Computational Robot Dynamics

Part 4: Forward Dynamics — The Composite Rigid Body Algorithm

Roy Featherstone

ISTITUTO ITALIANO
DI TECNOLOGIA
ADVANCED ROBOTICS

© 2021 Roy Featherstone

The simplest way to calculate the forward dynamics of a kinematic tree is via the joint-space equation of motion:

$$\boldsymbol{H}\ddot{\boldsymbol{q}} + \boldsymbol{C} = \boldsymbol{\tau}$$

where $\boldsymbol{H}$ is the joint-space inertia matrix and $\boldsymbol{C}$ is the joint-space bias force (the force needed to produce zero acceleration).

This is a 3-step process:

1. Calculate $\boldsymbol{C}$
2. Calculate $\boldsymbol{H}$
3. Solve $\boldsymbol{H}\ddot{\boldsymbol{q}} = \boldsymbol{\tau} - \boldsymbol{C}$ for $\ddot{\boldsymbol{q}}$

1. Calculate $\boldsymbol{C}$

   We can already do this using the RNEA:

   if $\quad \boldsymbol{H}\ddot{\boldsymbol{q}} + \boldsymbol{C} = \boldsymbol{\tau} = \mathrm{ID}(model, \boldsymbol{q}, \dot{\boldsymbol{q}}, \ddot{\boldsymbol{q}}, \boldsymbol{f}_{\mathrm{E}})$

   then $\quad \boldsymbol{C} = \mathrm{ID}(model, \boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{0}, \boldsymbol{f}_{\mathrm{E}})$

2. Calculate $\boldsymbol{H}$

   The best method is the *composite-rigid-body algorithm* (CRBA)

3. Solve $\boldsymbol{H}\ddot{\boldsymbol{q}} = \boldsymbol{\tau} - \boldsymbol{C}$ for $\ddot{\boldsymbol{q}}$

   Use a factorization that exploits *branch-induced sparsity*

Composite-Rigid-Body Algorithm

A robot's kinetic energy can be expressed in joint space as

$$T = \tfrac{1}{2}\,\dot{\boldsymbol{q}}^{\mathrm{T}}\boldsymbol{H}\dot{\boldsymbol{q}} = \tfrac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\dot{\boldsymbol{q}}_i^{\mathrm{T}}\boldsymbol{H}_{ij}\dot{\boldsymbol{q}}_j$$

but it is also the sum of the kinetic energies of the bodies:

$$T = \sum_{k=1}^{N}\tfrac{1}{2}\,\boldsymbol{v}_k^{\mathrm{T}}\boldsymbol{I}_k\boldsymbol{v}_k$$

We can obtain a formula for $\boldsymbol{H}$ by equating these two expressions.

Composite-Rigid-Body Algorithm

Substituting $\boldsymbol{v}_k = \sum_{i\in\kappa(k)} \boldsymbol{S}_i \dot{\boldsymbol{q}}_i$ into $T = \sum_{k=1}^{N} \frac{1}{2} \boldsymbol{v}_k^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{v}_k$ gives

$$T = \sum_{k=1}^{N} \tfrac{1}{2} \Big(\sum_{i\in\kappa(k)} \boldsymbol{S}_i \dot{\boldsymbol{q}}_i\Big)^{\mathrm{T}} \boldsymbol{I}_k \Big(\sum_{j\in\kappa(k)} \boldsymbol{S}_j \dot{\boldsymbol{q}}_j\Big)$$

$$= \tfrac{1}{2} \sum_{k=1}^{N} \sum_{i\in\kappa(k)} \sum_{j\in\kappa(k)} \dot{\boldsymbol{q}}_i^{\mathrm{T}} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j \dot{\boldsymbol{q}}_j$$

$$= \tfrac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k\in\nu(i)\cap\nu(j)} \dot{\boldsymbol{q}}_i^{\mathrm{T}} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j \dot{\boldsymbol{q}}_j$$

