vector(v\_1, ..., v\_n)}$$

$$n \longleftrightarrow \texttt{len(v)}$$

$$v_{i+1} \longleftrightarrow \texttt{v[i]}$$

vectorspace.py

```
class Vector:
    """A vector is defined by a sequence of scalars."""

    scalars = (int, float, complex)

    def __init__(self, *args):
        args = arguments(args)
        checkType(args, Vector.scalars)
        self.coordinates = list(args)

    def __len__(self):
        return len(self.coordinates)

    def __getitem__(self, index):
        return self.coordinates[index]

    def __setitem__(self, index, value):
        checkType([value], Vector.scalars)
        self.coordinates[index] = value

    def __repr__(self):
        return "Vector("+", ".join([repr(c) for c in self])+")"

    def __str__(self):
        n = max([len(str(c)) for c in self])
        return "\n".join(["|"+" "*(n-len(str(c)))+str(c)+"|" for c in self])

    def __eq__(self, vector):
        checkType([vector], Vector)
        return self.coordinates == vector.coordinates

    def copy(self):
        return eval(repr(self))
```

1.2 Matrices

$$\begin{pmatrix} m_{1,1} & \cdots & m_{1,n} \\ \vdots & & \vdots \\ m_{n,1} & \cdots & m_{n,n} \end{pmatrix} \longleftrightarrow \begin{matrix} \texttt{Matrix(} & \texttt{Vector(m\_1\_1, ..., m\_n\_1),} \\ & \vdots \\ & \texttt{Vector(m\_1\_n, ..., m\_n\_n)} \quad \texttt{)} \end{matrix}$$

$$n \longleftrightarrow \texttt{len(m)}$$

$$m_{i+1,j+1} \longleftrightarrow \texttt{m[j][i]}$$

vectorspace.py

```
class Matrix:
    """A matrix is defined by a sequence of column vectors."""

    def __init__(self, *args):
        args = arguments(args)
        checkType(args, Vector)
        checkSize(args, len(args))
        self.vectors = [v for v in args]

    def __len__(self):
        return len(self.vectors)

    def __getitem__(self, index):
        return self.vectors[index]

    def __setitem__(self, index, value):
        checkType(value, Vector)
        checkSize(value, len(self))
        self.vectors[index] = value

    def __repr__(self):
        return "Matrix("+", ".join([repr(v) for v in self])+")"

    def __str__(self):
        n = [max([len(str(c)) for c in v]) for v in self]
        p = [[" "*(n[j]-len(str(self[j][i])))+str(self[j][i])
              for j in range(len(self))]
              for i in range(len(self))]
        return "\n".join(["|"+" ".join(line)+"|" for line in p])

    def __eq__(self, matrix):
        checkType(matrix, Matrix)
        return self.vectors == matrix.vectors

    def copy(self):
        return eval(repr(self))
```

1.3 Magnitude functions

1.3.1 Addition

$$\sum_{i=1}^{n} a_i \quad \longleftrightarrow \quad \texttt{magAdd(a\_1, ..., a\_n)}$$

vectorspace.py

```
def magAdd(*args):
    """Return the sum of the elements in 'args'."""
    args = arguments(args)
    if isinstance(args[0], Vector.scalars):
        checkType(args, Vector.scalars)
        return sum(args)
    argType = checkType(args)
    if argType is Vector:
        argSize = checkSize(args)
        return Vector([sum([c[i] for c in args]) for i in range(argSize)])
    if argType is Matrix:
        argSize = checkSize(args)
        return Matrix([magAdd([c[i] for c in args]) for i in range(argSize)])
    raise TypeError("addition not defined for "+str(argType))
```

1.3.2 Substraction

$$a - b \quad \longleftrightarrow \quad \texttt{magSub(a, b)}$$

vectorspace.py

```
def magSub(a, b):
    """Return the difference of 'a' and 'b'."""
    if isinstance(a, Vector.scalars):
        checkType([b], Vector.scalars)
        return a-b
    argType = checkType([a, b])
    if argType is Vector:
        argSize = checkSize([a, b])
        return Vector([a[j]-b[j] for j in range(argSize)])
    if argType is Matrix:
        argSize = checkSize([a, b])
        return Matrix([magSub(a[j], b[j]) for j in range(argSize)])
    raise TypeError("substraction not defined for "+str(argType))
```

1.3.3 Multiplication

$$ab \quad \longleftrightarrow \quad \texttt{auxMul(a, b)}$$

vectorspace.py

```
def auxMul(a, b):
    """Return the product of 'a' and 'b'."""
    if isinstance(a, Vector.scalars):
        if isinstance(b, Vector.scalars):
            return a*b
        if isinstance(b, Vector):
            return Vector([a*c for c in b])
        if isinstance(b, Matrix):
            return Matrix([auxMul(a, v) for v in b])
    if isinstance(a, Vector) and isinstance(b, Vector.scalars):
        return Vector([b*c for c in a])
    if isinstance(a, Matrix):
        if isinstance(b, Vector.scalars):
            return Matrix([auxMul(b, v) for v in a])
        if isinstance(b, Vector):
            if len(a) is not len(b):
                raise ValueError(str(len(a))+"-dimensional matrix * "+
                                 str(len(b))+"-dimensional vector invalid")
            coordinates = [sum([a[j][i]*b[j] for j in range(len(a))])
                                             for i in range(len(a))]
            return Vector(coordinates)
        if isinstance(b, Matrix):
            if len(a) is not len(b):
                raise ValueError(str(len(a))+"-dimensional matrix * "+
                                 str(len(b))+"-dimensional matrix invalid")
            vectors = [auxMul(a, v) for v in b]
            return Matrix(vectors)
    raise TypeError(str(type(a))+" * "+str(type(b))+" multiplication invalid")
```

$$\prod_{i=1}^{n} a_i \quad \longleftrightarrow \quad \texttt{magMul(a\_1, ..., a\_n)}$$

vectorspace.py

```
def magMul(*args):
    """Return the product of the elements in 'args'."""
    i = len(args)-1
    a = args[i]
    while i is not 0:
        i -= 1
        a = auxMul(args[i], a)
    return a
```

1.3.4 Division

$$\frac{a}{b} \longleftrightarrow \texttt{magDiv(a, b)}$$

vectorspace.py

```
def magDiv(a, b):
    """Return the division of 'a' by 'b'."""
    checkType([b], Vector.scalars)
    return auxMul(a, 1/b)
```

1.3.5 Power

$$a^p \longleftrightarrow \texttt{magPwr(p)(a)}$$

vectorspace.py

```
def magPwr(p):
    """Return the 'p'-th power function."""
    def aux(a):
        if isinstance(a, Vector.scalars):
            return a**p
        if isinstance(a, Matrix) and isinstance(p, int) and p >= 0:
            if p is 0:
                return identity(len(a))
            return magMul([a for i in range(p)])
        raise TypeError("power "+str(p)+" not defined for "+str(type(a)))
    return aux
```

1.3.6 Logarithm

$$\log_b(a) \longleftrightarrow \texttt{magLog(b)(a)}$$

vectorspace.py

```
def magLog(b=math.e):
    """Return the logarithm to base 'b' function."""
    def aux(a):
        checkType([a], Vector.scalars)
        return math.log(a, b)
    return aux
```

1.3.7 Exponential

$$\exp(a) \quad \longleftrightarrow \quad \texttt{magExp(a)}$$

vectorspace.py

```
def magExp(a, n=1000):
    """Return the exponential of 'a'."""
    if isinstance(a, Vector.scalars):
        return math.exp(a)
    if isinstance(a, Matrix):
        p = [magDiv(magPwr(k)(a), math.factorial(k)) for k in range(n)]
        return magAdd(p)
    raise TypeError("exponential not defined for "+str(type(a)))
```

1.3.8 Scalar product

$$a \cdot b \quad \longleftrightarrow \quad \texttt{magScaPro(a, b)}$$

vectorspace.py

```
def magScaPro(a, b):
    """Return the scalar product of 'a' and 'b'."""
    checkType([a, b], Vector)
    argSize = checkSize([a, b])
    return sum([a[i]*b[i] for i in range(argSize)])
```

1.3.9 Vector product

$$a \times b \quad \longleftrightarrow \quad \texttt{magVecPro(a, b)}$$

vectorspace.py

```
def magVecPro(a, b):
    """Return the vector product of 'a' and 'b' in dimension 3."""
    checkType([a, b], Vector)
    checkSize([a, b], 3)
    x = a[1]*b[2] - a[2]*b[1]
    y = a[2]*b[0] - a[0]*b[2]
    z = a[0]*b[1] - a[1]*b[0]
    return Vector(x, y, z)
```

1.3.10 Norm

$$\|a\|_p \longleftrightarrow \left(\sum_{i=1}^{n} |a_i|^p\right)^{\frac{1}{p}} \longleftrightarrow \texttt{magNorm(p)(a)}$$

vectorspace.py

```
def magNorm(p=2):
    """Return the 'p'-norm function."""
    def aux(a):
        if isinstance(a, Vector.scalars):
            return abs(a)
        if isinstance(a, Vector):
            return sum([abs(c)**p for c in a])**(1/p)
        raise TypeError("norm "+str(p)+" not defined for "+str(type(a)))
    return aux
```

1.3.11 Determinant

$$\begin{array}{c} \\ i \end{array} \begin{array}{c} \begin{array}{ccc} & j & \end{array} \\ \begin{pmatrix} A & * & B \\ * & * & * \\ C & * & D \end{pmatrix} \end{array} \longleftrightarrow \texttt{submatrix(i, j, m)}$$

vectorspace.py

```
def submatrix(iCut, jCut, m):
    """Return 'm' without the line 'iCut' and the column 'jCut'."""
    p = [[m[j][i] for i in range(len(m)) if i is not iCut]
                  for j in range(len(m)) if j is not jCut]
    return Matrix([Vector(coordinates) for coordinates in p])
```

$$\det(m) \longleftrightarrow \sum_{j=1}^{n} (-1)^{1+j} m_{1,j} M_{1,j} \longleftrightarrow \texttt{magDet(m)}$$

vectorspace.py

```
def magDet(m):
    """Return the determinant of 'm' using recursion."""
    checkType([m], Matrix)
    if len(m) is 1:
        return m[0][0]
    p = [((-1)**j)*m[j][0]*magDet(submatrix(0, j, m)) for j in range(len(m))]
    return sum(p)
```

1.3.12 Identity

$$I_n \longleftrightarrow \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}_n \longleftrightarrow \texttt{identity(n)}$$

vectorspace.py

```
def identity(n):
    """Return the identity matrix of size 'n'."""
    p = [[1 if i is j else 0 for i in range(n)] for j in range(n)]
    return Matrix([Vector(coordinates) for coordinates in p])
```

1.3.13 Inverse

$$R_i \leftrightarrow R_j \longleftrightarrow \texttt{rowSwi(i, j, m)}$$

$$kR_i \to R_i \longleftrightarrow \texttt{rowMul(i, k, m)}$$

$$R_i + kR_j \to R_i \longleftrightarrow \texttt{rowAdd(i, k, j, m)}$$

vectorspace.py

```
def rowSwi(i, j, m):
    """Apply the elementary operation R'i' <-> R'j' to 'm'."""
    for k in range(len(m)):
        m[k][i], m[k][j] = m[k][j], m[k][i]

def rowMul(i, k, m):
    """Apply the elementary operation 'k' * R'i' -> R'i' to 'm'."""
    if k == 0:
        raise ValueError("'k' is zero")
    for j in range(len(m)):
        m[j][i] = k*m[j][i]

def rowAdd(i, k, j, m):
    """Apply the elementary operation R'i' + 'k' * R'j' -> R'i' to 'm'."""
    if i is j:
        raise ValueError("'i' = 'j'")
    for p in range(len(m)):
        m[p][i] += k*m[p][j]
```

i_{pivot} ⟷ `pivot(j, m)`

vectorspace.py

```
def pivot(j, m):
    """Return a pivot for the 'j'th column of 'm'."""
    row, absMax = j, abs(m[j][j])
    for i in range(j+1, len(m)):
        if abs(m[j][i]) > absMax:
            row, absMax = i, abs(m[j][i])
    return row
```

m^{-1} ⟷ `magInv(m)`

vectorspace.py

```
def magInv(m, checkDet=True):
    """Return the inverse of 'm' using Gaussian elimination."""
    if isinstance(m, Vector.scalars):
        return 1/m
    if not isinstance(m, Matrix):
        raise TypeError("inverse not defined for "+str(type(m)))
    if checkDet and magDet(m) == 0:
        raise ValueError("the determinant of the matrix is zero")

    auxMatrix = m.copy()
    invMatrix = identity(len(m))
    for j in range(len(m)):

        # Switching
        i = pivot(j, auxMatrix)
        rowSwi(i, j, auxMatrix)
        rowSwi(i, j, invMatrix)

        # Multiplication
        k = 1/auxMatrix[j][j]
        rowMul(j, k, auxMatrix)
        rowMul(j, k, invMatrix)

        # Addition
        for i in range(len(m)):
            if i is not j:
                k = -auxMatrix[j][i]
                rowAdd(i, k, j, auxMatrix)
                rowAdd(i, k, j, invMatrix)

    return invMatrix
```

