Studies of Nonlinear Problems by Fermi, Pasta, Ulam, and Tsingou

Their studies were first described in Los Alamos Report LA-1940 dated May 1955.

LA-1940 Page 4

“We have, therefore, a dynamical system of 64 particles with forces acting between neighbors with fixed end points. If \( x_i \) denotes the displacement of the \( i \)-th point from its original position, and \( \alpha \) denotes the coefficient of the quadratic term in the force between the neighboring mass points and \( \beta \) that of the cubic term, the equations were either

\[
\ddot{x}_i = (x_{i+1} + x_{i-1} - 2x_i) + \alpha \left[ (x_{i+1} - x_i)^2 - (x_i - x_{i-1})^2 \right] \quad i = 1, 2, \ldots, 64 ,
\]

or

\[
\ddot{x}_i = (x_{i+1} + x_{i-1} - 2x_i) + \beta \left[ (x_{i+1} - x_i)^3 - (x_i - x_{i-1})^3 \right] \quad i = 1, 2, \ldots, 64 .
\]

\( \alpha \) and \( \beta \) were chosen so that at the maximum displacement the nonlinear term was small, e.g., of the order of one-tenth of the linear term.

Another case studied recently was

\[
\ddot{x}_i = \delta_1(x_{i+1} - x_i) - \delta_2(x_i - x_{i-1}) + c ,
\]

where the parameters \( \delta_1, \delta_2, c \) were not constant but assumed different values depending on whether or not the quantities in parentheses were less than or greater than a certain value fixed in advance.”
The solution to the corresponding linear problem is a periodic vibration of the string. If the initial position of the string is, say, a single sine wave, the string will oscillate in this mode indefinitely.
the string in a simple configuration, for example in the first mode (or in other problems, starting with a combination of a few low modes), the purpose of our computations was to see how, due to nonlinear forces perturbing the periodic linear solution, the string would assume more and more complicated shapes, and, for $t$ tending to infinity, would get into states where all the Fourier modes acquire increasing importance. In order to see this, the shape of the string, that is to say, $x$ as a function of $i$ and the kinetic energy as a function of $i$ were analyzed periodically in Fourier series. Since the problem can be considered one of dynamics, this analysis amounts to a Lagrangian change of variables: instead of the original $\dot{x}_i$ and $x_i$, $i = 1, 2, \ldots 64$, we may introduce $a_k$ and $\dot{a}_k$, $k = 1, 2, \ldots 64$, where

$$a_k = \sum x_i \sin \frac{ik\pi}{64}.$$  
(4)

The sum of kinetic and potential energies in the problem with a quadratic force is

$$E_{x_i}^{\text{kin}} + E_{x_i}^{\text{pot}} = \frac{1}{2} \dot{x}_i^2 + \frac{(x_{i+1} - x_i)^2 + (x_i - x_{i-1})^2}{2}$$  
(5a)

$$E_{a_k}^{\text{kin}} + E_{a_k}^{\text{pot}} = \frac{1}{2} \dot{a}_k^2 + 2a_k^2 \sin^2 \frac{\pi k}{128}$$  
(5b)

if we neglect the contributions to the potential energy from the quadratic or higher order terms in the force. This amounts in our case to at most a few per cent."

Applying the DST to the FPU Chain

The problem can be discretized by replacing the continuous string of length $L$ with $N + 1$ points $x_0 = 0, x_1, x_2, \ldots, x_N = L$, the two end points $x_0$ and $x_N$ being fixed. There is a unit mass at each of the $N - 1$
interior points connected to its left and right neighbors by $N$ nonlinear springs.

```cpp
vector<double> Fourier_sine_transform(
    const vector<double>& x,
    bool inverse=false)
{
    vector<double> a(x.size());
    int N = x.size() - 1;
    const double pi = 4 * atan(1.0);
    for (int k = 1; k < N; ++k)
        for (int i = 1; i < N; ++i)
            a[k] += x[i] * sin(i * k * pi / double(N));
    if (inverse)
        for (int k = 1; k < N; ++k)
            a[k] *= 2.0 / double(N);
    return a;
}
```

```python
fpu.py

vector<double> Fourier_sine_transform(  
    const vector<double>& x,  
    bool inverse=false)
{
    vector<double> a(x.size());
    int N = x.size() - 1;
    const double pi = 4 * atan(1.0);
    for (int k = 1; k < N; ++k)
        for (int i = 1; i < N; ++i)
            a[k] += x[i] * sin(i * k * pi / double(N));
    if (inverse)
        for (int k = 1; k < N; ++k)
            a[k] *= 2.0 / double(N);
    return a;
}
```
def Fourier_sine_transform(x, inverse=False):
    a = [0.0 for i in range(len(x))]
    N = len(x) - 1
    for k in range(1, N):
        for i in range(1, N):
            a[k] += x[i] * sin(i * k * pi / float(N))
    if inverse:
        for k in range(1, N):
            a[k] *= 2.0 / float(N)
    return a