On comparing this with the expression for T in terms of $\boldsymbol{H}$ we can see that

$$\boldsymbol{H}_{ij} = \sum_{k\in\nu(i)\cap\nu(j)} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j$$

Composite-Rigid-Body Algorithm

Substituting $\boldsymbol{v}_k = \sum_{i\in\kappa(k)} \boldsymbol{S}_i \dot{\boldsymbol{q}}_i$ into $T = \sum_{k=1}^{N} \frac{1}{2} \boldsymbol{v}_k^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{v}_k$ gives

The sum over all triples i,j,k such that both joint i and joint j support body k

$$T \quad \cdots \quad \sum_{j\in\kappa(k)} \boldsymbol{S}_j \dot{\boldsymbol{q}}_j)$$

$$= \frac{1}{2} \sum_{k=1}^{N} \sum_{i\in\kappa(k)} \sum_{j\in\kappa(k)} \dot{\boldsymbol{q}}_i^{\mathrm{T}} \boldsymbol{S}_i^{\mathrm{T}} \cdots$$

The sum over all triples i,j,k such that body k is supported by both joint i and joint j

$$= \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k\in\nu(i)\cap\nu(j)} \dot{\boldsymbol{q}}_i^{\mathrm{T}} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j \dot{\boldsymbol{q}}_j$$

On comparing this with the expression for T in terms of $\boldsymbol{H}$ we can see that

$$\boldsymbol{H}_{ij} = \sum_{k\in\nu(i)\cap\nu(j)} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j$$

Composite-Rigid-Body Algorithm

Substituting $\boldsymbol{v}_k = \sum_{i\in\kappa(k)} \boldsymbol{S}_i \dot{\boldsymbol{q}}_i$ into $T = \sum_{k=1}^{N} \frac{1}{2} \boldsymbol{v}_k^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{v}_k$ gives

$$T = \sum_{k=1}^{N} \tfrac{1}{2} \Big(\sum_{i\in\kappa(k)} \boldsymbol{S}_i \dot{\boldsymbol{q}}_i \Big)^{\mathrm{T}} \boldsymbol{I}_k \Big(\sum_{j\in\kappa(k)} \boldsymbol{S}_j \dot{\boldsymbol{q}}_j \Big)$$

$$= \tfrac{1}{2} \sum_{k=1}^{N} \sum_{i\in\kappa(k)} \sum_{j\in\kappa(k)} \dot{\boldsymbol{q}}_i^{\mathrm{T}} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j \dot{\boldsymbol{q}}_j$$

$$= \tfrac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k\in\nu(i)\cap\nu(j)} \dot{\boldsymbol{q}}_i^{\mathrm{T}} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j \dot{\boldsymbol{q}}_j$$

On comparing this with the expression for T in terms of $\boldsymbol{H}$ we can see that

$$\boldsymbol{H}_{ij} = \sum_{k\in\nu(i)\cap\nu(j)} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j$$

$$\boxed{T = \tfrac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \dot{\boldsymbol{q}}_i^{\mathrm{T}} \boldsymbol{H}_{ij} \dot{\boldsymbol{q}}_j}$$

Composite-Rigid-Body Algorithm

From the definition of the sets $\nu(i)$ it follows that

$$\nu(i) \cap \nu(j) = \begin{cases} \nu(i) & \text{if } i \in \nu(j) \\ \nu(j) & \text{if } j \in \nu(i) \\ \emptyset & \text{otherwise} \end{cases}$$

