Time Integration in Chrono

Numerical integration of smooth and non-smooth Dynamics

Time Integration in Chrono

- Two classes of time stepping methods in Chrono
  - Time steppers for smooth dynamics
    - Classical multibody dynamics – rigid and flexible connected through joints
    - FEA
    - Fluid solid interaction problems
  - Time steppers for non-smooth dynamics
    - Scenarios w/ friction and contact

Time Integration – Smooth Dynamics

- Smooth dynamics:
  - Equations of Motion: formulated as Differential Algebraic Equations (DAE)
  - Time-stepping methods:
    - HHT
    - Euler implicit
    - Euler semi-implicit linearized
    - Newmark
  - Require solution of a linear system at each time step
    - MINRES
    - MKL
    - MUMPS
  - Discontinuous forces if any, are regularized via penalty
    - Can still have friction and contact, but is “smoothed”

Time Integration – Non-smooth Dynamics

- Non-smooth dynamics:
  - Equations of motion formulated using complementarity conditions
  - Time-stepping method:
    - Half-implicit symplectic Euler
  - Cone Complementarity Problem (CCP) solved at each time step
    - SOR
    - APGD
    - Barzilai-Borwein
  - Discontinuous forces: no need to be “smoothed”
  - No support for FEA yet

Smooth dynamics - DAE

The HHT Time Stepper

Linear Solvers

Differential Problems

- An Ordinary Differential Equation (**ODE**):

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, t)$$

- A Differential Algebraic Equation (**DAE**)
  - In implicit form:

$$\frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}, t)$$
$$\boldsymbol{g}(\boldsymbol{x}, t) = \boldsymbol{0}$$

  - Introduces constraints ***g***

The Chrono case

$$M\frac{d\boldsymbol{v}}{dt} = \boldsymbol{f}(\boldsymbol{q}, \boldsymbol{v}, t) + D_{\mathcal{B}}\widehat{\gamma}_{\mathcal{B}}(t)$$
$$C(\boldsymbol{q}, t) = \boldsymbol{0}$$

DAE Explicit Integrators

- **Explicit** integrators:

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{F}(\boldsymbol{x}(t))$$

  - Straightforward to implement - they do not require solving linear systems
  - Require very small time steps, due to stability reasons
  - The stiffer the problem, the smaller the time step
  - Lead to numerical drift when handling DAEs
  - Used by traditional DEM granular dynamics simulators

DAE Implicit Integrators

- **Implicit** integrators:

$$\boldsymbol{G}\left(\boldsymbol{x}(t+\Delta t), \boldsymbol{x}(t)\right) = \boldsymbol{0}$$

  - Can use large time steps
  - More complex: they find $\boldsymbol{x}(t+\Delta t)$ by solving a nonlinear system $\boldsymbol{G} = \boldsymbol{0}$ with Newton-Raphson
    - Jacobians matrices of $\boldsymbol{G}$ are needed (ex. stiffness matrices, etc.)
    - Require solution of one or more linear systems at each time step
  - Useful both for ODEs and DAEs – for the latter, they enforce the kinematic constraints well
  - Used in FEA problems, handle stiffness well

DAE Implicit Integrators in Chrono

- Classical Euler implicit
  - First order accurate, large numerical damping

- Euler semi-implicit linearized (1 step)
  - First order accurate, large numerical damping
  - Same time-stepping used for DVI non-smooth dynamics, it can use complementarity solvers

- Trapezoidal
  - Second order accurate, no numerical damping
  - Doesn’t work well with joints (kinematic constraints)

- Newmark (index 3 DAE)
  - Adjustable numerical damping, first order (except in particular case)

- HHT (index 3 DAE)
  - Second order accurate, adjustable numerical damping
  - Most used integrator for FEA problems in Chrono

Discretization of the Constrained EOM (1/3)