double E(
    const vector<double>& x,
    const vector<double>& dxdt,
    const double alpha)
{
    int N = x.size() - 1;
    double E_kin = 0;
    for (int i = 1; i < N; ++i)
E_kin += dxdt[i] * dxdt[i] / 2.0;
double E_pot = 0;
for (int i = 1; i <= N; ++i) {
    double dx = x[i] - x[i-1];
    E_pot += pow(dx, 2.0) / 2.0 + alpha * pow(dx, 3.0) / 3.0;
}
return E_kin + E_pot;

vector<double> E_mode(
    const vector<double>& x,
    const vector<double>& dxdt)
{
    int N = x.size() - 1;
    vector<double> a = Fourier_sine_transform(x);
    vector<double> dadt = Fourier_sine_transform(dxdt);
    vector<double> E_mode = vector<double>(x.size());
    const double pi = 4 * atan(1.0);
    for (int k = 1; k < N; ++k) {
        double E_kin = 0.5 * pow(dadt[k], 2.0);
        double E_pot = 2 * pow(a[k] * sin(k * pi / (2.0 * N)), 2.0);
        E_mode[k] = E_kin + E_pot;
def E(x, dxdt, alpha):
    N = len(x) - 1
    E_kin = 0.0
    for i in range(1, N):
        E_kin += dxdt[i]**2 / 2.0
    E_pot = 0.0
    for i in range(1, N+1):
        dx = x[i] - x[i-1]
        E_pot += dx**2 / 2.0 + alpha * dx**3 / 3.0
    return E_kin + E_pot

def E_mode(x, dxdt):
    N = len(x) - 1
    a = Fourier_sine_transform(x)
    dadt = Fourier_sine_transform(dxdt)
E_mode = [0.0 for i in range(len(x))]
for k in range(1, N):
    E_kin = 0.5 * dadt[k]**2
    E_pot = 2 * (a[k] * sin(k * pi / (2.0 * N)))**2
    E_mode[k] = E_kin + E_pot
return E_mode

void compute_accelerations(
    const vector<double>& x,
    vector<double>& d2xdt2,
    const double alpha)
{
    int N = x.size() - 1;
    for (int i = 1; i < N; ++i) {
        double dx_plus = x[i+1] - x[i], dx_minus = x[i] - x[i-1];
        d2xdt2[i] = dx_plus - dx_minus +
            alpha * (pow(dx_plus, 2.0) - pow(dx_minus, 2.0));
    }
}
void initialize(
    const vector<double>& a,
    vector<double>& x,
    vector<double>& dxdt,
    vector<double>& d2xdt2,
    const double alpha)
{
    int N = x.size() - 1;
    // zero all components
    for (int i = 0; i <= N; ++i)
        x[i] = dxdt[i] = d2xdt2[i] = 0.0;

    // inverse sine transform of mode amplitudes
    bool inverse = true;
    vector<double> dx = Fourier_sine_transform(a, inverse);
    for (int i = 1; i < N; ++i)
        x[i] = dx[i];

    // adjust amplitude
    double x_max = 0;
    for (int i = 1; i < N; ++i) {
        x_max = std::max(x_max, x[i]);
    }
double ax = abs(x[i]);
if (ax > x_max)
    x_max = ax;
}
for (int i = 1; i < N; ++i)
    x[i] /= x_max;

compute_accelerations(x, d2xdt2, alpha);
}

def compute_accelerations(x, d2xdt2, alpha):
    N = len(x) - 1
    for i in range(1, N):
        dx_plus = x[i+1] - x[i]
        dx_minus = x[i] - x[i-1]
        d2xdt2[i] = dx_plus - dx_minus + alpha * (dx_plus**2 - dx_minus**2)

def initialize(a, x, dxdt, d2xdt2, alpha):
    N = len(x) - 1
# zero all components
for i in range(N+1):
    x[i] = dxdt[i] = d2xdt2[i] = 0.0