So $$\boldsymbol{H}_{ij} = \sum_{k \in \nu(i) \cap \nu(j)} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_k \boldsymbol{S}_j \quad \Rightarrow \quad \boldsymbol{H}_{ij} = \begin{cases} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_i^{\mathrm{c}} \boldsymbol{S}_j & \text{if } i \in \nu(j) \\ \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_j^{\mathrm{c}} \boldsymbol{S}_j & \text{if } j \in \nu(i) \\ \boldsymbol{0} & \text{otherwise} \end{cases}$$

where $\boldsymbol{I}_i^{\mathrm{c}} = \sum_{j \in \nu(i)} \boldsymbol{I}_j$ is the inertia of a composite rigid body consisting of all of the bodies in the subtree $\nu(i)$. It can be computed recursively using

$$\boldsymbol{I}_i^{\mathrm{c}} = \boldsymbol{I}_i + \sum_{j \in \mu(i)} \boldsymbol{I}_j^{\mathrm{c}}$$

Basic Algorithm

$$\boldsymbol{I}_i^{\mathrm{c}} = \boldsymbol{I}_i + \sum_{j \in \mu(i)} \boldsymbol{I}_j^{\mathrm{c}}$$

$$\boldsymbol{H}_{ij} = \begin{cases} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_i^{\mathrm{c}} \boldsymbol{S}_j & \text{if } i \in \nu(j) \\ \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_j^{\mathrm{c}} \boldsymbol{S}_j & \text{if } j \in \nu(i) \\ \boldsymbol{0} & \text{otherwise} \end{cases}$$

Algorithm in Body Coordinates

$$\boldsymbol{I}_i^{\mathrm{c}} = \boldsymbol{I}_i + \sum_{j \in \mu(i)} {}^i\boldsymbol{X}_j^* \, \boldsymbol{I}_j^{\mathrm{c}} \, {}^j\boldsymbol{X}_i$$

$${}^{\lambda(j)}\boldsymbol{F}_i = {}^{\lambda(j)}\boldsymbol{X}_j^* \, {}^j\boldsymbol{F}_i \qquad ({}^i\boldsymbol{F}_i = \boldsymbol{I}_i^{\mathrm{c}} \boldsymbol{S}_i)$$

$$\boldsymbol{H}_{ij} = \begin{cases} {}^j\boldsymbol{F}_i^{\mathrm{T}} \boldsymbol{S}_j & \text{if } i \in \nu(j) \\ \boldsymbol{H}_{ji}^{\mathrm{T}} & \text{if } j \in \nu(i) \\ \boldsymbol{0} & \text{otherwise} \end{cases}$$

Pseudocode

$\boldsymbol{H} = \boldsymbol{0}$
for $i = 1$ **to** N **do**
  $\boldsymbol{I}_i^{\mathrm{c}} = \boldsymbol{I}_i$
end
for $i = N$ **to** 1 **do**
  if $\lambda(i) \neq 0$ **then**
    $\boldsymbol{I}_{\lambda(i)}^{\mathrm{c}} = \boldsymbol{I}_{\lambda(i)}^{\mathrm{c}} + {}^{\lambda(i)}\boldsymbol{X}_i^* \boldsymbol{I}_i^{\mathrm{c}}\, {}^{i}\boldsymbol{X}_{\lambda(i)}$
  end
  $\boldsymbol{F} = \boldsymbol{I}_i^{\mathrm{c}} \boldsymbol{S}_i$
  $\boldsymbol{H}_{ii} = \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{F}$
  $j = i$
  while $\lambda(j) \neq 0$ **do**
    $\boldsymbol{F} = {}^{\lambda(j)}\boldsymbol{X}_j^* \boldsymbol{F}$
    $j = \lambda(j)$
    $\boldsymbol{H}_{ij} = \boldsymbol{F}^{\mathrm{T}} \boldsymbol{S}_j$
    $\boldsymbol{H}_{ji} = \boldsymbol{H}_{ij}^{\mathrm{T}}$
  end
end

