Sequential Processing in Nature, ‘Anything but’ in Scientific Computation

an observation

Dominic Jones

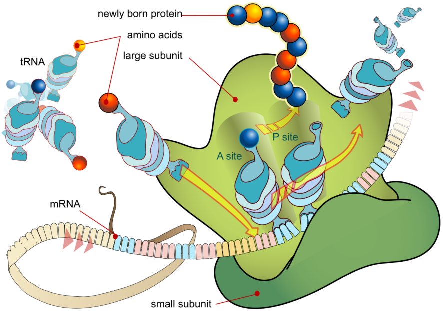
Netherhall House, London

January 2018

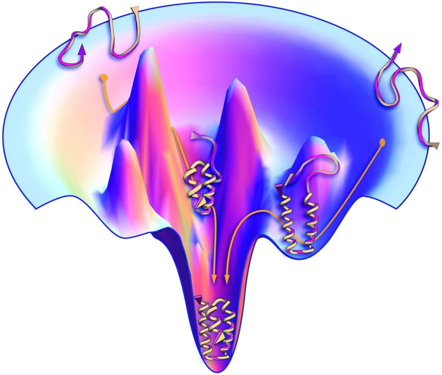
Nature - ‘It should just work’

Protein folding

Synthesising the chain



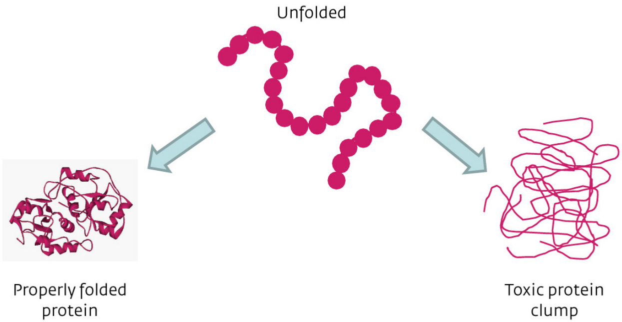
Finding the global minimum



Folding occurs in microseconds, but cannot be predicted

Abnormal folding

Deformed proteins cannot be mended



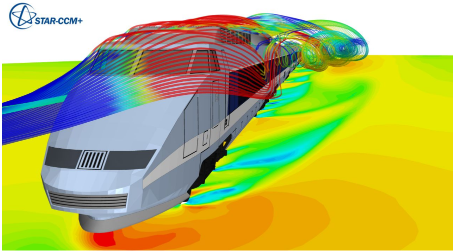
In brief

1. From a chain of amino acids a specific highly complex shape is required
2. There is no apparent error correction in the process (or very little)
3. It is usually successful
4. When it isn’t, Creutzfeldt-Jakob disease, Parkinson’s, Alzheimer’s, etc

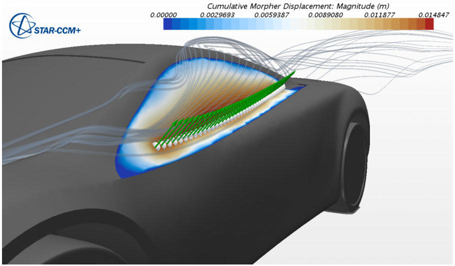
Scientific computation - ‘`T = abs(T)`’

Attempts at prediction and projection

Numerical approximation



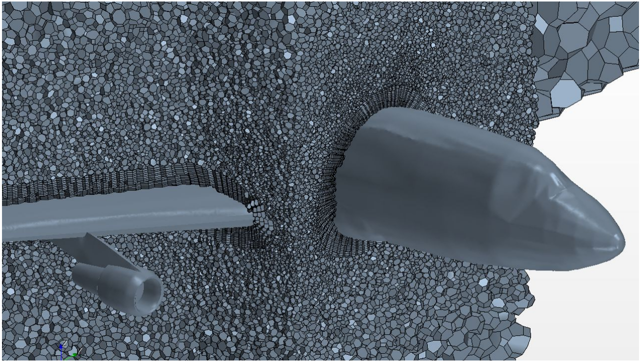
Local sensitivities



Iterative, discretised and linearised algorithms

All starts with a graph (mesh)

Resolve change at the small scales



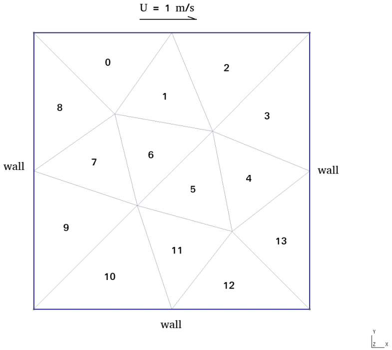
Mesh implies graph, implies matrix, implies matrix inversion