1.3.14 Rotation matrix

$$u = \frac{v}{\|v\|} \quad \theta = \|v\|$$

$$\begin{pmatrix} u_x^2(1-\cos\theta)+\cos\theta & u_x u_y(1-\cos\theta)-u_z\sin\theta & u_x u_z(1-\cos\theta)+u_y\sin\theta \\ u_x u_y(1-\cos\theta)+u_z\sin\theta & u_y^2(1-\cos\theta)+\cos\theta & u_y u_z(1-\cos\theta)-u_x\sin\theta \\ u_x u_z(1-\cos\theta)-u_y\sin\theta & u_y u_z(1-\cos\theta)+u_x\sin\theta & u_z^2(1-\cos\theta)+\cos\theta \end{pmatrix}$$

$$\updownarrow$$

`magRotMat(v)`

vectorspace.py

```
def magRotMat(v):
    """Return the rotation matrix associated with the rotation vector 'v'."""
    checkType([v], Vector)
    checkSize([v], 3)

    theta = magNorm(2)(v)
    if theta == 0:
        return identity(3)

    u = auxMul(v, 1/theta)
    x, y, z = u[0], u[1], u[2]
    c, s = math.cos(theta), math.sin(theta)

    r1 = Vector(x*x*(1-c)+c, x*y*(1-c)+z*s, x*z*(1-c)-y*s)
    r2 = Vector(x*y*(1-c)-z*s, y*y*(1-c)+c, y*z*(1-c)+x*s)
    r3 = Vector(x*z*(1-c)+y*s, y*z*(1-c)-x*s, z*z*(1-c)+c)

    return Matrix(r1, r2, r3)
```

1.3.15 Component

$$v_i \quad \longleftrightarrow \quad \texttt{magCom(i)(v)}$$

vectorspace.py

```
def magCom(i):
    """Return the 'i'th component of 'v'."""
    def aux(v):
        checkType([v], Vector)
        return v[i]
    return aux
```

2 quantities

2.1 Units

quantities.py

```
class Unit:
    """Represents the unit of a physical quantity."""

    base = [("kg",  [1, 0, 0, 0, 0, 0, 0], "mass"),
            ("m",   [0, 1, 0, 0, 0, 0, 0], "length"),
            ("s",   [0, 0, 1, 0, 0, 0, 0], "time"),
            ("A",   [0, 0, 0, 1, 0, 0, 0], "electric current"),
            ("K",   [0, 0, 0, 0, 1, 0, 0], "temperature"),
            ("mol", [0, 0, 0, 0, 0, 1, 0], "amount of substance"),
            ("cd",  [0, 0, 0, 0, 0, 0, 1], "luminous intensity")]

    derived = [("C",  [ 0,  0,  1,  1,  0,  0,  0], "electric charge"),
               ("F",  [-1, -2,  4,  2,  0,  0,  0], "electrical capacitance"),
               ("H",  [ 1,  2, -2, -2,  0,  0,  0], "electrical inductance"),
               ("J",  [ 1,  2, -2,  0,  0,  0,  0], "energy"),
               ("lx", [ 0, -2,  0,  0,  0,  0,  1], "illuminance"),
               ("N",  [ 1,  1, -2,  0,  0,  0,  0], "force"),
               ("Pa", [ 1, -1, -2,  0,  0,  0,  0], "pressure"),
               ("S",  [-1, -2,  3,  2,  0,  0,  0], "electrical conductance"),
               ("T",  [ 1,  0, -2, -1,  0,  0,  0], "magnetic field"),
               ("V",  [ 1,  2, -3, -1,  0,  0,  0], "voltage"),
               ("W",  [ 1,  2, -3,  0,  0,  0,  0], "power"),
               ("Wb", [ 1,  2, -2, -1,  0,  0,  0], "magnetic flux")]

    usual = [("",        [ 0,  0,  0,  0,  0,  0,  0], "dimensionless"),
             ("V.m-1",   [ 1,  1, -3, -1,  0,  0,  0], "electric field"),
             ("C.m-1",   [ 0, -1,  1,  1,  0,  0,  0], "linear charge"),
             ("C.m-2",   [ 0, -2,  1,  1,  0,  0,  0], "surface charge"),
             ("C.m-3",   [ 0, -3,  1,  1,  0,  0,  0], "volume charge"),
             ("F.m-1",   [-1, -3,  4,  2,  0,  0,  0], "permittivity"),
             ("H.m-1",   [ 1,  1, -2, -2,  0,  0,  0], "permeability"),
             ("A.m-1",   [ 0, -1,  0,  1,  0,  0,  0], "magnetization"),
             ("A.m-2",   [ 0, -2,  0,  1,  0,  0,  0], "current density"),
             ("m-1",     [ 0, -1,  0,  0,  0,  0,  0], "wavenumber"),
             ("m2",      [ 0,  2,  0,  0,  0,  0,  0], "area"),
             ("m3",      [ 0,  3,  0,  0,  0,  0,  0], "volume"),
             ("kg.m-1",  [ 1, -1,  0,  0,  0,  0,  0], "linear mass"),
             ("kg.m-2",  [ 1, -2,  0,  0,  0,  0,  0], "surface mass"),
             ("kg.m-3",  [ 1, -3,  0,  0,  0,  0,  0], "volume mass"),
             ("S.m-1",   [-1, -3,  3,  2,  0,  0,  0], "conductivity"),
             ("m.s-1",   [ 0,  1, -1,  0,  0,  0,  0], "speed"),
             ("m.s-2",   [ 0,  1, -2,  0,  0,  0,  0], "acceleration"),
             ("m.s-3",   [ 0,  1, -3,  0,  0,  0,  0], "jerk"),
             ("m.s-4",   [ 0,  1, -4,  0,  0,  0,  0], "jounce"),
             ("N.s",     [ 1,  1, -1,  0,  0,  0,  0], "momentum"),
             ("N.m.s",   [ 1,  2, -1,  0,  0,  0,  0], "angular momentum"),
             ("kg.m2",   [ 1,  2,  0,  0,  0,  0,  0], "moment of inertia"),
             ("J.K-1",   [ 1,  2, -2,  0, -1,  0,  0], "heat capacity"),
             ("K.W-1",   [-1, -2,  3,  0,  1,  0,  0], "thermal resistance")]
```

```
    def __init__(self, u=None):
        """Convert 'u' to its associated dimensional list."""

        if u is None:
            self.dimension = [0, 0, 0, 0, 0, 0, 0]

        elif isinstance(u, Unit):
            self.dimension = list(u.dimension)

        elif isinstance(u, list):
            checkSize([u], 7)
            checkType(u, int)
            self.dimension = list(u)

        elif isinstance(u, str):

            def convert1(s):
                for table in [Unit.base, Unit.derived, Unit.usual]:
                    for u in table:
                        if s == u[0]:
                            return u[1]
                raise ValueError(s+" is not a valid unit")

            def convert2(s):
                if s == "":
                    return 1
                try:
                    return int(s)
                except:
                    raise ValueError(s+" is not a valid power")

            dimension = [0, 0, 0, 0, 0, 0, 0]
            for g in u.split("."):
                d = convert1(g.strip("+-0123456789"))
                p = convert2(g.strip(string.ascii_letters))
                dimension = [dimension[i] + d[i]*p for i in range(7)]

            self.dimension =  dimension

        else:
            raise TypeError(str(u)+" is not a valid unit")

    def __repr__(self):
        return "Unit("+str(self.dimension)+")"

    def __floordiv__(self, d):
        checkType([d], (Unit, list))
        if isinstance(d, Unit):
            d = d.dimension
        q1, q2 = float('inf'), float('inf')
        for i in range(7):
            if d[i] is not 0:
                q1 = min(max(0, self.dimension[i]//d[i]), q1)
                q2 = min(max(0, -self.dimension[i]//d[i]), q2)
        return q1-q2
```

```
    def __str__(self):
        """Convert the dimensional list into a string."""

        # search in usual units
        for u in self.usual:
            if self.dimension == u[1]:
                return u[0]

        # decomposition into derived units
        q, w = 0, 0
        for i in range(len(Unit.derived)):
            s_i = Unit.derived[i][0]         # symbol
            d_i = Unit.derived[i][1]         # dimension
            q_i = self//d_i                  # quotient
            w_i = sum([abs(c) for c in d_i]) # weight
            if q_i is not 0 and w_i > w:
                s, d, q, w = s_i, d_i, q_i, w_i
        if q is not 0:
            if q is 1:
                p = ""
            else:
                p = str(q)
            r = str(Unit([self.dimension[i] - q*d[i] for i in range(7)]))
            if r is "":
                 return s+p
            return s+p+"."+r

        # decomposition into base units
        s = []
        for i in range(7):
            if self.dimension[i] is not 0:
                if self.dimension[i] is 1:
                    s.append(Unit.base[i][0])
                else:
                    s.append(Unit.base[i][0]+str(self.dimension[i]))
        return ".".join(s)

    def __eq__(self, u):
        checkType([u], (Unit, list))
        if isinstance(u, list):
            return self.dimension == u
        return self.dimension == u.dimension

    def name(self):
        for table in [Unit.base, Unit.derived, Unit.usual]:
            for u in table:
                if self.dimension == u[1]:
                    return u[2]
        return "unknown"

    def dimensionString(self):
        p, d = ["M", "L", "T", "I", "\u03F4", "N", "J"], self.dimension
        return ".".join([p[i]+str(d[i]) for i in range(7) if d[i] is not 0])

    def copy(self):
        return eval(repr(self))
```

$kg + kg \longleftrightarrow kg \longleftrightarrow$ `uniAdd(Unit("kg"), Unit("kg"))` $\longleftrightarrow$ `Unit("kg")`

quantities.py

```
def uniAdd(*args):
    """Verify that the units are the same and return the unit."""
    if not all([args[0] == args[i] for i in range(1, len(args))]):
        raise UnitError("the arguments have different units")
    return args[0]
```

$N.m \longleftrightarrow J \longleftrightarrow$ `uniMul(Unit("N"), Unit("m"))` $\longleftrightarrow$ `Unit("J")`

quantities.py

```
def uniMul(*args):
    """Return the product unit."""
    return Unit([sum([u.dimension[i] for u in args]) for i in range(7)])
```

$m.s^{-1} \longleftrightarrow$ `uniDiv(Unit("m"), Unit("s"))` $\longleftrightarrow$ `Unit("m.s-1")`

quantities.py

```
def uniDiv(a, b):
    """Return the divided unit."""
    return Unit([a.dimension[i] - b.dimension[i] for i in range(7)])
```

$m^2 \longleftrightarrow$ `uniPow(2)(Unit("m"))` $\longleftrightarrow$ `Unit("m2")`

quantities.py

```
def uniPwr(p):
    """Return the 'p'-th power function."""
    def aux(u):
        d = [u.dimension[i]*p for i in range(7)]
        if isinstance(p, int):
            return Unit(d)
        if isinstance(p, float):
            q = [int(c) for c in d]
            if d == q:
                return Unit(q)
        raise ValueError("invalid power "+str(p)+" for unit "+str(u))
    return aux
```