Algorithm in Body Coordinates

$$\boldsymbol{I}_i^{\mathrm{c}} = \boldsymbol{I}_i + \sum_{j \in \mu(i)} {}^{i}\boldsymbol{X}_j^* \boldsymbol{I}_j^{\mathrm{c}}\, {}^{j}\boldsymbol{X}_i$$

$${}^{\lambda(j)}\boldsymbol{F}_i = {}^{\lambda(j)}\boldsymbol{X}_j^*\, {}^{j}\boldsymbol{F}_i \qquad ({}^{i}\boldsymbol{F}_i = \boldsymbol{I}_i^{\mathrm{c}} \boldsymbol{S}_i)$$

$$\boldsymbol{H}_{ij} = \begin{cases} {}^{j}\boldsymbol{F}_i^{\mathrm{T}} \boldsymbol{S}_j & \text{if } i \in \nu(j) \\ \boldsymbol{H}_{ji}^{\mathrm{T}} & \text{if } j \in \nu(i) \\ \boldsymbol{0} & \text{otherwise} \end{cases}$$

Matlab Code

```
function  [H,C] = HandC( model, q, qd, f_ext )
 :
IC = model.I;

for i = model.NB:-1:1
  if model.parent(i) ~= 0
    IC{model.parent(i)} = IC{model.parent(i)} + Xup{i}'*IC{i}*Xup{i};
  end
end

H = zeros(model.NB);

for i = 1:model.NB
  fh = IC{i} * S{i};
  H(i,i) = S{i}' * fh;
  j = i;
  while model.parent(j) > 0
    fh = Xup{j}' * fh;
    j = model.parent(j);
    H(i,j) = S{j}' * fh;
    H(j,i) = H(i,j);
  end
end
```

$${}^{i}\boldsymbol{X}_{\lambda(i)}^{\mathrm{T}} = {}^{\lambda(i)}\boldsymbol{X}_{i}^{*}$$