Matrix inversion: scaling up

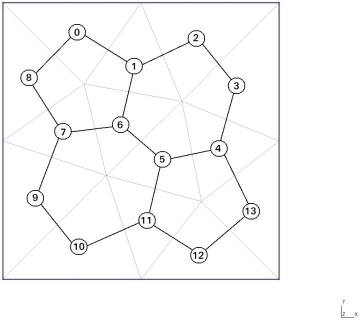
```
void inverse(int n, float a[][2*n_max])
{
  for (int i = 0; i < n; i++)
    for (int j = n; j < 2*n; j++)
      a[i][j] = (i == j-n? 1: 0);

  for (int i = 0; i < n; i++) {          // linear
    float aii = a[i][i];
    for (int j = i; j < 2*n; j++)
      a[i][j] = a[i][j] / aii;

    for (int j = 0; j < n; j++) {        // quadratic!
      if (i != j) {
        float aji = a[j][i];
        for (int k = 0; k < 2*n; k++)    // cubic!!
          a[j][k] = a[j][k] - aji * a[i][k]; }}}}
```

Computation: slow but compact

Geometry and mesh



Connectivity graph



Topology Representation

Symmetric, sparse (48 non-zeros), irregular

$$
\begin{bmatrix}
a_{0,0} & a_{0,1} & & & & & & & a_{0,8} & & & & & \\
a_{1,0} & a_{1,1} & a_{1,2} & & & & a_{1,6} & & & & & & & \\
 & a_{2,1} & a_{2,2} & a_{2,3} & & & & & & & & & & \\
 & & a_{3,2} & a_{3,3} & a_{3,4} & & & & & & & & & \\
 & & & a_{4,3} & a_{4,4} & a_{4,5} & & & & & & & & a_{4,13} \\
 & & & & a_{5,4} & a_{5,5} & a_{5,6} & & & & & a_{5,11} & & \\
 & a_{6,1} & & & & a_{6,5} & a_{6,6} & a_{6,7} & & & & & & \\
 & & & & & & a_{7,6} & a_{7,7} & a_{7,8} & a_{7,9} & & & & \\
a_{8,0} & & & & & & & a_{8,7} & a_{8,8} & & & & & \\
 & & & & & & & a_{9,7} & & a_{9,9} & a_{9,10} & & & \\
 & & & & & & & & & a_{10,9} & a_{10,10} & a_{10,11} & & \\
 & & & & & a_{11,5} & & & & & a_{11,10} & a_{11,11} & a_{11,12} & \\
 & & & & & & & & & & & a_{12,11} & a_{12,12} & a_{12,13} \\
 & & & & a_{13,4} & & & & & & & & a_{13,12} & a_{13,13}
\end{bmatrix}
$$

Sparsity and Indirection

- Dense storage: 196 values, 24% efficient
- Direct access to values

```
float[14][14] A;
A[2][3] = 3.142;
```

- Compressed row storage: 111 values, 56% efficient
- Requires indirection to access values (10x slower)

```
int[15] IA = [0, 3, 7, 10, ...];
int[48] JA = [0, 1, 8,  0, 1, 2, 6,  1, 2, 3,  ...];
float[48] A;

A[JA[IA[2]+2]] = 3.142; // i.e. A[2][3] = 3.142;
```

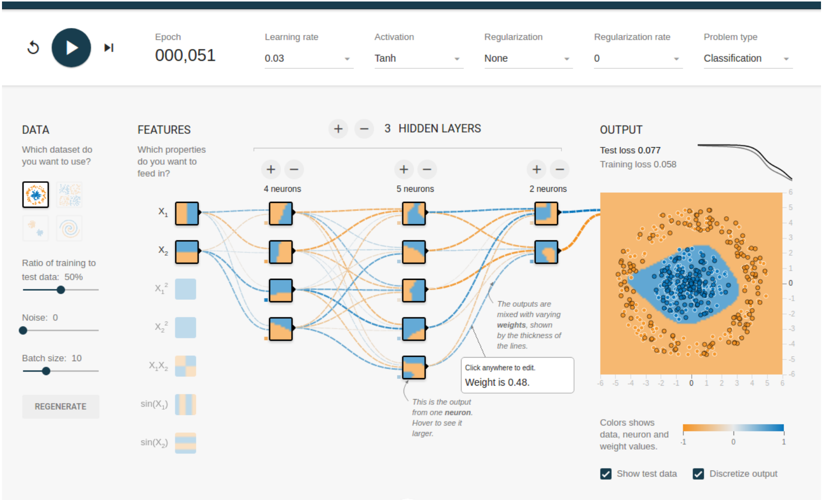
In brief

1. Many scientific algorithms have graphs and matrices, and perform matrix inversion at their core
2. They are often woefully inefficient due to the requirement to use compressed storage of irregular data
3. The compromise on efficiency is to leverage compact memory footprint
4. In the process of linearising, much of the ‘beauty’ of underlying the mathematics gives way to crude approximation

Bridging the Nature — Computation dichotomy

Artificial Neural Networks

TensorFlow demo



Properties of ANN

1. Weights are associated with nodes and inter-nodal connections
2. Processes are relatively straight forward - matrix-vector products and local reductions
3. Its kernel is essentially plastic - number of hidden layers and number of nodes on each layer govern its form
4. It has somewhat of the ‘it should just work’ essence - a result will always be produced

Where the analogy breaks down

1. *These* nodes and these connections are trained for *this* problem
2. Training is complicated - requires derivative of every output with respect to every weight
3. Improvement of the weights requires vast resources relative to evaluating the actual problem

In brief

1. Somehow, the need for feedback (or at least how it is presently done) appears to lack a natural analogy
2. For protein folding, it is as though all possible solutions (and so the right one) are 'known' at once
3. This is somewhat characteristic of quantum computing, and moreover, its solution is *probably* correct

Incidentally...