2.2 Physical quantities

quantities.py

```
class Quantity:
    """A physical quantity is defined by a magnitude and a unit."""

    magnitudes = tuple(list(Vector.scalars)+[Vector, Matrix])

    def __init__(self, magnitude, unit=None, rewriteUnit=False):
        if isinstance(magnitude, Quantity):
            if not rewriteUnit:
                unit = magnitude.unit
            magnitude = magnitude.magnitude
        checkType([magnitude], Quantity.magnitudes)
        self.magnitude, self.unit = magnitude, Unit(unit)

    def __repr__(self):
        return "Quantity("+repr(self.magnitude)+", "+repr(self.unit)+")"

    def __str__(self):
        return str(self.magnitude)+" "+str(self.unit)

    def __eq__(self, q):
        return self.magnitude == q.magnitude and self.unit == q.unit

    def __lt__(self, q):
        checkUnit([self, q])
        return self.magnitude < q.magnitude

    def __gt__(self, q):
        checkUnit([self, q])
        return self.magnitude > q.magnitude

    def __le__(self, q):
        checkUnit([self, q])
        return self.magnitude <= q.magnitude

    def __ge__(self, q):
        checkUnit([self, q])
        return self.magnitude >= q.magnitude

    def __neg__(self):
        return Quantity(magMul(-1, self.magnitude), self.unit)

    def __getitem__(self, index):
        return Quantity(self.magnitude[index], self.unit)

    def __len__(self):
        return len(self.magnitude)

    def copy(self):
        return eval(repr(self))
```

2.3 Storage

quantities.py

```
class Storage:
    """Stores a sequence of comparable physical quantities."""

    def __init__(self, unit=None, magType=None, data=None):
        if unit is not None:
            unit = Unit(unit)
        if data is None:
            data = []
        self.unit, self.magType, self.data = unit, magType, data

    def __repr__(self):
        if self.magType is None:
            magType = "None"
        elif isinstance(self.magType, type):
            magType = self.magType.__name__
        else:
            magType = "("+", ".join([t.__name__ for t in self.magType])+")"
        return "Storage("+repr(self.unit)+", "+magType+", "+repr(self.data)+")"

    def serialize(self, value):
        if self.magType is Vector:
            return value.coordinates
        if self.magType is Matrix:
            return [v.coordinates for v in value.vectors]
        return value

    def deserialize(self, value):
        if self.magType is Vector:
            return Vector(value)
        if self.magType is Matrix:
            return Matrix([Vector(v) for v in value])
        return value

    def add(self, value):
        value = Quantity(value)
        if self.unit is None:
            self.unit = Unit(value.unit)
        if self.magType is None:
            self.magType = type(value.magnitude)
        checkUnit([value], self.unit)
        checkType([value.magnitude], self.magType)
        self.data.append(self.serialize(value.magnitude))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        return Quantity(self.deserialize(self.data[index]), self.unit)

    def copy(self):
        return eval(repr(self))
```

3 calculation

3.1 Type management

The functions of module calculation adapt to the type of their argument as follows:

f		
constant	⟼	constant
function	⟼	function

The following two functions simplify this management of types.

calculation.py

```
def makeCallable(w):
    """Return:
    - 'w' if 'w' is a function
    - the constant function of value 'w' if 'w' is a constant"""
    if callable(w):
        return w
    def aux(r):
        return w
    return aux

def appropriateType(f, p):
    """Return:
    - the function 'f' if an item from the list 'p' is a function
    - the value of the constant function 'f' else"""
    for w in p:
        if callable(w):
            return f
    return f(None)
```

3.2 Function maker

calculation.py

```
def functionMaker(magFun, uniFun=None):
    """Return the generalized version of the function."""
    if uniFun is None:
        def uniFun(u):
            if u.dimension != [0, 0, 0, 0, 0, 0, 0]:
                raise DimensionError("the quantity is not dimensionless")
            return [0, 0, 0, 0, 0, 0, 0]
    def aux1(*args1):
        args1 = arguments(args1)
        def aux2(*args2):
            q = [Quantity(makeCallable(a)(*args2)) for a in args1]
            m, u = [a.magnitude for a in q], [a.unit for a in q]
            return Quantity(magFun(*m), uniFun(*u))
        return appropriateType(aux2, args1)
    return aux1
```

3.3 Generalized functions

3.3.1 Basic functions

calculation.py

```
add = functionMaker(magAdd, uniAdd)

sub = functionMaker(magSub, uniAdd)

mul = functionMaker(magMul, uniMul)

div = functionMaker(magDiv, uniDiv)

exp = functionMaker(magExp)

scaPro = functionMaker(magScaPro, uniMul)

vecPro = functionMaker(magVecPro, uniMul)

norm = functionMaker(magNorm(2), uniAdd)

inv = functionMaker(magInv, uniPwr(-1))

rotMat = functionMaker(magRotMat)

sin = functionMaker(math.sin)

cos = functionMaker(math.cos)

tan = functionMaker(math.tan)

arcSin = functionMaker(math.asin)

arcCos = functionMaker(math.acos)

arcTan = functionMaker(math.atan)

def pwr(p):
    return functionMaker(magPwr(p), uniPwr(p))

def log(b):
    return functionMaker(magLog(b))

def det(m):
    def aux(r):
        q = Quantity(makeCallable(m)(r))
        return Quantity(magDet(q.magnitude), uniPwr(len(q.magnitude))(q.unit))
    return appropriateType(aux, [m])

def com(i):
    return functionMaker(magCom(i), uniAdd)
```

3.3.2 Derivative

$$f : \mathbb{R} \longrightarrow \mathbb{R}$$

$$\frac{d}{dx} \longleftrightarrow \texttt{der}$$

$$\frac{df}{dx} \longleftrightarrow \texttt{der(f)}$$

$$\frac{df}{dx}(x) \longleftrightarrow \texttt{der(f)(x)}$$

calculation.py

```
def der(f, d=0.001):
    """Return the derivative of the function 'f'."""
    def aux(x):
        delta = sub(f(add(x, Quantity(d, x.unit))),
                    f(sub(x, Quantity(d, x.unit))))
        if not isinstance(delta.magnitude, Vector.scalars):
            raise ValueError("'f' is not a scalar function")
        return Quantity((delta).magnitude/(2*d), uniDiv(delta.unit, x.unit))
    return aux
```

3.3.3 Partial derivative

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}$$

$$\frac{\partial}{\partial x_i} \longleftrightarrow \texttt{parDer(i)}$$

$$\frac{\partial f}{\partial x_i} \longleftrightarrow \texttt{parDer(i)(f)}$$

$$\frac{\partial f}{\partial x_i}(r) \longleftrightarrow \texttt{parDer(i)(f)(r)}$$

calculation.py

```
def parDer(i, d=0.001):
    """Return the partial derivative function with respect to axis 'i'."""
    def aux1(f):
        def aux2(r):
            def g(x):
                # g: scalar -> scalar
                v = r.copy()
                v.magnitude[i] += x.magnitude
                return f(v)
            return der(g, d)(Quantity(0, r.unit))
        return aux2
    return aux1
```

3.3.4 Simpson's rule

$$\frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right) \longleftrightarrow \texttt{simpson(a, b)(f)}$$

calculation.py

```
def simpson(a, b):
    """Return the operator integral over ['a','b'] using Simpson rule's."""
    def aux(f):
        return mul(sub(b,a), add(f(a), mul(f(div(add(a,b), 2)), 4), f(b)), 1/6)
    return aux
```

3.3.5 Definite integral

$$\int_a^b f(t)dt \longleftrightarrow \texttt{defInt(a, b)(f)}$$

calculation.py

```
def defInt(a, b, d=0.001):
    """Return the operator integral over ['a','b']."""
    def aux(f):
        if b == a:
            return Quantity(0, uniMul(f(a).unit, a.unit))
        delta = sub(b, a)
        n = int(div(delta, d).magnitude)+1
        e = div(delta, n)
        p = [simpson(add(a, mul(e, i)), add(a, mul(e, 1+i)))(f)
             for i in range(n)]
        return add(p)
    return aux
```

3.3.6 Indefinite integral

$$\int_a^x f(t)dt \longleftrightarrow \texttt{indInt(a)(f)(x)}$$

calculation.py

```
def indInt(a=0, d=0.001):
    """Return the operator integral."""
    def aux1(f):
        def aux2(x):
            return defInt(a, x, d)(f)
        return aux2
    return aux1
```

3.3.7 Vec

$$v(r) \longleftrightarrow \begin{pmatrix} c_1(r) \\ \vdots \\ c_n(r) \end{pmatrix} \longleftrightarrow \texttt{vec(c\_1, ..., c\_n)(r)}$$

calculation.py

```
def vec(*c):
    """Assemble scalar quantities into a vector."""
    def v(*r):
        c_r = [Quantity(makeCallable(a)(*r)) for a in arguments(c)]
        unit = checkUnit([a for a in c_r if a.magnitude is not 0])
        return Quantity(Vector([a.magnitude for a in c_r]), unit)
    return appropriateType(v, c)
```

3.3.8 Mat

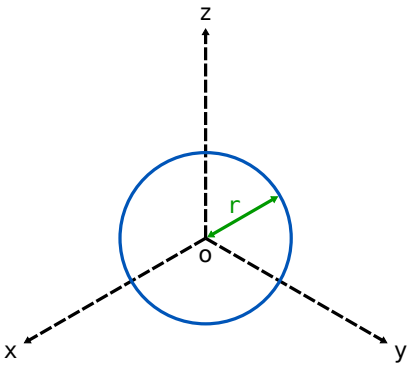
$$m(r) \longleftrightarrow \begin{pmatrix} v_1(r) & \cdots & v_n(r) \end{pmatrix} \longleftrightarrow \texttt{mat(v\_1, ..., v\_n)(r)}$$

calculation.py

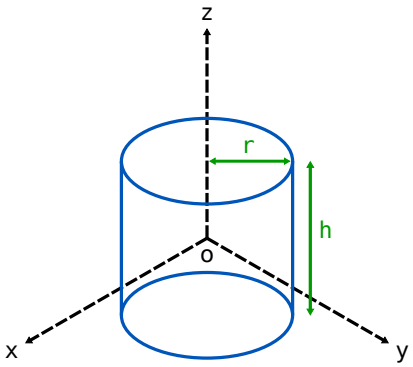
```
def mat(*v):
    """Assemble vector quantities into a matrix."""
    def m(*r):
        v_r = [Quantity(makeCallable(a)(*r)) for a in arguments(v)]
        unit = checkUnit([a for a in v_r if a.magnitude is not 0])
        for i in range(len(v_r)):
            if v_r[i].magnitude is 0:
                v_r[i].magnitude = Vector([0 for j in range(len(v_r))])
        return Quantity(Matrix([a.magnitude for a in v_r]), unit)
    return appropriateType(m, v)
```