main loop broken into two, and loop order reversed

you can see here a few deviations from the pseudocode

missing transpose: scalars (1x1 matrices) are symmetric

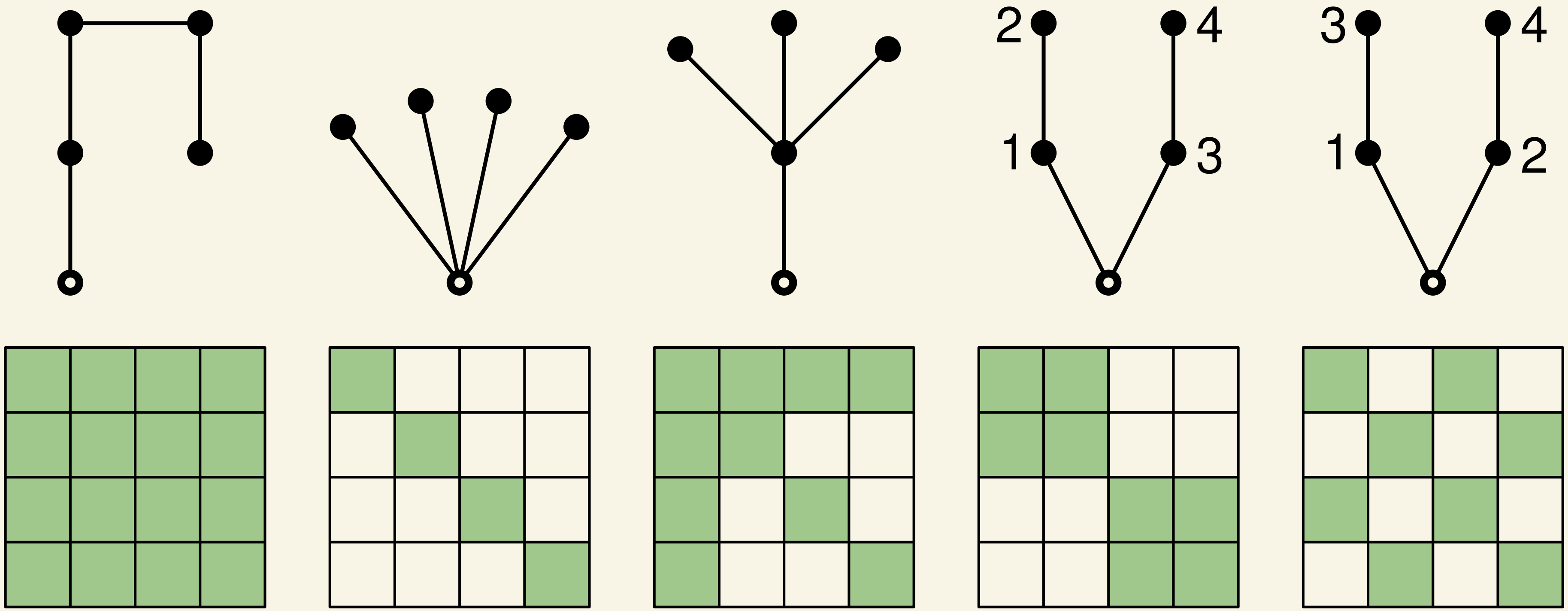
Branch-Induced Sparsity

$$\boldsymbol{H}_{ij} = \begin{cases} \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_i^{\mathrm{c}} \boldsymbol{S}_j & \text{if } i \in \nu(j) \\ \boldsymbol{S}_i^{\mathrm{T}} \boldsymbol{I}_j^{\mathrm{c}} \boldsymbol{S}_j & \text{if } j \in \nu(i) \\ \boldsymbol{0} & \text{otherwise} \end{cases}$$

This zero means that $\boldsymbol{H}_{ij}$ will always be zero whenever i and j are on separate branches. So the presence of branches in the tree causes a pattern of permanently zero-valued elements in $\boldsymbol{H}$. We call this pattern *branch-induced sparsity*.

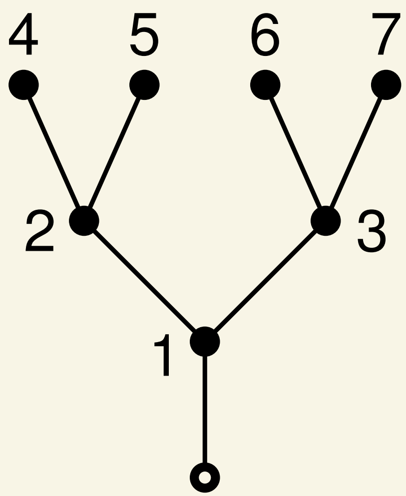
We can exploit this sparsity to reduce the cost and complexity of solving the equation of motion.

