Partikelsystem:

N partiklar (punktmassor) där partikel i (index i) har massan m^i,
position $\vec{x}^i = \begin{pmatrix} \vec{x}^i{}_x \\ \vec{x}^i{}_y \end{pmatrix}$ och hastigheten $\vec{v}^i = \begin{pmatrix} \vec{v}^i{}_x \\ \vec{v}^i{}_y \end{pmatrix}$ (i 2D).

Newtons andra lag:

$$\dot{\vec{x}}^i = \vec{v}^i$$

$$\dot{\vec{v}}^i = \frac{\vec{F}^i}{m^i}$$

Parvisa krafter:

$$F^i = \sum_{j=0}^{N} F^{ij}$$

Generell ODE-form (vektorform):

$$\vec{u} = \begin{pmatrix} \vec{x}_x \\ \vec{x}_y \\ \vec{v}_x \\ \vec{v}_y \end{pmatrix} \qquad \vec{f} = \begin{pmatrix} \vec{v}_x \\ \vec{v}_y \\ \vec{F}_x \\ \vec{F}_y \end{pmatrix} \qquad \dot{\vec{u}} = \vec{f}(\vec{u})$$

Lös med generell ODE-lösare (`solve()` från modul 3, tidsstegning)

Exempelvis:

Trapetsmetoden: $u^{n+1} = u^n + \frac{k}{2}\left(f(u^n) + f(u^{n+1})\right)$

Bakåt Euler: $u^{n+1} = u^n + kf(u^{n+1})$

```python
# Initial values
x[0, 0] = 0.0  # Particle 0 starts in x=0
[...]

# Pack values into u
u[0 * M : 1 * M] = x[:, 0]
u[1 * M : 2 * M] = x[:, 1]
u[2 * M : 3 * M] = v[:, 0]
u[3 * M : 4 * M] = v[:, 1]

def f3body(t, u):
    # Unpack values into x and v
    x[:, 0] = u[0 * M : 1 * M]
    x[:, 1] = u[1 * M : 2 * M]
    v[:, 0] = u[2 * M : 3 * M]
    v[:, 1] = u[3 * M : 4 * M]

    a = zeros((M, 2))

    m = 1.0     # Mass
    E = 100.0   # Stiffness coefficient
    B = 4.0     # Damping coefficient
    L = 2.0     # Spring rest length

    for i0 in range(0, M):
        for i1 in range(0, M):
            # Don't compute force with self
            if (i0 == i1):
                continue

            r = norm(x[i1, :] - x[i0, :])
            e = (x[i1, :] - x[i0, :]) / r
            vr = v[i1, :] - v[i0, :]

            F = E * (r - L) * e     # Elastic spring force
            D = B * dot(vr, e) * e  # Damping spring force

            # Gravity force
            G = 0 # Add this yourself

            a[i0, :] += (F + D + G) / m

    # Pack values into fval
    fval = zeros(4 * M)
    fval[0 * M : 1 * M] = v[:, 0] # (velocities)
    fval[1 * M : 2 * M] = v[:, 1]
    fval[2 * M : 3 * M] = a[:, 0] # (forces / mass)
    fval[3 * M : 4 * M] = a[:, 1]

    return fval
```

Fixpunktsform:

Skriv ekvationen $R(q) = 0$ i fixpunktsform: $q = g(q)$
(Det finns hur många former som helst.)

Fixpunktsiteration: $q_{m+1} = g(q_m)$

I vårt exempel hade vi redan en fixpunktsform:

$$q_{m+1} = g(q) = u^n + kf(q_m)$$

(Notera olika index for tidssteg och iteration.)