4 geometry

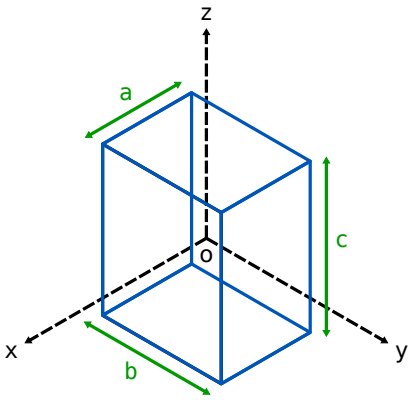
4.1 Volumes



`Volume(r)`



`Volume(r, h)`

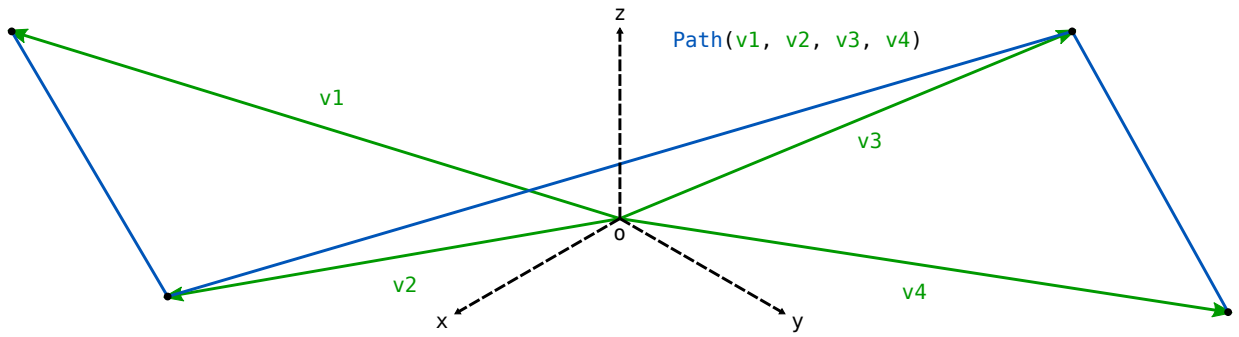


`Volume(a, b, c)`

geometry.py

```
class Volume:
    """Represents a volume."""

    def __init__(self, *args):
        args = [Quantity(a, Unit("m")) for a in arguments(args)]
        checkUnit(args, Unit("m"))
        checkType([a.magnitude for a in args], Vector.scalars)
        if len(args) is 1:
            self.geometry = "sphere"
        elif len(args) is 2:
            self.geometry = "cylinder"
        elif len(args) is 3:
            self.geometry = "cuboid"
        else:
            raise ValueError("invalid dimensions")
        self.size = list(args)

    def __repr__(self):
        return "Volume("+", ".join([str(d.magnitude) for d in self.size])+")"

    def volume(self):
        if len(self.size) is 1:
            return mul(pwr(3)(self.size[0]), (4/3)*math.pi)
        if len(self.size) is 2:
            return mul(pwr(2)(self.size[0]), self.size[1], math.pi)
        return mul(self.size[0], self.size[1], self.size[2])

    def copy(self):
        return eval(repr(self))
```

4.2 Path

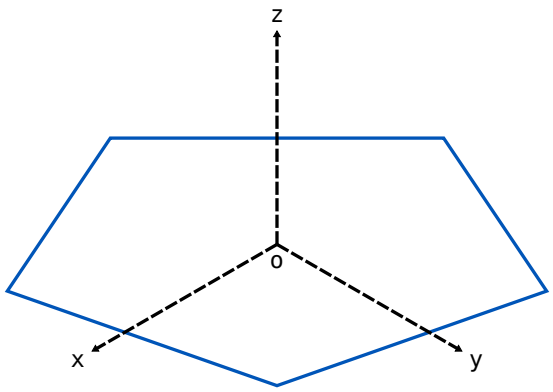


geometry.py

```
class Path:
    """Represents a path connecting the points defined by the vectors."""

    def __init__(self, *args):
        args = arguments(args)
        if isinstance(args[0], Storage):
            if args[0].magType is not Vector:
                raise ValueError("Storage unit invalid :"+str(args[0].unit))
            if args[0].unit != Unit("m"):
                raise ValueError("Storage type invalid :"+str(args[0].magType))
            self.points = args[0]
            self.dimension = checkSize(self.points.data)
        else:
            self.points = Storage("m", Vector)
            for a in [Quantity(a, Unit("m")) for a in args]:
                self.points.add(a)
            self.dimension = checkSize(self.points)

    def __repr__(self):
        return "Path("+repr(self.points)+")"

    def __getitem__(self, index):
        return self.points[index]

    def __setitem__(self, index, value):
        value = Quantity(value, "m")
        checkType([value.magnitude], Vector)
        checkSize([value], self.dimension)
        self.points[index] = value

    def __len__(self):
        return len(self.points)

    def length(self):
        if len(self) is 1:
            return Quantity(0, "m")
        return add([norm(sub(self[i+1], self[i])) for i in range(len(self)-1)])

    def copy(self):
        return eval(repr(self))
```

4.3 Functions

4.3.1 Regular polygons



regularPolygon(5)

geometry.py

```
def regularPolygon(n, r=1):
    """Return the path of a regular polygon on the plane (O, x, y)."""
    points, r = Storage("m", Vector), Quantity(r, "m")
    for i in range(n):
        a = 2*math.pi*(i/n)
        points.add(vec(mul(r, cos(a)), mul(r, sin(a)), 0))
    points.add(points[0].copy())
    return Path(points)
```

4.3.2 Subdivision

geometry.py

```
def subdivide(d, path):
    """Return a path whose distance between two points is less than 'd'."""
    checkType([path], Path)
    points, d = Storage("m", Vector), Quantity(d, "m")
    for i in range(len(path)-1):
        r = norm(sub(path[i], path[i+1]))
        if r <= d:
            points.add(path[i].copy())
        else:
            n = int(div(r, d).magnitude)
            for k in range(n):
                t = k/n
                points.add(add(mul(1-t, path[i]), mul(t, path[i+1])))
    points.add(path[-1].copy())
    return Path(points)
```

5 frames

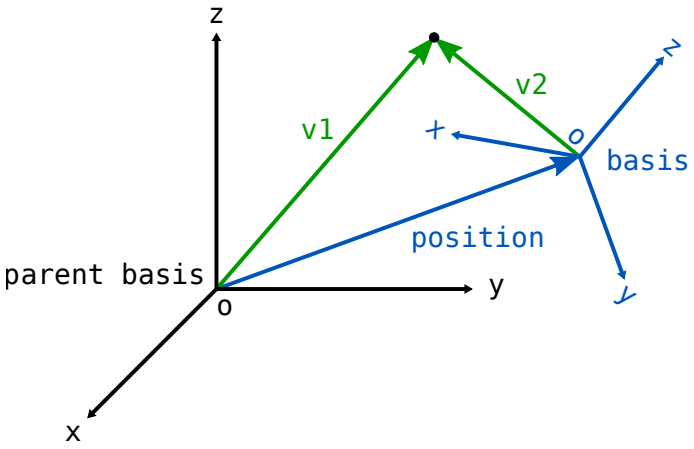
5.1 Bases

Vectors			Matrices		
$P_E^B v_E$	⟷	B.inside(v_E)	$P_E^B m_E P_B^E$	⟷	B.inside(m_E)
$P_B^E v_B$	⟷	B.outside(v_B)	$P_B^E m_B P_E^B$	⟷	B.outside(m_B)

frames.py

```
class Basis:
    """A basis is defined by a change-of-basis matrix."""

    def __init__(self, matrixBtoE, matrixEtoB=None):
        matrixBtoE = Quantity(matrixBtoE).magnitude
        checkType([matrixBtoE], Matrix)
        if matrixEtoB is not None:
            matrixEtoB = Quantity(matrixEtoB).magnitude
            checkType([matrixEtoB], Matrix)
        self.matrixBtoE = matrixBtoE
        self.matrixEtoB = matrixEtoB

    def __repr__(self):
        return "Basis("+repr(self.matrixBtoE)+", "+repr(self.matrixEtoB)+")"

    def matrixInverse(self):
        if self.matrixEtoB is None:
            self.matrixEtoB = inv(self.matrixBtoE).magnitude
        return self.matrixEtoB

    def inside(self, a):
        def aux(*args):
            q = Quantity(makeCallable(a)(*args), "m")
            if isinstance(q.magnitude, Vector):
                return mul(self.matrixInverse(), q)
            if isinstance(q.magnitude, Matrix):
                return mul(self.matrixInverse(), q, self.matrixBtoE)
            raise TypeError("no change of basis for "+str(type(q.magnitude)))
        return appropriateType(aux, [a])

    def outside(self, a):
        def aux(*args):
            q = Quantity(makeCallable(a)(*args), "m")
            if isinstance(q.magnitude, Vector):
                return mul(self.matrixBtoE, q)
            if isinstance(q.magnitude, Matrix):
                return mul(self.matrixBtoE, q, self.matrixInverse())
            raise TypeError("no change of basis for "+str(type(q.magnitude)))
        return appropriateType(aux, [a])

    def copy(self):
        return eval(repr(self))
```

5.2 Configurations



`v1` ⟷ `configuration.outside(v2)`

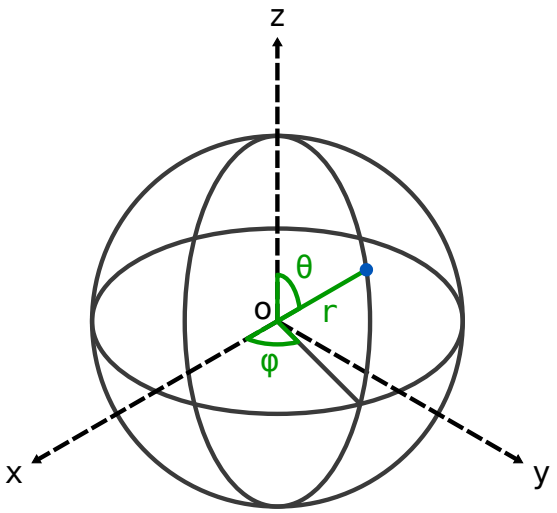
`v2` ⟷ `configuration.inside(v1)`

frames.py

```
class Configuration:
    """A configuration is defined by a position vector and a basis."""

    def __init__(self, position, basis):
        position = Quantity(position, "m")
        checkType([position.magnitude], Vector)
        checkUnit([position], Unit("m"))
        checkType([basis], Basis)
        self.position = position
        self.basis = basis

    def __repr__(self):
        position = self.position.magnitude
        return "Configuration("+repr(position)+", "+repr(self.basis)+")"

    def inside(self, v):
        """Return 'v' with respect to the configuration."""
        return self.basis.inside(sub(v, self.position))

    def outside(self, v):
        """Return 'v' with respect to the parent basis."""
        return add(self.basis.outside(v), self.position)

    def copy(self):
        return eval(repr(self))
```

5.3 Coordinate transformations

5.3.1 Spherical coordinate system

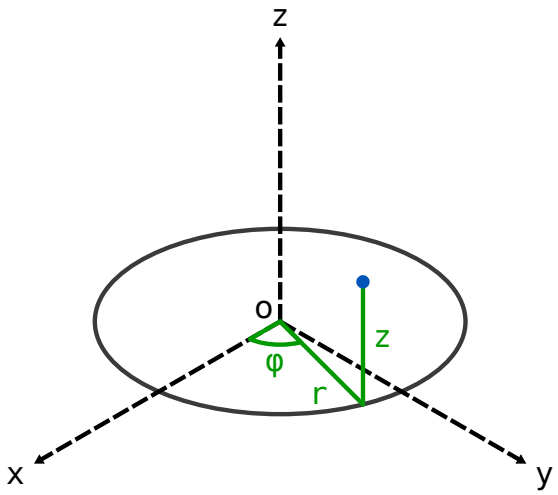


$$
\begin{aligned}
r &= \sqrt{x^2 + y^2 + z^2} \\
\theta &= \arccos\left(\frac{z}{r}\right) \\
\varphi &= \arctan\left(\frac{y}{x}\right) \\
\\
x &= r \sin\theta \cos\varphi \\
y &= r \sin\theta \sin\varphi \\
z &= r \cos\theta
\end{aligned}
$$

frames.py

```
def toSpherical(v):
    """(r: radial distance, theta: polar angle, phi: azimuthal angle)"""
    v = Quantity(v, "m")
    checkType([v.magnitude], Vector)
    r = norm(v)
    if r.magnitude == 0:
        return [r, Quantity(0), Quantity(0)]
    theta = arcCos(div(v[2], r))
    if v.magnitude[0] == 0:
        if v.magnitude[1] > 0:
            return [r, theta, Quantity(math.pi/2)]
        if v.magnitude[1] < 0:
            return [r, theta, Quantity(-math.pi/2)]
        return [r, theta, Quantity(0)]
    return [r, theta, arcTan(div(v[1], v[0]))]

def fromSpherical(r, theta, phi):
    """(r: radial distance, theta: polar angle, phi: azimuthal angle)"""
    r, theta, phi = Quantity(r, "m"), Quantity(theta), Quantity(phi)
    checkType([a.magnitude for a in [r, theta, phi]], Vector.scalars)
    if r.magnitude < 0:
        raise ValueError("'r' < 0")
    if theta.magnitude < 0 or theta.magnitude > math.pi:
        raise ValueError("'theta' < 0 or 'theta' > pi")
    if phi.magnitude < 0 or phi.magnitude > 2*math.pi:
        raise ValueError("'phi' < 0 or 'phi' > 2*pi")
    x, y = mul(r, sin(theta), cos(phi)), mul(r, sin(theta), sin(phi))
    return vec(x, y, mul(r, cos(theta)))
```