Sparsity Patterns

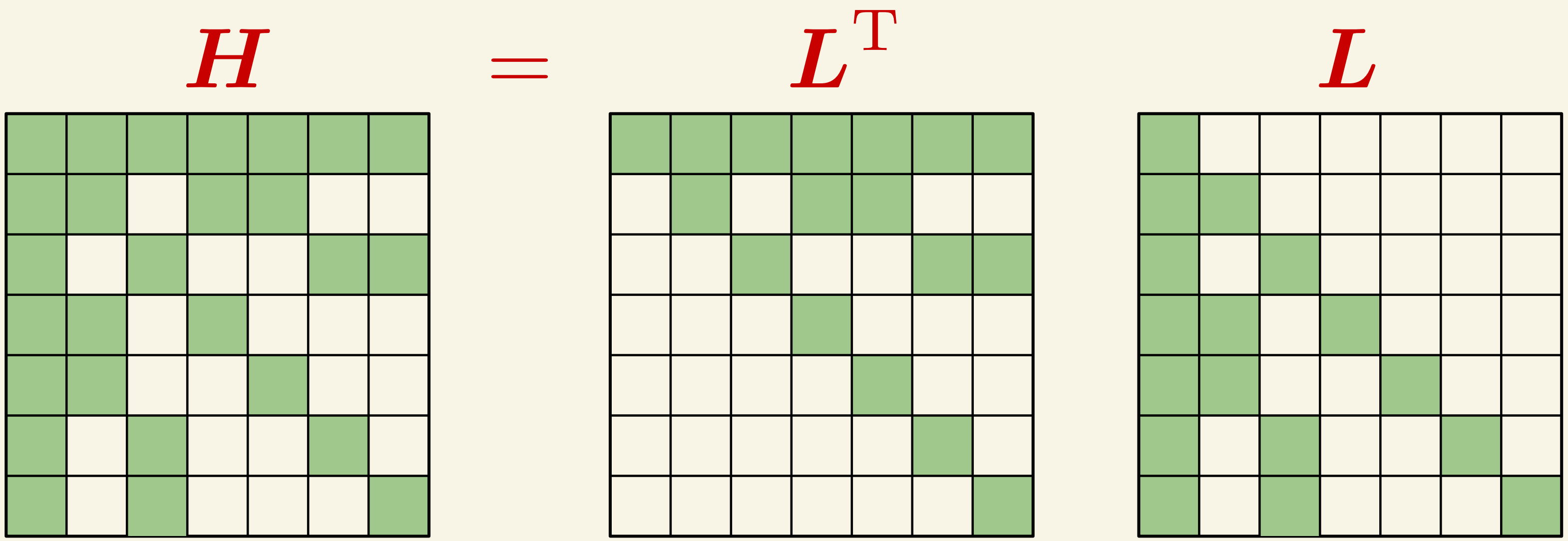


= nonzero submatrix or element

Sparse Factorization



If we factorize $\boldsymbol{H}$ into $\boldsymbol{L}^{\mathrm{T}}\boldsymbol{L}$ instead of the usual $\boldsymbol{L}\boldsymbol{L}^{\mathrm{T}}$ then the sparsity pattern is preserved in the factors.



Sparse Factorization

```
function LTDL(H,λ)
for k = n to 1 do
    i = λ(k)
    while i ≠ 0 do
        a = H_ki / H_kk
        j = i
        while j ≠ 0 do
            H_ij = H_ij - H_kj a
            j = λ(j)
        end
        H_ki = a
        i = λ(i)
    end
end
```