- The discretized equations solved at each time t_{n+1} are:

$$\begin{cases} \mathbf{M}\ddot{\mathbf{q}}_{n+1} + \mathbf{\Phi}_{\mathbf{q}}^T(\mathbf{q}_{n+1})\lambda_{n+1} - \mathbf{Q}^A(\dot{\mathbf{q}}_{n+1}, \mathbf{q}_{n+1}, t_{n+1}) = \mathbf{0} \\ \frac{1}{\beta h^2}\mathbf{\Phi}(\mathbf{q}_{n+1}, t_{n+1}) = \mathbf{0} \end{cases}$$

- Generalized positions $\mathbf{q}_{n+1}$ and velocities $\dot{\mathbf{q}}_{n+1}$ above expressions are functions of the accelerations $\ddot{\mathbf{q}}_{n+1}$:

$$\mathbf{q}_{n+1} = \mathbf{q}_n + h\dot{\mathbf{q}}_n + \frac{h^2}{2}\left[(1-2\beta)\ddot{\mathbf{q}}_n + 2\beta\ddot{\mathbf{q}}_{n+1}\right]$$

$$\dot{\mathbf{q}}_{n+1} = \dot{\mathbf{q}}_n + h\left[(1-\gamma)\ddot{\mathbf{q}}_n + \gamma\ddot{\mathbf{q}}_{n+1}\right]$$

These are Newmark's formulas that express the generalized positions and velocities as functions of the generalized accelerations

Discretization of the Constrained EOM (2/3)

- The unknowns are the accelerations and the Lagrange multipliers
  - The number of unknowns is equal to the number of equations

- The equations that must be solved now are algebraic and nonlinear
  - Differential problem has been transformed into an algebraic one
  - The new problem: find acceleration and Lagrange multipliers that satisfy

$$\begin{bmatrix} \mathbf{M}\ddot{\mathbf{q}}_{n+1} + \mathbf{\Phi}_{\mathbf{q}}^{T}(\mathbf{q}_{n+1})\lambda_{n+1} - \mathbf{Q}^{A}(\dot{\mathbf{q}}_{n+1}, \mathbf{q}_{n+1}, t_{n+1}) \\ \frac{1}{\beta h^2}\mathbf{\Phi}(\mathbf{q}_{n+1}, t_{n+1}) \end{bmatrix} = \mathbf{0}$$

- We have to use Newton's method
  - We need the Jacobian of the nonlinear system of equations (chain rule will be used to simplify calculations)
  - This looks exactly like what we had to do when for Kinematics analysis of a mechanism (there we solved $\Phi(\mathbf{q},t)=0$ to get the positions $\mathbf{q}$)

Discretization of the Constrained EOM (3/3)

- Define the following two functions:

$$\bar{\mathbf{\Psi}}(\ddot{\mathbf{q}}_{n+1}, \dot{\mathbf{q}}_{n+1}, \mathbf{q}_{n+1}, \lambda_{n+1}) \triangleq \mathbf{M}\ddot{\mathbf{q}}_{n+1} + \mathbf{\Phi}_{\mathbf{q}}^{T}(\mathbf{q}_{n+1})\lambda_{n+1} - \mathbf{Q}^{A}(\dot{\mathbf{q}}_{n+1}, \mathbf{q}_{n+1}, t_{n+1})$$

$$\bar{\mathbf{\Omega}}(\mathbf{q}_{n+1}) \triangleq \frac{1}{\beta h^2}\mathbf{\Phi}(\mathbf{q}_{n+1}, t_{n+1})$$

- Once we use the Newmark discretization formulas, these functions depend in fact only on the accelerations $\ddot{\mathbf{q}}_{n+1}$ and Lagrange multipliers $\boldsymbol{\lambda}_{n+1}$

- To make this clear, define the new functions:

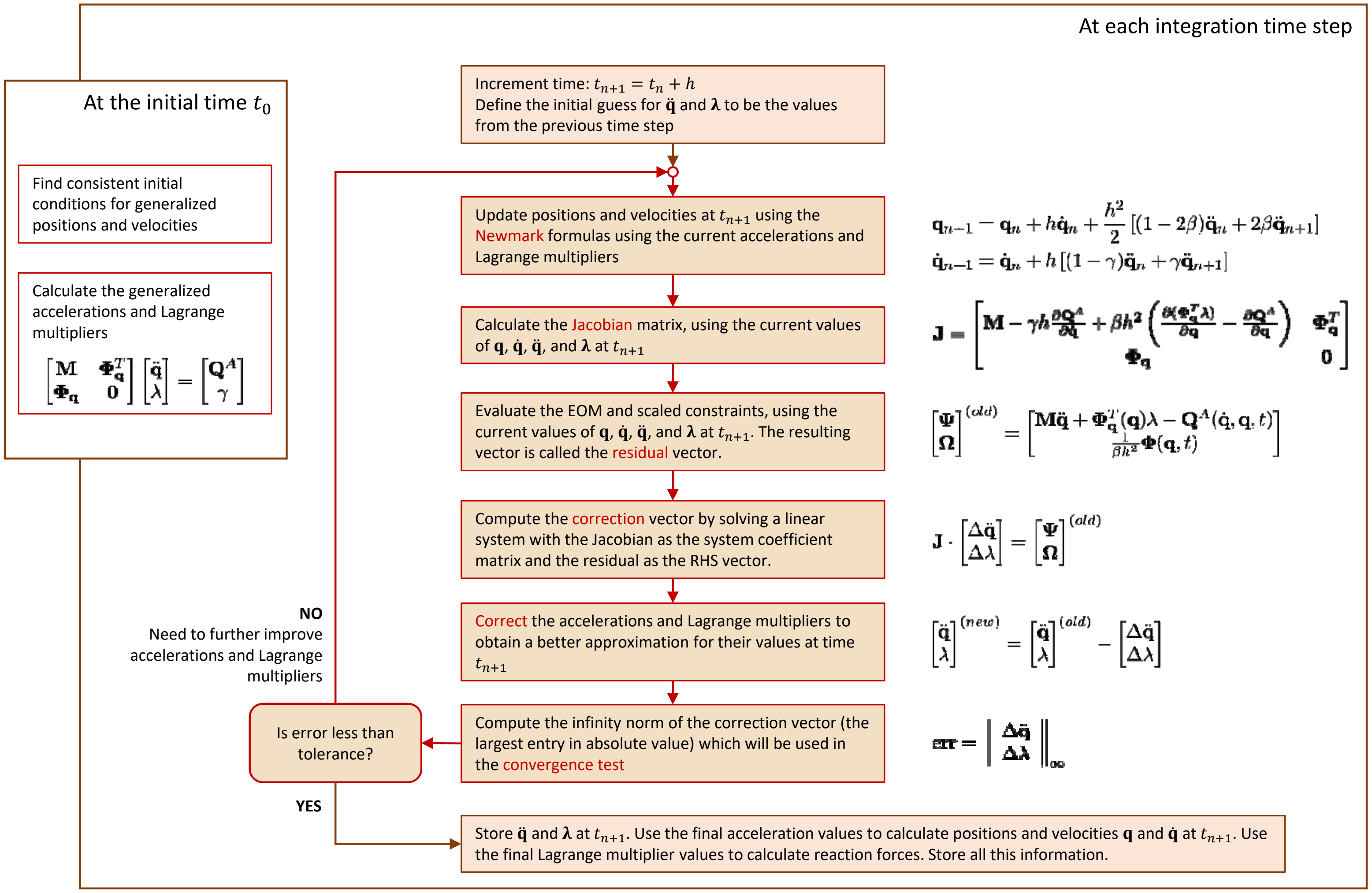
$$\mathbf{\Psi}(\ddot{\mathbf{q}}_{n+1}, \boldsymbol{\lambda}_{n+1}) \equiv \bar{\mathbf{\Psi}}(\ddot{\mathbf{q}}_{n+1}, \dot{\mathbf{q}}_{n+1}(\ddot{\mathbf{q}}_{n+1}), \mathbf{q}_{n+1}(\ddot{\mathbf{q}}_{n+1}), \boldsymbol{\lambda}_{n+1})$$

$$\mathbf{\Omega}(\ddot{\mathbf{q}}_{n+1}) \equiv \bar{\mathbf{\Omega}}(\mathbf{q}_{n+1}(\ddot{\mathbf{q}}_{n+1}))$$