5.3.2 Cylindrical coordinate system



$$
\begin{aligned}
r &= \sqrt{x^2 + y^2} \\
\varphi &= \arctan\left(\frac{y}{x}\right) \\
z &= z
\end{aligned}
$$

$$
\begin{aligned}
x &= r\cos\varphi \\
y &= r\cos\varphi \\
z &= z
\end{aligned}
$$

frames.py

```
def toCylindrical(v):
    """(r : radial distance, phi: angular coordinate, z: height)"""
    v = Quantity(v, "m")
    checkType([v.magnitude], Vector)
    r = pwr(1/2)(add(pwr(2)(v[0]), pwr(2)(v[1])))
    z = v[2]
    if v.magnitude[0] == 0:
        if v.magnitude[1] > 0:
            return [r, Quantity(math.pi/2), z]
        if v.magnitude[1] < 0:
            return [r, Quantity(-math.pi/2), z]
        return [r, Quantity(0), z]
    return [r, arcTan(div(v[1], v[0])), z]

def fromCylindrical(r, phi, z):
    """(r : radial distance, phi: angular coordinate, z: height)"""
    r, phi, z = Quantity(r, "m"), Quantity(phi), Quantity(z, "m")
    checkType([a.magnitude for a in [r, phi, z]], Vector.scalars)
    if r.magnitude < 0:
        raise ValueError("r < 0")
    if phi.magnitude < 0 or phi.magnitude > 2*math.pi:
        raise ValueError("phi < 0 or phi > 2*pi")
    return vec(mul(r, cos(phi)), mul(r, sin(phi)), z)
```

6 fields

6.1 Operators

6.1.1 Gradient

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}$$

$$\nabla f \quad \longleftrightarrow \quad \texttt{gradient(f)}$$

fields.py

```
def gradient(f, d=0.001):
    """Return the gradient of 'f'."""
    def aux(r):
        return vec([parDer(i, d)(f)(r) for i in range(len(r))])
    return aux
```

6.1.2 Divergence

$$v : \mathbb{R}^n \longrightarrow \mathbb{R}^n$$

$$\nabla \cdot v \quad \longleftrightarrow \quad \texttt{divergence(v)}$$

fields.py

```
def divergence(v, d=0.001):
    """Return the divergence of 'v'."""
    def aux(r):
        return add([parDer(i, d)(com(i)(v))(r) for i in range(len(r))])
    return aux
```

6.1.3 Scalar Laplacian

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}$$

$$\Delta f \quad \longleftrightarrow \quad \texttt{scaLap(f)}$$

fields.py

```
def scaLap(f, d=0.001):
    """Return the scalar Laplacian of 'f'."""
    return divergence(gradient(f, d), d)
```

6.1.4 Vector Laplacian

$$v : \mathbb{R}^n \longrightarrow \mathbb{R}^n$$

$$\Delta v \quad \longleftrightarrow \quad \texttt{vecLap(v)}$$

fields.py

```
def vecLap(v, d=0.001):
    """Return the vector Laplacian of 'v'."""
    def aux(r):
        return vec([scaLap(com(i)(v), d)(r) for i in range(len(r))])
    return aux
```

6.1.5 Curl

$$v : \mathbb{R}^n \longrightarrow \mathbb{R}^n$$

$$\Delta \times v \quad \longleftrightarrow \quad \begin{pmatrix} \frac{\partial v_z}{\partial y} - \frac{\partial v_y}{\partial z} \\ \frac{\partial v_x}{\partial z} - \frac{\partial v_z}{\partial x} \\ \frac{\partial v_y}{\partial x} - \frac{\partial v_x}{\partial y} \end{pmatrix} \quad \longleftrightarrow \quad \texttt{curl(v)}$$

fields.py

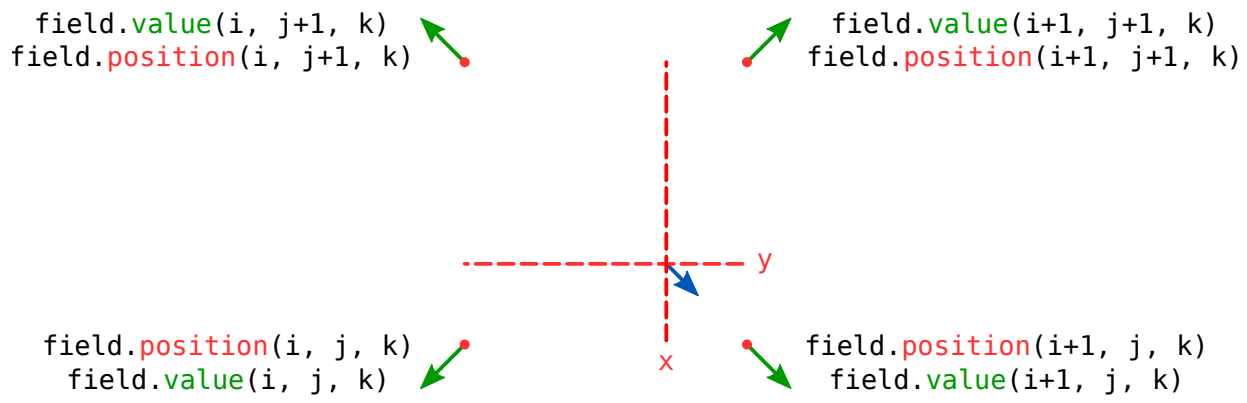
```
def curl(v, d=0.001):
    """Return the curl of 'v'."""
    v_x = com(0)(v)
    v_y = com(1)(v)
    v_z = com(2)(v)
    def aux(r):
        x = sub(parDer(1,d)(v_z)(r), parDer(2,d)(v_y)(r))
        y = sub(parDer(2,d)(v_x)(r), parDer(0,d)(v_z)(r))
        z = sub(parDer(0,d)(v_y)(r), parDer(1,d)(v_x)(r))
        return vec(x, y, z)
    return aux
```

6.2 Field samples

fields.py

```
class Field:
    """Simplifies the sampling of a field."""

    def __init__(self, vol, spg, cfg=None, stg=None, n=None):
        checkType([vol], Volume) # shape of the sampling area
        spg = Quantity(spg, "m") # spacing of measurements
        if cfg is None:
            cfg = Configuration(Vector(0, 0, 0), Basis(identity(3)))
        else:
            checkType([cfg], Configuration)
        if stg is None:
            stg = Storage()
            if vol.geometry is "cuboid":
                n = [int(div(vol.size[0], spg).magnitude)+1,
                     int(div(vol.size[1], spg).magnitude)+1,
                     int(div(vol.size[2], spg).magnitude)+1, 0]
            elif vol.geometry is "cylinder":
                n = [int(div(vol.size[0], spg).magnitude/2),
                     int(div(vol.size[0], spg).magnitude*math.pi/2),
                     int(div(vol.size[1], spg).magnitude)+1, 0]
            elif vol.geometry is "sphere":
                n = [int(div(vol.size[0], spg).magnitude/2),
                     int(div(vol.size[0], spg).magnitude*math.pi/4),
                     int(div(vol.size[0], spg).magnitude*math.pi/2), 0]
        else:
            checkType([stg], Storage), checkType([n], list), checkType(n, int)
        self.vol, self.spg, self.cfg, self.stg, self.n = vol, spg, cfg, stg, n

    def __repr__(self):
        args = [self.vol, self.spg, self.cfg, self.stg, self.n]
        return "Field("+", ".join([repr(a) for a in args])+")"

    def position(self, i, j, k):
        if self.vol.geometry is "cuboid":
            v = Vector(i-(self.n[0]-1)/2, j-(self.n[1]-1)/2, k-(self.n[2]-1)/2)
            return self.cfg.outside(mul(self.spg, v))
        if self.vol.geometry is "cylinder":
            phi, z = 2*math.pi*(j/self.n[1]), mul(self.spg, k-self.n[2]/2)
            return self.cfg.outside(fromCylindrical(mul(self.spg,i+1), phi, z))
        theta, phi = math.pi*(j/self.n[1]), 2*math.pi*(k/self.n[2])
        return self.cfg.outside(fromSpherical(mul(self.spg,i+1), theta, phi))

    def sample(self, f):
        self.n[3] +=1
        for i in range(self.n[0]):
            for j in range(self.n[1]):
                for k in range(self.n[2]):
                    self.stg.add(f(self.position(i, j, k)))

    def value(self, i, j, k, t=0):
        return self.stg[k + self.n[2]*(j + self.n[1]*(i + self.n[0]*t))]
```

6.3 Approximation




```
continuousApproximation(field)(Vector(x, y, z))
```

fields.py

```
def continuousApproximation(field, t=0):
    """Return a continuous function of the field according to the samples."""
    checkType([field], Field)
    checkType([t], int)
    if field.vol.geometry is not "cuboid":
        raise ValueError("invalid field shape: "+field.vol.geometry)

    def aux(r):
        r = field.cfg.inside(Quantity(r, "m")).magnitude
        for axe in range(3):
            if abs(r[axe]) > field.spg.magnitude*(field.n[axe]-1)/2:
                raise ValueError("'r' is outside the sampled zone")
        i = (field.n[0]-1)//2 + int(r[0]/field.spg.magnitude)
        j = (field.n[1]-1)//2 + int(r[1]/field.spg.magnitude)
        k = (field.n[2]-1)//2 + int(r[2]/field.spg.magnitude)
        if i is field.n[0]-1:
            i -=1
        if j is field.n[1]-1:
            j -=1
        if k is field.n[2]-1:
            k -=1
        d = [[r[0] - field.spg.magnitude*(i-(field.n[0]-1)/2)],
             [r[1] - field.spg.magnitude*(j-(field.n[1]-1)/2)],
             [r[2] - field.spg.magnitude*(k-(field.n[2]-1)/2)]]
        for axe in range(3):
            d[axe].append(field.spg.magnitude - d[axe][0])
        values, weight, totVol = [], [], field.spg.magnitude**3
        for c in range(2):
            for b in range(2):
                for a in range(2):
                    values.append(field.value(i+a, j+b, k+c, t))
                    weight.append(d[0][1-a]*d[1][1-b]*d[2][1-c]/totVol)
        return add([mul(weight[i], values[i]) for i in range(8)])
    return aux
```

7 mechanics

7.1 Solids

Inertia tensors

$$\begin{pmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{pmatrix}$$

Sphere

$$\begin{pmatrix} \frac{m(3r^2+4h^2)}{12} & 0 & 0 \\ 0 & \frac{m(3r^2+4h^2)}{12} & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{pmatrix}$$

Cylinder

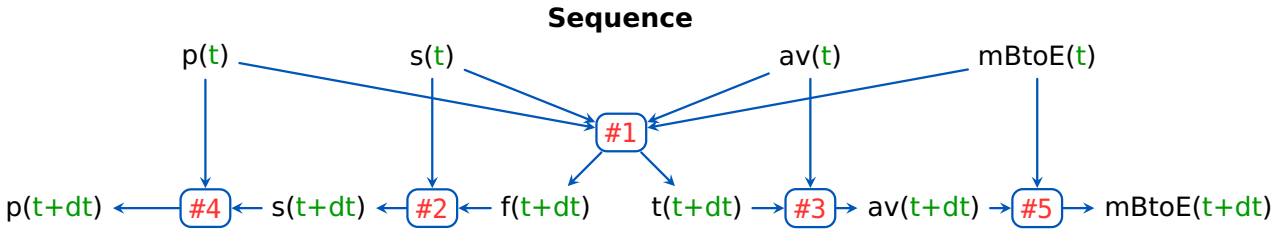
$$\begin{pmatrix} \frac{m(b^2+c^2)}{12} & 0 & 0 \\ 0 & \frac{m(a^2+c^2)}{12} & 0 \\ 0 & 0 & \frac{m(a^2+b^2)}{12} \end{pmatrix}$$