factorize $\boldsymbol{H}$ *in situ*

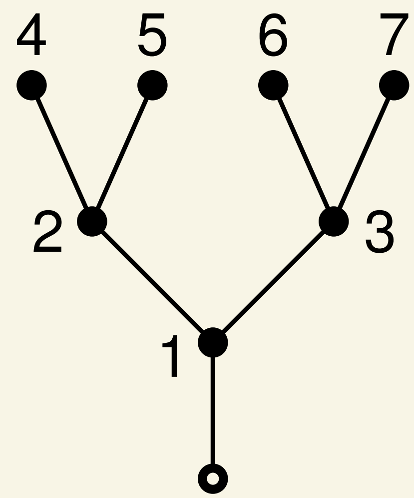
outer loop runs backwards

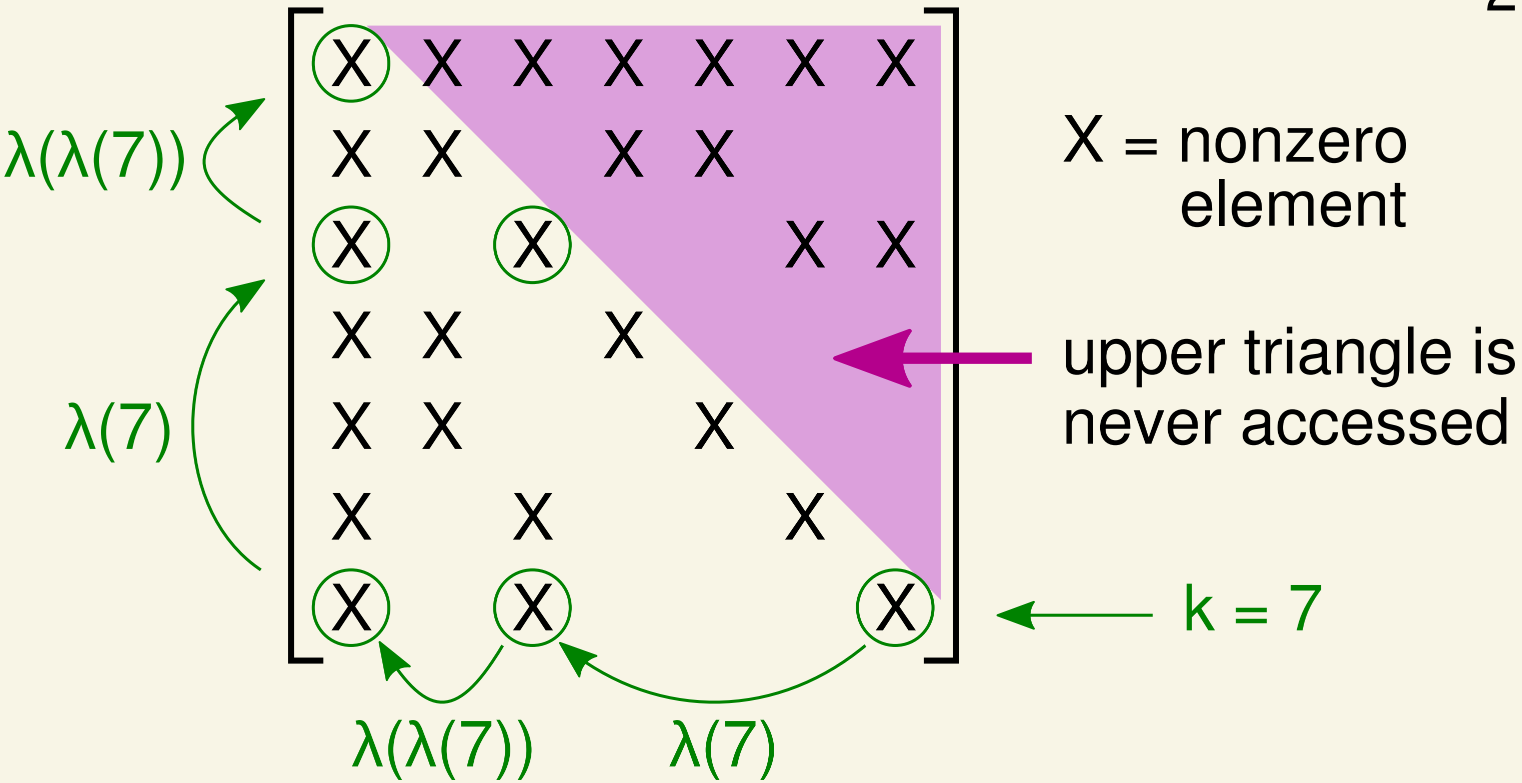
(this is what makes the function factorize $\boldsymbol{H}$ into $\boldsymbol{L}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{L}$ instead of $\boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^{\mathrm{T}}$)

inner loops visit only the ancestors of k

(this is what gives the algorithm its efficiency)

Sparse Factorization





By iterating only over the ancestors of k, the algorithm performs the least possible amount of work; for example, by updating only 6 elements when k=7 instead of 28 elements.

Exploiting Sparsity

How big are the benefits of exploiting branch-induced sparsity?

- reduction in computational complexity from $O(n^3)$ to $O(nd^2)$, where n is the number of variables and d is the depth of the tree.

- for a typical humanoid or quadruped, the cost of solving the equation of motion (i.e., solving $\boldsymbol{H}\ddot{\boldsymbol{q}} + \boldsymbol{C} = \boldsymbol{\tau}$ for $\ddot{\boldsymbol{q}}$) is reduced by approximately a factor of 4.

Note: the composite-rigid-body algorithm automatically exploits branch-induced sparsity by calculating only the nonzero elements in $\boldsymbol{H}$. Its complexity is $O(nd)$.