- Therefore, we must solve for $\ddot{\mathbf{q}}_{n+1}$ and $\boldsymbol{\lambda}_{n+1}$ the following system

$$\begin{bmatrix} \mathbf{\Psi}(\ddot{\mathbf{q}}_{n+1}, \boldsymbol{\lambda}_{n+1}) \\ \mathbf{\Omega}(\ddot{\mathbf{q}}_{n+1}) \end{bmatrix} = \mathbf{0}$$

Time Stepping,



At each integration time step

At the initial time t_0

Find consistent initial conditions for generalized positions and velocities

Calculate the generalized accelerations and Lagrange multipliers

$$\begin{bmatrix} \mathbf{M} & \mathbf{\Phi}_{\mathbf{q}}^T \\ \mathbf{\Phi}_{\mathbf{q}} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \ddot{\mathbf{q}} \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{Q}^A \\ \gamma \end{bmatrix}$$

Increment time: $t_{n+1} = t_n + h$
Define the initial guess for $\ddot{\mathbf{q}}$ and $\boldsymbol{\lambda}$ to be the values from the previous time step

Update positions and velocities at t_{n+1} using the Newmark formulas using the current accelerations and Lagrange multipliers

$$\mathbf{q}_{n-1} - \mathbf{q}_n + h\dot{\mathbf{q}}_n + \frac{h^2}{2}\left[(1-2\beta)\ddot{\mathbf{q}}_n + 2\beta\ddot{\mathbf{q}}_{n+1}\right]$$
$$\dot{\mathbf{q}}_{n-1} = \dot{\mathbf{q}}_n + h\left[(1-\gamma)\ddot{\mathbf{q}}_n + \gamma\ddot{\mathbf{q}}_{n+1}\right]$$

Calculate the Jacobian matrix, using the current values of $\mathbf{q}$, $\dot{\mathbf{q}}$, $\ddot{\mathbf{q}}$, and $\boldsymbol{\lambda}$ at t_{n+1}

$$\mathbf{J} = \begin{bmatrix} \mathbf{M} - \gamma h \frac{\partial \mathbf{Q}^A}{\partial \dot{\mathbf{q}}} + \beta h^2 \left(\frac{\partial(\mathbf{\Phi}_{\mathbf{q}}^T \lambda)}{\partial \mathbf{q}} - \frac{\partial \mathbf{Q}^A}{\partial \mathbf{q}} \right) & \mathbf{\Phi}_{\mathbf{q}}^T \\ \mathbf{\Phi}_{\mathbf{q}} & \mathbf{0} \end{bmatrix}$$

Evaluate the EOM and scaled constraints, using the current values of $\mathbf{q}$, $\dot{\mathbf{q}}$, $\ddot{\mathbf{q}}$, and $\boldsymbol{\lambda}$ at t_{n+1}. The resulting vector is called the residual vector.

$$\begin{bmatrix} \mathbf{\Psi} \\ \mathbf{\Omega} \end{bmatrix}^{(old)} = \begin{bmatrix} \mathbf{M}\ddot{\mathbf{q}} + \mathbf{\Phi}_{\mathbf{q}}^T(\mathbf{q})\lambda - \mathbf{Q}^A(\dot{\mathbf{q}}, \mathbf{q}, t) \\ \frac{1}{\beta h^2}\mathbf{\Phi}(\mathbf{q}, t) \end{bmatrix}$$

Compute the correction vector by solving a linear system with the Jacobian as the system coefficient matrix and the residual as the RHS vector.

$$\mathbf{J} \cdot \begin{bmatrix} \Delta\ddot{\mathbf{q}} \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} \mathbf{\Psi} \\ \mathbf{\Omega} \end{bmatrix}^{(old)}$$