Cuboid

mechanics.py

```
class Solid:
    """Represents a solid."""

    def __init__(self, cfg, shp, m=1, q=0, i=0, M=Vector(0, 0, 0)):
        checkType([cfg], Configuration)
        checkType([shp], (Volume, Path))
        self.cfg = cfg               # configuration
        self.shp = shp               # shape
        self.m = Quantity(m, "kg")   # mass
        self.q = Quantity(q, "C")    # electric charge
        self.i = Quantity(i, "A")    # intensity
        self.M = Quantity(M, "A.m2") # magnetic moment
        self.tensor = None           # inertia tensor

    def calculateTensor(self):
        if self.shp.geometry is "sphere":
            k = mul((2/5), self.m, pwr(2)(self.shp.size[0]))
            self.tensor = mul(k, identity(3))
        if self.shp.geometry is "cylinder":
            k = div(add(mul(3, pwr(2)(self.shp.size[0])),
                        mul(4, pwr(2)(self.shp.size[1]))), 12)
            z = vec(0, 0, mul(pwr(2)(self.shp.size[0]), 1/2))
            x, y = vec(k, 0, 0), vec(0, k, 0)
            self.tensor = mul(self.m, mat(x, y, z))
        if self.shp.geometry is "cuboid":
            x = vec(add(pwr(2)(self.shp.size[1]),pwr(2)(self.shp.size[2])),0,0)
            y = vec(0,add(pwr(2)(self.shp.size[0]),pwr(2)(self.shp.size[2])),0)
            z = vec(0,0,add(pwr(2)(self.shp.size[0]),pwr(2)(self.shp.size[1])))
            self.tensor = mul(self.m, 1/12, mat(x, y, z))

    def __repr__(self):
        args = [self.cfg, self.shp, self.m, self.q, self.i, self.M]
        return "Solid("+", ".join([repr(a) for a in args])+")"

    def copy(self):
        return eval(repr(self))
```

7.2 Trajectories



mechanics.py

```
class Trajectory:
    """Generates and saves the trajectory of a solid."""

    def __init__(self, solid, data=None):
        checkType([solid], Solid)
        self.solid = solid
        if data is None:
            self.data = {"times": Storage("s", Vector.scalars),
                         "mEtoB": Storage("", Matrix),  # orientation
                         "mBtoE": Storage("", Matrix),  # orientation
                         "p": Storage("m", Vector),     # position
                         "f": Storage("N", Vector),     # force
                         "t": Storage("N.m", Vector),   # torque
                         "s": Storage("m.s-1", Vector), # speed
                         "a": Storage("m.s-2", Vector), # acceleration
                         "av": Storage("s-1", Vector),  # angular velocity
                         "aa": Storage("s-2", Vector)}  # angular acceleration
        else:
            checkType([data], dict)
            self.data = data

        # initial conditions
        self.iniSpe = Quantity(Vector(0, 0, 0), "m.s-1")
        self.iniAngVel = Quantity(Vector(0, 0, 0), "s-1")

        # force and torque
        def zeroForce(time, solid, speed, angularVelocity):
            return Quantity(Vector(0, 0, 0), "N")
        def zeroTorque(time, solid, speed, angularVelocity):
            return Quantity(Vector(0, 0, 0), "N.m")
        self.F = zeroForce
        self.T = zeroTorque

        # decomposition of the process for n-body simulations
        self.waitingData = False

    def __repr__(self):
        return "Trajectory("+repr(self.solid)+", "+repr(self.data)+")"

    def generate(self, timeInterval, dt):
        for i in range(int(div(timeInterval, dt).magnitude)):
            self.calculate(dt)
            self.save()
```

```
    def path(self):
        return Path(self.data["p"])

    def __len__(self):
        return len(self.data["times"])

    def calculate(self, dt):
        """Calculate the new data."""
        dt = Quantity(dt, "s")
        if self.waitingData:
            raise Exception("data not saved")
        self.waitingData = True
        if len(self) is 0:
            self.solid.calculateTensor()
            self.time = -dt
            self.s = self.iniSpe
            self.av = self.iniAngVel
        self.time = add(dt, self.time)
        #1 (force and torque)
        self.f = self.F(self.time, self.solid, self.s, self.av)
        self.t = self.T(self.time, self.solid, self.s, self.av)
        #2 (speed)
        self.a = div(self.f, self.solid.m)
        self.s = add(self.s, mul(dt, self.a))
        #3 (angular velocity)
        M = self.solid.cfg.basis.inside(self.t)
        self.aa = euler(self.solid.tensor, M, self.av)
        self.aa = self.solid.cfg.basis.outside(self.aa)
        self.av = add(self.av, mul(dt, self.aa))
        #4 (position)
        self.p = add(self.solid.cfg.position, mul(dt, self.s))
        #5 (orientation)
        rot = rotMat(mul(dt, self.av))
        self.mBtoE = mul(rot, self.solid.cfg.basis.matrixBtoE)
        self.mEtoB = inv(self.mBtoE)

    def save(self):
        """Save the new data."""
        if not self.waitingData:
            raise Exception('no waiting data')
        self.waitingData = False
        self.data["times"].add(self.time)
        self.data["f"].add(self.f)
        self.data["t"].add(self.t)
        self.data["a"].add(self.a)
        self.data["aa"].add(self.aa)
        self.data["s"].add(self.s)
        self.data["av"].add(self.av)
        self.data["p"].add(self.solid.cfg.position)
        self.data["mBtoE"].add(self.solid.cfg.basis.matrixBtoE)
        self.data["mEtoB"].add(self.solid.cfg.basis.matrixInverse())
        self.solid.cfg = Configuration(self.p, Basis(self.mBtoE, self.mEtoB))

    def copy(self):
        return eval(repr(self))
```

7.3 Functions

7.3.1 Basis creator

mechanics.py

```
def basisCreator(z):
    """Return a direct basis in which 'z' is the ordinate axis."""
    z = Quantity(z).magnitude
    checkType([z], Vector)
    if z[0] == 0 and z[1] == 0 and z[2] >= 0:
        return Basis(identity(3))
    if z[0] == 0 and z[1] == 0 and z[2] < 0:
        x, y, z = Vector(0, 1, 0), Vector(1, 0, 0), Vector(0, 0, -1)
        return Basis(Matrix(x, y, z))
    z = magDiv(z, magNorm(2)(z))
    x = magVecPro(z, Vector(0, 0, 1))
    x = magDiv(x, magNorm(2)(x))
    return Basis(Matrix(x, magVecPro(z, x), z))
```

7.3.2 Euler equations

$$\left.\begin{aligned} I_1\frac{d\Omega_1}{dt} + (I_3 - I_2)\Omega_2\Omega_3 = M_1 \\ I_2\frac{d\Omega_2}{dt} + (I_1 - I_3)\Omega_3\Omega_1 = M_2 \\ I_3\frac{d\Omega_3}{dt} + (I_2 - I_1)\Omega_1\Omega_2 = M_3 \end{aligned}\right\} \quad \begin{pmatrix} \frac{d\Omega_1}{dt} \\ \frac{d\Omega_2}{dt} \\ \frac{d\Omega_3}{dt} \end{pmatrix} \quad \longleftrightarrow \quad \texttt{euler(I, M, w)}$$

mechanics.py

```
def euler(I, M, w):
    """Return the angular acceleration using Euler's rotation equations.
    - I: Inertia tensor (solid basis) /!\ must be a diagonal matrix
    - M: torque (solid basis)
    - w: angular velocity (solid basis)"""
    I = Quantity(I, "kg.m2")
    M = Quantity(M, "N.m")
    w = Quantity(w, "s-1")

    checkType([I.magnitude], Matrix)
    checkType([M.magnitude], Vector)
    checkType([w.magnitude], Vector)

    w1, w2, w3 = w
    M1, M2, M3 = M
    I1, I2, I3 = I[0][0], I[1][1], I[2][2]

    a1 = div(add(mul(w2, w3, sub(I2, I3)), M1), I1)
    a2 = div(add(mul(w3, w1, sub(I3, I1)), M2), I2)
    a3 = div(add(mul(w1, w2, sub(I1, I2)), M3), I3)
    return vec(a1, a2, a3)
```

8 gravitation

gravitation.py

```
G = Quantity(6.67408*10**-11, "N.m2.kg-2") # Newtonian constant of gravitation
```

8.1 Potential

$$\Phi(\vec{r}) \longleftrightarrow -\frac{Gm}{\|\vec{r}-\vec{r}_m\|} \longleftrightarrow \texttt{gravitationnalPotential(solid)(r)}$$

gravitation.py

```
def gravitationalPotential(solid):
    """Return the gravitational potential generated by 'solid'."""
    checkType([solid], Solid)
    def aux(r):
        return -div(mul(G, solid.m), norm(sub(r, solid.cfg.position)))
    return aux
```

8.2 Field

$$\vec{\mathcal{G}}(\vec{r}) \longleftrightarrow -\nabla\Phi(\vec{r}) \longleftrightarrow -\frac{Gm(\vec{r}-\vec{r}_m)}{\|\vec{r}-\vec{r}_m\|^3} \longleftrightarrow \texttt{gravitationnalField(origin)(r)}$$

gravitation.py

```
def gravitationalField(origin):
    """Return the gravitational field according to a potential or a solid."""
    if callable(origin):
        return mul(-1, gradient(origin))
    if isinstance(origin, Solid):
        def aux(r):
            r = sub(r, origin.cfg.position)
            n = norm(r)
            if n.magnitude == 0:
                return Quantity(Vector(0, 0, 0), "m.s-2")
            return div(mul(-1, G, origin.m, r), pwr(3)(n))
        return aux
    raise TypeError(name(origin)+" is not a valid origin")
```

8.3 Force

$$\vec{F} \quad \longleftrightarrow \quad m\vec{g} \quad \longleftrightarrow \quad \texttt{gravitationnalForce(solid, g)}$$

gravitation.py

```
def gravitationalForce(solid, g):
    """Return the gravitationnel force applied on 'solid'."""
    checkType([solid], Solid)
    return mul(solid.m, g(solid.cfg.position))
```

General relativity arrives with the next version...