# inverse sine transform of mode amplitudes
inverse = True
dx = Fourier_sine_transform(a, inverse)
for i in range(1, N):
    x[i] = dx[i]

# adjust amplitude
x_max = 0.0
for i in range(1, N):
    ax = abs(x[i])
    if ax > x_max:
        x_max = ax
for i in range(1, N):
    x[i] /= x_max

compute_accelerations(x, d2xdt2, alpha)
The Velocity-Verlet Algorithm

This algorithm is commonly used in molecular dynamics simulations because it is *symplectic* and conserves the total energy of a system of particles obeying Hamilton’s equations better than general purpose algorithms such as Runge-Kutta.

Consider a “molecule” subject to a conservative force $F = ma$ in one dimension, expand the solution in a Taylor series

$$x(t + \delta t) = x(t) + v(t)\delta t + \frac{1}{2}a(t)\delta t^2 + \frac{1}{6}\dot{a}(t)\delta t^3 + \mathcal{O}(\delta t^4)$$

and add

$$x(t - \delta t) = x(t) - v(t)\delta t + \frac{1}{2}a(t)\delta t^2 - \frac{1}{6}\dot{a}(t)\delta t^3 + \mathcal{O}(\delta t^4)$$

to obtain the Verlet algorithm

$$x(t + \delta t) = 2x(t) - x(t - \delta t) + a(t)\delta t^2 + \mathcal{O}(\delta t^4)$$

$$v(t) = \frac{x(t + \delta t) - x(t - \delta t)}{2\delta t} + \mathcal{O}(\delta t^2)$$

This algorithm is *not self-starting* because it involves 3 points in time.

The Velocity-Verlet algorithm is self-starting with comparable accuracy

$$x(t + \delta t) = x(t) + v(t)\delta t + \frac{1}{2}a(t)\delta t^2 + \mathcal{O}(\delta t^4)$$

$$v(t + \delta t) = v(t) + \frac{1}{2}[a(t) + a(t + \delta t)]\delta t + \mathcal{O}(\delta t^2)$$
This is a two-step algorithm. The next position is computed using $x(t)$, $v(t)$, $a(t)$, and used to compute $a(t + \delta t)$ and hence $v(t + \delta t)$.

```cpp
void velocity_Verlet(
    const double dt,
    vector<double>& x,
    vector<double>& dxdt,
    vector<double>& d2xdt2,
    const double alpha)
{
    int N = x.size() - 1;
    for (int i = 1; i < N; ++i) {
        x[i] += dxdt[i] * dt + 0.5 * d2xdt2[i] * dt * dt;
        dxdt[i] += 0.5 * d2xdt2[i] * dt;
    }
    compute_accelerations(x, d2xdt2, alpha);
    for (int i = 1; i < N; ++i)
        dxdt[i] += 0.5 * d2xdt2[i] * dt;
}
```
def velocity_Verlet(dt, x, dxdt, d2dxt2, alpha):
    N = len(x) - 1
    for i in range(1, N):
        x[i] += dxdt[i] * dt + 0.5 * d2dxt2[i] * dt**2
        dxdt[i] += 0.5 * d2dxt2[i] * dt
    compute_accelerations(x, d2dxt2, alpha)
    for i in range(1, N):
        dxdt[i] += 0.5 * d2dxt2[i] * dt

LA-1940

Page 8: “The general features of our computation are these: in each problem, the system was started from
rest at time $t = 0$.”

Page 12:

int main()
Fig. 1. The quantity plotted is the energy (kinetic plus potential in each of the first five modes). The units of energy are arbitrary. $N = 32$; $\alpha = 1/4$; $\delta t^2 = 1/8$. The initial form of the string was a single sine wave. The higher modes never exceeded in energy 20 of our units. About 30,000 computation cycles were calculated.
cout << " Studies of Nonlinear Problems by Fermi, Pasta, Ulam, Tsingou"
<< " -------------------------------

int N = 32;
double alpha = 1.0 / 4.0;
double delta_t = sqrt(1.0 / 8.0);
cout << " N = " << N << 't' << " alpha = " << alpha << 't'
     << " delta_t = " << delta_t << '
;

int computation_steps_per_cycle = 100;
double dt = delta_t / computation_steps_per_cycle;
cout << " integration time step dt = " << dt << endl;