Correct the accelerations and Lagrange multipliers to obtain a better approximation for their values at time t_{n+1}

$$\begin{bmatrix} \ddot{\mathbf{q}} \\ \lambda \end{bmatrix}^{(new)} = \begin{bmatrix} \ddot{\mathbf{q}} \\ \lambda \end{bmatrix}^{(old)} - \begin{bmatrix} \Delta\ddot{\mathbf{q}} \\ \Delta\lambda \end{bmatrix}$$

Compute the infinity norm of the correction vector (the largest entry in absolute value) which will be used in the convergence test

$$\mathbf{err} = \left\| \begin{matrix} \Delta\ddot{\mathbf{q}} \\ \Delta\boldsymbol{\lambda} \end{matrix} \right\|_{\infty}$$

Is error less than tolerance?

NO
Need to further improve accelerations and Lagrange multipliers

YES

Store $\ddot{\mathbf{q}}$ and $\boldsymbol{\lambda}$ at t_{n+1}. Use the final acceleration values to calculate positions and velocities $\mathbf{q}$ and $\dot{\mathbf{q}}$ at t_{n+1}. Use the final Lagrange multiplier values to calculate reaction forces. Store all this information.

Configuring the Integrator in Chrono

- It can be changed with `SetIntegrationType()`
- Additional parameters via `std::static_pointer_cast<...>(my_system.GetTimestepper())`

```
// change the time integration to Euler, for DVI too
my_system.SetIntegrationType(ChSystem::INT_EULER_IMPLICIT_LINEARIZED);
```

```
// change the time integration to HHT:
my_system.SetIntegrationType(ChSystem::INT_HHT);
auto integrator = std::static_pointer_cast<ChTimestepperHHT>(my_system.GetTimestepper());
integrator->SetAlpha(-0.2);
integrator->SetMaxiters(8);
integrator->SetAbsTolerances(5e-05, 1.8e00);
integrator->SetMode(ChTimestepperHHT::POSITION);
integrator->SetModifiedNewton(false);
integrator->SetScaling(true);
integrator->SetVerbose(true);
```

New Subtopic:
Linear System Solvers

- All DAE solvers require the solution of linear systems

- Linear system solvers are independent from the time integrator
  - One can mix and match

- Available linear system solvers
  - MINRES (iterative solver, free)
  - MKL (direct solver, requires license)
  - MUMPS (direct solver, free)

- Moving forward:
  - MUMPS with OpenBLAS since they are both free and licensed under BSD

Linear System Solvers: MINRES

- Available in the main Chrono unit
- A Krylov-type iterative solver
- Convergence might slow down when large mass or stiffness ratios are used
- Robust in case of redundant constraints
- Warm starting can be used to reuse last solution (faster solution)

```
    // Change solver settings
    my_system.SetSolverType(ChSystem::SOLVER_MINRES);
    my_system.SetSolverWarmStarting(true);
    my_system.SetMaxItersSolverSpeed(200); // Max number of iterations for main solver
    my_system.SetMaxItersSolverStab(200); // Used only by few time integrators
    my_system.SetTolForce(1e-13);
```

Linear System Solvers: MKL

- MKL Intel libraries must be licensed and installed on your system,
- Available in the optional Chrono::MKL unit (enable it in CMake)
- Direct parallel solver: no iterations are needed
- Not robust in case of redundant constraints

```
#include "chrono_mkl/ChSolverMKL.h"
...
// change the solver to MKL:
ChSolverMKL<>* mkl_solver_stab = new ChSolverMKL<>;
ChSolverMKL<>* mkl_solver_speed = new ChSolverMKL<>;
my_system.ChangeSolverSpeed(mkl_solver_speed);
my_system.ChangeSolverStab(mkl_solver_stab); // Used only by few time integrators
mkl_solver_speed->SetSparsityPatternLock(true);
mkl_solver_stab->SetSparsityPatternLock(true); // Used only by few time integrators
```