9 electromagnetism

electromagnetism.py

```
mu_0 = Quantity(4*math.pi*10**-7, "H.m-1") # Vacuum permeability
epsilon_0 = Quantity(8.854187817*10**-12, "F.m-1") # Permittivity
```

9.1 Potentials

9.1.1 Electric potential

$$V(\vec{r}) \quad \longleftrightarrow \quad \frac{q}{4\pi\epsilon_0\|\vec{r}-\vec{r}_q\|} \quad \longleftrightarrow \quad \texttt{electricPotential(solid)(r)}$$

electromagnetism.py

```
def electricPotential(solid):
    """Return the electric potential generated by 'solid'."""
    checkType([solid], Solid)
    def aux(r):
        r = sub(r, origin.cfg.position)
        return div(origin.q, mul(4*math.pi, epsilon_0, norm(r)))
    return aux
```

9.1.2 Magnetic potential

$$\vec{A}(\vec{r}) \quad \longleftrightarrow \quad \frac{\mu_0}{4\pi}\oint_C \frac{Id\vec{l}}{\|\vec{r}-\vec{r}_{d\vec{l}}\|} \quad \longleftrightarrow \quad \texttt{magneticPotential(solid)(r)}$$

electromagnetism.py

```
def magneticPotential(solid):
    """Return the magnetic potential generated by 'solid'."""
    checkType([solid], Solid)
    def aux(r):
        r, p = origin.cfg.inside(Quantity(r, "m")), []
        for i in range(len(origin.shp)-1):
            r_dl = add(mul((1/2), origin.shp[i]), mul((1/2), origin.shp[i+1]))
            dl, n = sub(origin.shp[i+1], origin.shp[i]), norm(sub(r, r_dl))
            if n.magnitude != 0:
                p.append(div(dl, n))
        A = mul(mu_0, 1/(4*math.pi), origin.i, add(p))
        return origin.cfg.basis.outside(A)
    return aux
```

9.2 Fields

9.2.1 Electric field

$$\vec{E}(\vec{r}) \longleftrightarrow -\nabla V(\vec{r}) \longleftrightarrow \frac{q(\vec{r}-\vec{r}_q)}{4\pi\epsilon_0\|\vec{r}-\vec{r}_q\|^3} \longleftrightarrow \texttt{electricField(origin)(r)}$$

electromagnetism.py

```
def electricField(origin):
    """Return the electric field according to a solid or a potential."""
    if callable(origin):
        return mul(-1, gradient(origin))
    if isinstance(origin, Solid):
        def aux(r):
            r = substraction(r, origin.cfg.position)
            n = norm(r)
            if n.magnitude == 0:
                return Quantity(Vector(0, 0, 0), "V.m-1")
            return mul(div(origin.q, mul(4*math.pi, epsilon_0, pwr(3)(n))), r)
        return aux
    raise TypeError(name(origin)+" is not a valid origin")
```

9.2.2 Magnetic field

$$\vec{B}(\vec{r}) \longleftrightarrow (\nabla \times \vec{A})(\vec{r}) \longleftrightarrow \frac{\mu_0}{4\pi}\oint_C \frac{Id\vec{l}\times(\vec{r}-\vec{r}_{d\vec{l}})}{\|\vec{r}-\vec{r}_{d\vec{l}}\|^3} \longleftrightarrow \texttt{magneticField(origin)(r)}$$

electromagnetism.py

```
def magneticField(origin):
    """Return the magnetic field according to a solid or a potential."""
    if callable(origin):
        return curl(origin)
    if isinstance(origin, Solid):
        checkType([origin.shp], Path)
        def aux(r):
            r, p = origin.cfg.inside(Quantity(r, "m")), []
            for i in range(len(origin.shp)-1):
                r_dl = mul((1/2), add(origin.shp[i], origin.shp[i+1]))
                dl, a = sub(origin.shp[i+1], origin.shp[i]), sub(r, r_dl)
                n = norm(a)
                if n.magnitude != 0:
                    p.append(div(vecPro(dl, a), pwr(3)(n)))
            B = mul(mu_0, 1/(4*math.pi), origin.i, add(p))
            return origin.cfg.basis.outside(B)
        return aux
    raise TypeError(name(origin)+" is not a valid origin")
```

9.3 Forces

9.3.1 Lorentz force

$$\vec{F}_L \quad \longleftrightarrow \quad q\left(\vec{E} + \vec{v} \times \vec{B}\right) \quad \longleftrightarrow \quad \texttt{lorentzForce(solid, E, v, B)}$$

electromagnetism.py

```
def lorentzForce(solid, E, v, B):
    """Return the lorentz force applied on 'solid' by 'E' and 'B'."""
    checkType([solid], Solid)
    v = Quantity(v, "m.s-1")
    r = solid.cfg.position
    return mul(solid.q, add(E(r), vecPro(v, B(r))))
```

9.3.2 Magnetic force

$$\vec{F} \quad \longleftrightarrow \quad \nabla\left(\vec{M} \cdot \vec{B}\right) \quad \longleftrightarrow \quad \texttt{magneticForce(solid, B)}$$

electromagnetism.py

```
def magneticForce(solid, B):
    """Return the magnetic force applied on 'solid' by 'B'."""
    checkType([solid], Solid)
    M = solid.cfg.basis.outside(solid.M)
    return gradient(scaPro(M, B))(solid.cfg.position)
```

9.3.3 Magnetic torque

$$\vec{\Gamma} \quad \longleftrightarrow \quad \vec{M} \times \vec{B} \quad \longleftrightarrow \quad \texttt{magneticTorque(solid, B)}$$

electromagnetism.py

```
def magneticTorque(solid, B):
    """Return the torque generated by the magnetic field 'B' on 'solid'."""
    checkType([solid], Solid)
    M = solid.cfg.basis.outside(solid.M)
    return vecPro(M, B(solid.cfg.position))
```

10 supervision

supervision.py

```
import numpy
import time
import datetime
import matplotlib.pyplot
import matplotlib.animation
import mpl_toolkits.mplot3d
```

10.1 Rendering

supervision.py

```
class Rendering:
    """Facilitates dynamic or static rendering of trajectories and fields."""

    ordains = {"f":"Force (N)", "t":"Torque (N.m)",
               "s":"Speed (m/s)", "av":"Angular velocity (rad/s)"}

    defSet = {"nameFig": "Rendering",                  # figure name
              "nameAxe": ["x (m)", "y (m)", "z (m)"], # axis names
              "coloDef": "white",                      # default color
              "timeFra": 10,                           # frame duration (ms)
              "dispAxe": True,                         # display the axes
              "coloFie": "grey",                       # field quivers color
              "normFie": False,                        # normalize the quivers
              "dispCon": True,                         # display configuration
              "dispPos": True,                         # display path
              "dispFor": True,                         # display net force
              "dispTor": True,                         # display net torque
              "sizeDef": None,                         # size of the solid axes
              "coloTra": [],                           # colors of the solids
              "coloAxe": ["green", "green", "green"], # axis colors
              "coloFor": "red",                        # force color
              "coloTor": "blue",                       # torque color
              "coloEnv": [],                           # object colors
              "dispGra": True,                         # display the graphics
              "listGra": ["s", "av"]}                  # choice of graphics

    def __init__(self, display=[], settings=None, comment=None):
        self.field, self.trajectories, self.environment = None, [], []
        self.classify(display)
        self.settings, self.comment = settings, comment
        if settings is None:
            self.settings = Rendering.defSet.copy()
        checkType([self.settings], dict)
        if comment is None:
            date = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            self.comment = ["Date: "+date]
        checkType(self.comment, str)
```

```
    def classify(self, item):
        """Classify the items to display."""
        if isinstance(item, list):
            for element in item:
                self.classify(element)
        elif isinstance(item, Field):
            self.field = item
        elif isinstance(item, Trajectory):
            self.trajectories.append(item)
        elif isinstance(item, (Solid, Path, Volume)):
            self.environment.append(item)
        elif item is not None:
            raise TypeError(str(type(item))+" can not be displayed")

    def __repr__(self):
        display = repr([self.field, self.trajectories, self.environment])
        settings = repr(self.settings)
        comment = repr(self.comment)
        return "Rendering("+display+", "+settings+", "+comment+")"

    def plotGraphic(self, name, graphic):
        """Plot the graphic of ordains 'name'."""
        if not name in self.ordains:
            raise ValueError(str(name)+" is not a valid graphic name")
        graphic.set_ylabel(self.ordains[name])
        for i in range(len(self.trajectories)):
            t = self.trajectories[i]
            graphic.plot([a.magnitude for a in t.data["times"]],
                         [norm(s).magnitude for s in t.data[name]],
                         color=self.settings["coloTra"][i])

    def normCoef(self, a):
        """Return the normalization coefficient for the values in 'a'."""
        def aux(b):
            """Recursive reading."""
            if isinstance(b, Quantity):
                return norm(b).magnitude
            if isinstance(b, (list, Storage)):
                return max([aux(c) for c in b])
            raise TypeError("normCoef not defined on "+str(type(a)))
        m = aux(a)
        if m == 0:
            return 1
        return 1/m

    def size(self, item):
        """Return the characteristic size of the item."""
        if isinstance(item, Solid):
            return norm(item.cfg.position).magnitude+self.size(item.shp)
        if isinstance(item, Volume):
            return max(d.magnitude/2 for d in item.size)
        if isinstance(item, Path):
            return max([norm(p).magnitude for p in item])
        else:
            raise TypeError("size not defined on "+str(type(item)))
```

```
    def prepareData(self, show=False):
        """Prepare the data before displaying."""
        if self.field is None:
            nF, sF = float('inf'), 0
        else:
            nF, self.quivLen = self.field.n[3], self.field.spg
            sF = self.field.spg.magnitude*max(self.field.n[0:2])/2
        if len(self.trajectories) is 0:
            nT, sT = float('inf'), 0
        else:
            nT = min([len(t.data["times"]) for t in self.trajectories])
            p = [self.size(t.solid.shp) for t in self.trajectories]
            sT = max([max([norm(v).magnitude + p[i]
                           for v in self.trajectories[i].data["p"]])
                           for i in range(len(self.trajectories))])
        if len(self.environment) is 0:
            sE = 0
        else:
            sE = max([self.size(item) for item in self.environment])

        # axes size
        self.sizeAxe = max(sF, sT, sE)

        # number of frames
        if self.field is None and len(self.trajectories) is 0:
            self.numbFra = 1
        else:
            self.numbFra = min(nF, nT)

        # size coef
        if self.settings['sizeDef'] is None:
            coefSiz = 0.05*self.sizeAxe
        else:
            coefSiz = self.settings['sizeDef']

        # colors
        n1 = len(self.environment)-len(self.settings["coloEnv"])
        n2 = len(self.trajectories)-len(self.settings["coloTra"])
        if n1 > 0:
            self.settings["coloEnv"].extend(n1*[self.settings["coloDef"]])
        if n2 > 0:
            self.settings["coloTra"].extend(n2*[self.settings["coloDef"]])

        # normalization
        if len(self.trajectories) is not 0 and self.settings["dispFor"]:
            coefFor = self.normCoef([t.data["f"] for t in self.trajectories])
            coefFor = 2*coefSiz*coefFor
        if len(self.trajectories) is not 0 and self.settings["dispTor"]:
            coefTor = self.normCoef([t.data["t"] for t in self.trajectories])
            coefTor = 2*coefSiz*coefTor
        if self.field is not None:
            if self.settings["normFie"]:
                coefFie = 1
            else:
                coefFie = self.normCoef(self.field.stg)
```

```
        # canvas 3D settings
        matplotlib.pyplot.style.use('dark_background')
        name, size = self.settings["nameFig"], (19.20, 10.80)
        self.figure = matplotlib.pyplot.figure(name, figsize=size)
        self.axes = matplotlib.pyplot.subplot(121, projection='3d')
        self.axes.set_xlim(-self.sizeAxe/1.5 ,self.sizeAxe/1.5)
        self.axes.set_ylim(-self.sizeAxe/1.5 ,self.sizeAxe/1.5)
        self.axes.set_zlim(-self.sizeAxe/1.5 ,self.sizeAxe/1.5)
        self.axes.set_xlabel(self.settings["nameAxe"][0])
        self.axes.set_ylabel(self.settings["nameAxe"][1])
        self.axes.set_zlabel(self.settings["nameAxe"][2])
        self.axes.grid(False)
        self.axes.xaxis.pane.fill = False
        self.axes.yaxis.pane.fill = False
        self.axes.zaxis.pane.fill = False
        if self.settings["dispAxe"]:
            matplotlib.pyplot.axis('on')
        else:
            matplotlib.pyplot.axis('off')

        # graphics
        if len(self.trajectories) is not 0 and self.settings["dispGra"]:
            self.graphic1 = matplotlib.pyplot.subplot(222)
            self.graphic2 = matplotlib.pyplot.subplot(224)
            self.plotGraphic(self.settings["listGra"][0], self.graphic1)
            self.plotGraphic(self.settings["listGra"][1], self.graphic2)
            self.graphic1.set_xlabel('Time (s)')
            self.graphic2.set_xlabel('Time (s)')
            start = self.trajectories[0].data["times"][0].magnitude
            end = self.trajectories[0].data["times"][-1].magnitude
            self.dur = (end-start)/10

        # trajectories data conversion
        self.dataTraAxe, self.dataTraPos = [], []
        self.dataTraFor, self.dataTraTor = [], []
        for i in range(len(self.trajectories)):

            # configurations
            if self.settings["dispCon"]:
                l = []
                for k in range(self.numbFra):
                    mBtoE = self.trajectories[i].data["mBtoE"][k]
                    mEtoB = self.trajectories[i].data["mEtoB"][k]
                    o = self.trajectories[i].data["p"][k].magnitude
                    c = Configuration(o, Basis(mBtoE, mEtoB))
                    p = []
                    axes = [add(mul(mBtoE[0], coefSiz), o).magnitude,
                            add(mul(mBtoE[1], coefSiz), o).magnitude,
                            add(mul(mBtoE[2], coefSiz), o).magnitude]
                    for v in axes:
                        x, y = numpy.array([[o[0], v[0]], [o[1], v[1]]])
                        z = numpy.array([[o[2], v[2]], [o[2], v[2]]])
                        p.append([x, y, z])
                    l.append(p)
                self.dataTraAxe.append(l)
```

```
            # paths
            if self.settings["dispPos"]:
                l = []
                for k in range(self.numbFra):
                    positions = [self.trajectories[i].data["p"][j].magnitude
                                 for j in range(k)]
                    x, y = numpy.array([[m[0] for m in positions],
                                        [m[1] for m in positions]])
                    z = numpy.array([[m[2] for m in positions],
                                     [m[2] for m in positions]])
                    l.append([x, y, z])
                self.dataTraPos.append(l)

            # forces
            if self.settings["dispFor"]:
                l = []
                for k in range(self.numbFra):
                    f = mul(coefFor, self.trajectories[i].data["f"][k])
                    p = self.trajectories[i].data["p"][k]
                    f, p = f.magnitude, p.magnitude
                    x, y = numpy.array([[p[0], p[0]+f[0]], [p[1], p[1]+f[1]]])
                    z = numpy.array([[p[2], p[2]+f[2]], [p[2], p[2]+f[2]]])
                    l.append([x, y, z])
                self.dataTraFor.append(l)

            # torques
            if self.settings["dispTor"]:
                l = []
                for k in range(self.numbFra):
                    t = mul(coefTor, self.trajectories[i].data["t"][k])
                    p = self.trajectories[i].data["p"][k]
                    t, p = t.magnitude, p.magnitude
                    x, y = numpy.array([[p[0], p[0]+t[0]], [p[1], p[1]+t[1]]])
                    z = numpy.array([[p[2], p[2]+t[2]], [p[2], p[2]+t[2]]])
                    l.append([x, y, z])
                self.dataTraTor.append(l)

        # field data conversion
        self.dataFiePos = []
        self.dataFieVal = []
        if self.field is not None:
            if self.field.stg.magType is Vector:
                x, y, z = numpy.meshgrid(numpy.arange(0, self.field.n[1], 1.0),
                                         numpy.arange(0, self.field.n[0], 1.0),
                                         numpy.arange(0, self.field.n[2], 1.0))

                # positions
                for i in range(len(x)):
                    for j in range(len(x[0])):
                        for k in range(len(x[0][0])):
                            position = self.field.position(i, j, k)
                            x[i][j][k] = position.magnitude[0]
                            y[i][j][k] = position.magnitude[1]
                            z[i][j][k] = position.magnitude[2]
                self.dataFiePos = [x, y, z]
```

```
                # values
                for t in range(self.numbFra):
                    u = 0*numpy.array(x)
                    v = 0*numpy.array(y)
                    w = 0*numpy.array(z)
                    for i in range(self.field.n[0]):
                        for j in range(self.field.n[1]):
                            for k in range(self.field.n[2]):
                                sample = self.field.value(i, j, k, t)
                                sample = mul(coefFie, sample)
                                u[i][j][k] = sample.magnitude[0]
                                v[i][j][k] = sample.magnitude[1]
                                w[i][j][k] = sample.magnitude[2]
                    self.dataFieVal.append([u, v, w])

        # initialization
        x, y = numpy.array([[],[]])
        z = numpy.array([[],[]])
        if len(self.trajectories) is not 0:
            self.plotTraFor = [self.axes.plot_wireframe(x, y, z)
                               for i in range(len(self.trajectories))]
            self.plotTraTor = [self.axes.plot_wireframe(x, y, z)
                               for i in range(len(self.trajectories))]
            self.plotTraPos = [self.axes.plot_wireframe(x, y, z)
                               for i in range(len(self.trajectories))]
            self.plotTraAxe = [[self.axes.plot_wireframe(x, y, z),
                                self.axes.plot_wireframe(x, y, z),
                                self.axes.plot_wireframe(x, y, z)]
                               for i in range(len(self.trajectories))]
        if self.field is not None:
            self.quivers = self.axes.quiver(x, y, z, x, y, z)

        # animation
        if self.numbFra is not 1:
            self.animation = matplotlib.animation.FuncAnimation(self.figure,
                                self.update, frames=self.numbFra,
                                interval=self.settings["timeFra"])
        # snapshot
        else:
            self.update(0)

        # environment
        for i in range(len(self.environment)):
            self.draw(self.environment[i], self.settings["coloEnv"][i])

        # show
        if show:
            matplotlib.pyplot.show()

    def display(self, items=[]):
        """Start displaying data."""
        self.classify(items)
        infos = "\n".join(["(i) "+a for a in self.comment])
        print("\n[INFORMATIONS]\n"+infos)
        self.prepareData(True)
```

```
    def update(self, k):
        """Update function for matplotlib animation."""
        if len(self.trajectories) is not 0 and self.settings["dispGra"]:
            end = self.trajectories[0].data["times"][k].magnitude
            start = end - self.dur
            self.graphic1.set_xlim(start, end)
            self.graphic2.set_xlim(start, end)
        for i in range(len(self.trajectories)):
            if self.settings["dispCon"]:
                for j in range(3):
                    x, y, z = self.dataTraAxe[i][k][j]
                    self.axes.collections.remove(self.plotTraAxe[i][j])
                    self.plotTraAxe[i][j] = self.axes.plot_wireframe(x, y, z,
                                                color=self.settings["coloAxe"][j])
            if self.settings["dispPos"]:
                x, y, z = self.dataTraPos[i][k]
                self.axes.collections.remove(self.plotTraPos[i])
                self.plotTraPos[i] = self.axes.plot_wireframe(x, y, z,
                                                color=self.settings["coloTra"][i])
            if self.settings["dispFor"]:
                x, y, z = self.dataTraFor[i][k]
                self.axes.collections.remove(self.plotTraFor[i])
                self.plotTraFor[i] = self.axes.plot_wireframe(x, y, z,
                                                color=self.settings["coloFor"])
            if self.settings["dispTor"]:
                x, y, z = self.dataTraTor[i][k]
                self.axes.collections.remove(self.plotTraTor[i])
                self.plotTraTor[i] = self.axes.plot_wireframe(x, y, z,
                                                color=self.settings["coloTor"])
        if self.field is not None:
            x, y, z = self.dataFiePos
            u, v, w = self.dataFieVal[k]
            self.axes.collections.remove(self.quivers)
            self.quivers = self.axes.quiver(x, y, z, u, v, w,
                                            length=self.quivLen.magnitude,
                                            normalize=self.settings["normFie"],
                                            linewidth=.5,
                                            color=self.settings["coloFie"])

    def draw(self, item, color='white', f=None):
        """Draw 'item' on axes."""
        if isinstance(item, Solid):
            self.draw(item.shp, color, item.cfg.outside)
        elif isinstance(item, Path):
            if callable(f):
                vectors = [f(v) for v in item]
            else:
                vectors = [v for v in item]
            x, y = numpy.array([[v.magnitude[0] for v in vectors],
                                [v.magnitude[1] for v in vectors]])
            z = numpy.array([[v.magnitude[2] for v in vectors],
                             [v.magnitude[2] for v in vectors]])
            self.axes.plot_wireframe(x, y, z, color=color)
        else:
            raise Exception(str(type(item))+" can not be displayed")
```