Konvergerar iterationen?

```
def fixedpoint(g, q0):
    # Choose initial guess, parameters
    q = q0
    TOL = 1.0e-8
    res = 1.0

    # Iterate until convergence
    while (res > TOL):
        s = g(q)
        res = norm(s - q)
        q = s

    return x
```

Iterationen kan bete sig på olika sätt:

Snabbt konvergerandes

Divergerandes

Långsamt konvergerandes

Långsamt konvergerandes och alternaterande

Contraction mapping:

$f(0)=0$ Skriv som fixpunktsiteration

$u^{n+1}=g(u^n)$

$e^{n+1}=u^{n+1}-u^n=g(u^n)-g(u^{n-1})$

$|e^{n+1}|=|g(u^n)-g(u^{n+1})|$ Om g är lipschitz-kontinuerlig

$|e^{n+1}|=|g(u^n)-g(u^{n+1})|\leq L|u^n-u^{n-1}|$

Läs 76-77 kap. $|e^{n+1}|\leq L|e^n|$

L måste vara mindre än 1 i fixpunktsmetoden för att få konvergens. [SF1613]

$|g'(u)|<1$

Om L = 0 får vi Newtons metod.

Newtons metod konvergerar (nästan) alltid.

Generell algebraisk ekvationslösning:

R(q) = 0

Exempel: $R(q) = x^2 - 2 = 0$ (roten ur 2)

Även system (se linjära system senare till exempel)

Newtons metod:

Vi kan skriva en generell fixpunktsform för ekvationen $R(q) = 0$:

$$q = g(q) = q - \alpha R(q)$$

Konvergens:

$$g'(q) = 1 - \alpha' R(q) - \alpha R'(q)$$

Använd $R(q) = 0$:

$$g'(q) = 1 - \alpha R'(q)$$

Optimal metod:

$$g'(q) = 0$$

$$\Downarrow$$

$$1 - \alpha R'(q) = 0$$

$$\Downarrow$$

$$\alpha = \frac{1}{R(q)}$$

Alltså: Newtons metod:

$$q = q - \frac{R(q)}{R'(q)}$$

Samma sak, men $R'(q)$ är en matris:

$$q_{m+1} = q_m - \frac{R(q_m)}{R'(q_m)}$$

$$J = R'(q_m) \quad \Rightarrow \quad J \cdot q_{m+1} = J \cdot q_m - R(q_m)$$

(J är en jacobimatris)

```python
def newton(f, x0):
    class LocalData:
        def __init__(self):
            self.f = f

    def g(x, iter, data):
        f = data.f

        # Compute Jacobian
        J = jacobian(f, x)
        # Compute right hand side
        r = dot(J, x) - f(x)
        # Solve linear system
        y = solve(J, r)

        return y

    data = LocalData()

    # Iterate the Newton g(x)
    return fixedpoint(g, x0, data)
```

```python
def newton_fixedpoint_adapter(g):
    def R(x):
        return x - g(x)

    return R

def g(u):
    t = t0 + k
    return u0 + 0.5 * k * f(t0, u0) + 0.5 * k * f(t, u)

fnewton = newton_fixedpoint_adapter(g)
return newton(fnewton, u0)
```

Jacobi-iteration:

$$\mathbf{A}\vec{x}=\vec{b} \quad \Rightarrow \quad \{\mathbf{A}=\mathbf{D}+\mathbf{M}\} \quad \Rightarrow \quad x=\mathbf{D}^{-1}(-\mathbf{M}\vec{x}+\vec{b})=\vec{g}(\vec{x})$$

$$\|\vec{g}\,'\|=\|\mathbf{D}^{-1}\mathbf{M}\|<1$$

Fungerar endast för icke-linjära system.

Steepest Descent:

$$\vec{x}=\vec{x}-\alpha(\mathbf{A}\vec{x}-\vec{b})=\vec{g}(\vec{x}) \qquad (\vec{r}=\mathbf{A}\vec{x}-\vec{b})$$

$$\|\vec{g}\,'\|=\|\mathbf{I}-\alpha\mathbf{A}\|=0 \quad \Rightarrow \quad \alpha=\frac{\langle\vec{r};\,\vec{r}\rangle}{\langle\vec{r};\,\mathbf{A}\vec{r}\rangle}$$

Conjugate Gradient:

Samma som Steepest Decent, men man räknar $\vec{r}$ som en ortogonalsering mot Krylov-vektorer/rum.