Linear System Solvers: MUMPS

- Work in progress to be wrapped up by mid January
- Direct parallel solver
- Developed in France/UK, relies on OpenBLAS, which developed in China
- Free solution, source code available for MUMPS & OpenBLAS

```
#include "chrono_mumps/ChSolverMUMPS.h"
...
// change the solver to MUMPS:
ChSolverMUMPS<>* mumps_solver_stab = new ChSolverMUMPS<>;
ChSolverMUMPS<>* mumps_solver_speed = new ChSolverMUMPS<>;
my_system.ChangeSolverSpeed(mumps_solver_speed);
my_system.ChangeSolverStab(mumps_solver_stab); // Used only by few time integrators
mumps_solver_speed->SetSparsityPatternLock(true);
mumps_solver_stab->SetSparsityPatternLock(true); // Used only by few time integrators
```

Non-Smooth dynamics - DVI

The DVI time-stepper

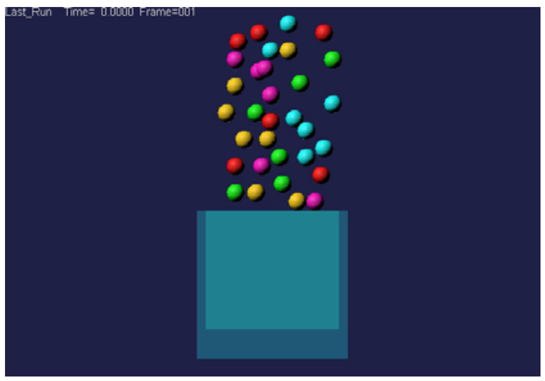
The CCP solvers

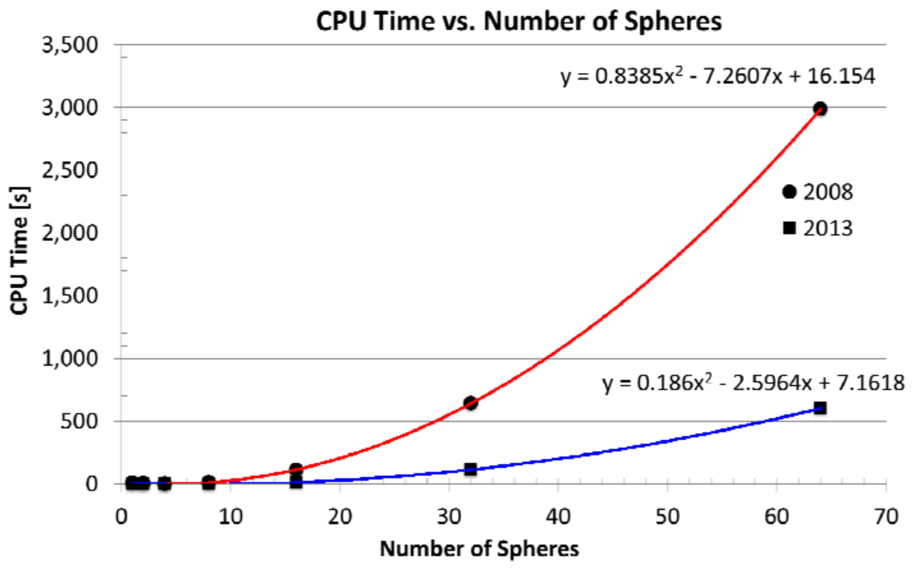
The Cornerstone, DVI Method

- Need to solve fast quadratic optimization problem with conic constraints

$$\gamma^{\star} = \underset{\substack{\gamma_i \in \Upsilon_i \\ 1 \le i \le N_c}}{\operatorname{argmin}} \left(\frac{1}{2}\gamma^T \mathbf{N} \gamma + \mathbf{r}^T \gamma \right)$$