Transpose is like dependency tracing

1. Given a sequence of operations: $X(D)$, $Q(X)$, $L(Q)$, the Jacobian of Q w.r.t. X can be written as

$$J_Q = \begin{bmatrix} \frac{\partial Q_1}{\partial X_1} & \cdots & \frac{\partial Q_1}{\partial X_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial Q_m}{\partial X_1} & \cdots & \frac{\partial Q_m}{\partial X_n} \end{bmatrix} = \frac{\mathrm{d}Q}{\mathrm{d}X}$$

2. The derivative of the whole system is the chain of the Jacobians of each operation, giving

$$\frac{\mathrm{d}L}{\mathrm{d}D} = \frac{\mathrm{d}L}{\mathrm{d}Q}\frac{\mathrm{d}Q}{\mathrm{d}X}\frac{\mathrm{d}X}{\mathrm{d}D}$$

Transpose is like dependency tracing

1. The derivative of the system could be built up by inputting *tangents* of J_X

$$\frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}D_\mathrm{i}} = \frac{\mathrm{d}\boldsymbol{L}}{\mathrm{d}\boldsymbol{Q}}\frac{\mathrm{d}\boldsymbol{Q}}{\mathrm{d}\boldsymbol{X}}\frac{\mathrm{d}\boldsymbol{X}}{\mathrm{d}D_\mathrm{i}}$$

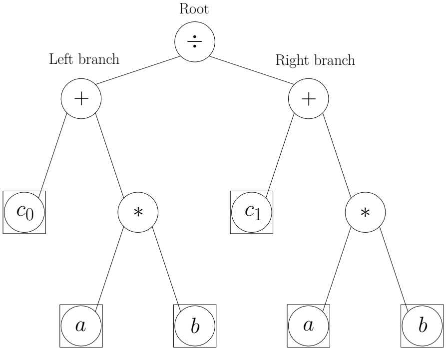
2. But by turning the tangents approach “inside-out”, taking the transpose of the derivative of the system for a particular *gradient* of J_L

$$\frac{\mathrm{d}L_\mathrm{j}}{\mathrm{d}\boldsymbol{D}}^T = \frac{\mathrm{d}\boldsymbol{X}}{\mathrm{d}\boldsymbol{D}}^T\frac{\mathrm{d}\boldsymbol{Q}}{\mathrm{d}\boldsymbol{X}}^T\frac{\mathrm{d}L_\mathrm{j}}{\mathrm{d}\boldsymbol{Q}}^T$$

offers an efficient way of relating an output to all of its inputs.

Give the output a handle on its history

```
auto eval(A const &a,
          B const &b)
{
  auto c0 = 3;
  auto c1 = 4;

  auto t0 = a * b;
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;
  auto r  = t1 / t2;

  return r;
}
```



Instead of computing values, construct expression trees (graphs!)

'Adjoint' differentiation

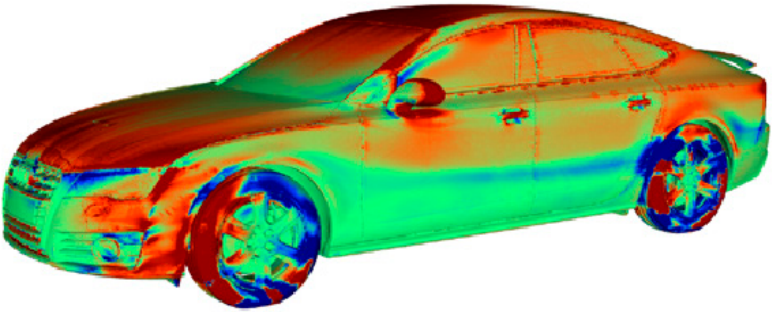
Apply the chain rule and transpose...

```
t0 = a * b      // 1
t1 = c0 + t0    // 2
t2 = c1 + t0    // 3
r  = t1 / t2    // 4
```

```
// 4
t1_d += (1 / t2)     * r_d
t2_d -= (t1 / t2^2) * r_d

// 3
t0_d += t2_d

// 2
t0_d += t1_d

// 1
a_d  += b * t0_d
b_d  += a * t0_d
```

Directed design

Move surface in or out to reduce drag



In brief

1. There is some analogy with entanglement and wave function collapsing in classical computation
2. It gives all solutions in one evaluation
3. But, everything really is back-to-front! (and implementing it is a nightmare!)