10.2 Useful functions

10.2.1 Console

supervision.py

```
def console(path):
    """Menu to choose and display saved data."""
    def menu1():
        while True:
            print("\n[FILES]\n|0| Quit")
            files = os.listdir(path)
            n = len(str(len(files)))
            for i in range(len(files)):
                print("|"+(n-len(str(i+1)))*" "+str(i+1)+"| "+files[i])
            choice = int(input(">>> Choice: "))
            if choice is 0:
                break
            menu2(eval(load(path+'/'+files[int(choice-1)])))
    def menu2(item):
        while True:
            print("\n[ACTION]\n|0| Back\n|1| Display\n|2| Export")
            choice = int(input(">>> Choice: "))
            if choice is 0:
                break
            if choice is 1:
                item.display()
            if choice is 2:
                menu3(item)
    def menu3(item):
        settings = [0, 0, 0]
        item.prepareData()
        while True:
            print("\n[SETTINGS]")
            print("(i) Number of frames: "+str(item.numbFra))
            print("|0| Back")
            print("|1| Vertical angle: "+str(settings[0])+"\u00B0")
            print("|2| Horizontal angle: "+str(settings[1])+"\u00B0")
            print("|3| Frame: "+str(settings[2])+" (for PDF)")
            print("|4| Export as PDF")
            print("|5| Export as MP4")
            choice = int(input(">>> Choice: "))
            if choice is 0:
                break
            if choice in [1, 2, 3]:
                settings[choice-1] = int(input(">>> New value: "))
            date = datetime.datetime.now().strftime("%Y-%m-%d-%H%M%S")
            item.axes.view_init(settings[0], settings[1])
            if choice is 4:
                item.update(settings[2])
                item.figure.savefig(date+".pdf")
            if choice is 5:
                Writer = matplotlib.animation.writers['ffmpeg']
                writer = Writer(fps=24)
                item.animation.save(date+".mp4", writer=writer)
    menu1()
```

10.2.2 Timer

```
timer = Timer(n)
for i in range(n):
    timer.before()
    # instructions
    timer.after()
```

⟶

```
[FIELD CALCULATION]
(i) 5 minutes
(i) 4 minutes
(i) 3 minutes
(i) 2 minutes
```

supervision.py

```python
class Timer:
    """Gives an estimate of the remaining calculation time."""

    def __init__(self, n):
        self.n = n # number of sequences
        self.c = 0 # counter
        self.dt = 0 # average duration of a sequence
        self.rm = 0 # remaining minutes
        self.rh = 0 # remaining hours

    def before(self):
        """Must be called at the beginning of the sequence."""
        self.t1 = time.time()

    def after(self):
        """Must be called at the end of the sequence."""
        self.dt = (self.c*self.dt + time.time()-self.t1)/(self.c+1)
        self.c += 1
        newRM = int(self.dt*(self.n-self.c)/60)+1
        if newRM >= 60:
            newRH = newRM//60 + 1
            if newRH != self.rh:
                self.rh = newRH
                if self.rh is 1:
                    print("(i) ", self.rh, "hour")
                else:
                    print("(i) ", self.rh, "hours")
        elif newRM is not self.rm and self.n is not 1:
            self.rm = newRM
            if self.rm is 1:
                print("(i)", self.rm, "minute")
            else:
                print("(i)", self.rm, "minutes")
```

10.3 Sorts

10.3.1 Merge sort

supervision.py

```
def mergeSort(compare=lambda a, b: a < b):
    """Not in-place and stable."""
    def merge(L1, L2):
        L, i1, i2, n1, n2 = [], 0, 0, len(L1), len(L2)
        for i in range(n1+n2):
            if n2 == i2 or i1 < n1 and compare(L1[i1], L2[i2]):
                L.append(L1[i1])
                i1 += 1
            else:
                L.append(L2[i2])
                i2 += 1
        return L
    def sort(L):
        if len(L) <= 1:
            return L[:]
        m = len(L)//2
        return merge(sort(L[:m]), sort(L[m:]))
    return sort
```

10.3.2 Quick sort

supervision.py

```
def quickSort(compare=lambda a, b: a < b):
    """In-place and not stable."""
    def partition(L, g, d):
        pivot , m = L[g], g
        for i in range(g+1, d):
            if compare(L[i], pivot):
                m += 1
                if i > m:
                    L[i], L[m] = L[m], L[i]
        if m > g:
            L[m], L[g] = L[g], L[m]
        return m
    def sort(L):
        def aux(g, d):
            if g < d-1:
                m = partition(L, g, d)
                aux(g, m)
                aux(m+1, d)
        aux(0, len(L))
        return(L)
    return sort
```