// create vectors for position, velocity, acceleration, mode amplitudes
vector<double> x = vector<double>(N+1);
vector<double> dxdt = x, d2xdt2 = x, a = x;

// initialize with lowest eigenmode of the linear chain
a[1] = 1;
initialize(a, x, dxdt, d2xdt2, alpha);
vector<double> E_a = E_mode(x, dxdt);
double E_scale = E_a[1] / 300.0;

ofstream ofs_fig1("fpu1.dat"), ofs_fig8("fpu8.dat");

for (int i = 0; i <= N; i++)
    ofs_fig8 << i << ' ' << x[i] << '
';

print(" Studies of Nonlinear Problems by Fermi, Pasta, Ulam, Tsingou")
print(" ------------------------------------------------------------")
N = 32
alpha = 1.0 / 4.0
delta_t = sqrt(1.0 / 8.0)
print(" N = ", N, " alpha = ", alpha, " delta_t = ", delta_t)

computation_steps_per_cycle = 100
dt = delta_t / computation_steps_per_cycle
print(" integration time step dt = ", dt)
# create vectors for position, velocity, acceleration, mode amplitudes
x = [0.0 for i in range(N+1)]
dxdt = [0.0 for i in range(N+1)]
d2xdt2 = [0.0] * (N + 1)
a = [0.0] * (N + 1)

# initialize with lowest eigenmode of the linear chain
a[1] = 1.0
initialize(a, x, dxdt, d2xdt2, alpha)

E_a = E_mode(x, dxdt)
E_scale = E_a[1] / 300.0

fpu1 = open("fpu1.data", "w")
fpu8 = open("fpu8.data", "w")

for i in range(N+1):
    fpu8.write(str(i) + '	' + str(x[i]) + '
')
Page 8: “The general features of our computation are these: in each problem, the system was started from rest at time $t = 0$.”

Page 19:

```cpp
int nc = 1000;
int computation_cycles = 30 * nc;
for (int cycle = 0; cycle <= computation_cycles; ++cycle) {
    // save total and mode energies every nc cycles
    if (cycle % (nc/10) == 0) {
        double t = cycle / double(nc);
        double energy = E(x, dxdt, alpha);
        E_a = E_mode(x, dxdt);
        cout << " " << t << '\t' << energy;
        ofs_fig1 << t << '\t' << energy;
        for (int k = 1; k <= 5; ++k) {
            cout << '\t' << E_a[k] / E_scale;
        }
    }
}
```
Fig. 8. This drawing shows not the energy but the actual shapes, i.e., the displacement of the string at various times (in cycles) indicated on each curve.
ofs_fig1 << 't' << E_a[k] / E_scale;
}
// sum energies in modes 6 through N
energy = 0;
for (int k = 6; k <= N; ++k)
    energy += E_a[k] * E_scale;
cout << 't' << energy << '
';
ofs_fig1 << 't' << energy << '
';
}

if (cycle == 1000 || cycle == 10000 || cycle == 12000 ||
cycle == 14000 || cycle == 19000 || cycle == 22000 ||
cycle == 28311)
{
    ofs_fig8 << '
';
    for (int i = 0; i <= N; i++)
        ofs_fig8 << i << 't' << x[i] << '
';
}

for (int step = 0; step < computation_steps_per_cycle; ++step)
    velocity_Verlet(dt, x, dxdt, d2xdt2, alpha);
```python
nc = 1000
computation_cycles = 30 * nc
for cycle in range(computation_cycles+1):
    # save total and mode energies every nc cycles
    if cycle % (nc//10) == 0:
        t = cycle / float(nc)
        energy = E(x, dxdt, alpha)
        E_a = E_mode(x, dxdt)
        data = str(t) + '	' + str(energy)
        for k in range(1, 6):
            data += '	' + str(E_a[k] / E_scale)
        # sum energies in modes 6 through N
        energy = 0;
        for k in range(6, N+1):
            energy += E_a[k]
```

---

```python
phy.py
```
energy += E_a[k] * E_scale;
data += \t' + str(energy)
print(data)
fpu1.write(data + '\n')

if cycle in [1000, 10000, 12000, 14000, 19000, 22000, 28311]:
fpu8.write('\n')
for i in range(N+1):
fpu8.write(str(i) + '\t' + str(x[i]) + '\n')

for step in range(computation_steps_per_cycle):
    velocity_Verlet(dt, x, dxdt, d2xdt2, alpha)