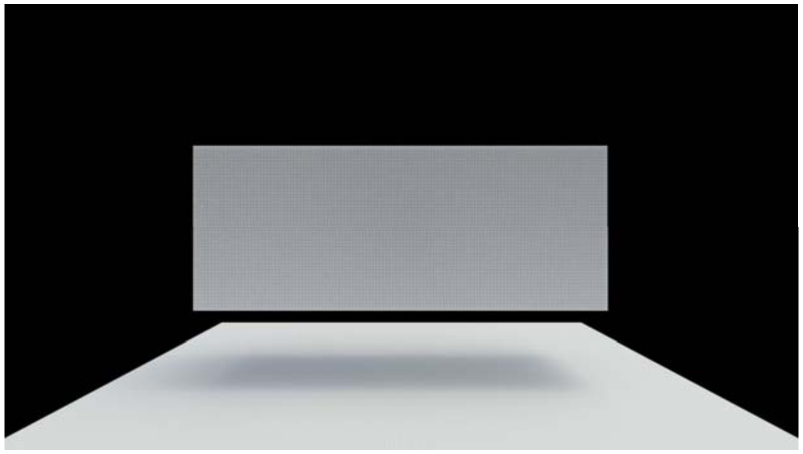
3 Second Dynamics – 1 million spheres dropping in a bucket [Commercial Software Simulation]





[Luning]→

1 Million Bodies, Parallel Simulation on GPU card



- 20 second long simulation
- Two hours to finish simulation
- GPU Card: GTX680
- Optimization problem for γ
  - Approx. 4 million variables
  - Solved at each time step
  - Problem looks like

$$\gamma^* = \underset{\substack{\gamma_i \in \Upsilon_i \\ 1 \le i \le N_c}}{\operatorname{argmin}} \left(\frac{1}{2} \gamma^T \mathbf{N} \gamma + \mathbf{r}^T \gamma \right)$$

"Leveraging Parallel Computing in Multibody Dynamics," D. Negrut, A. Tasora, H. Mazhar, T. Heyn, P. Hahn, Multibody System Dynamics, vol. 27, pp. 95-117, 2012.

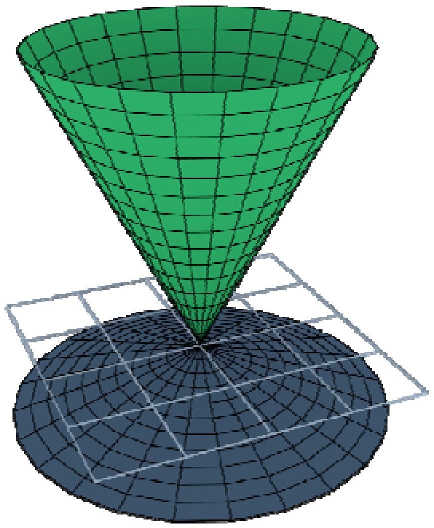
CCP Solvers in Chrono

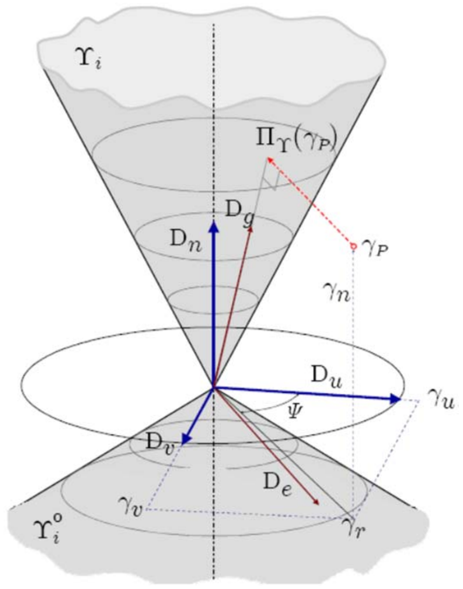
- Fixed-point solvers:
  - Projected-SOR (Jacobi) ⟵
  - Projected-GaussSeidel
  - Projected-Symmetric-SOR

- Krylov spectral methods
  - Barzilai-Borwein
  - Nesterov Accelerated Projected Gradient Descent (APGD)

The Friction Cone and Projection Upon Friction Cone

- Projection upon friction cone





Projected Jacobi (Projected-SOR)

```
ALGORITHM JACOBI($\mathbf{N}$, $\mathbf{r}$, $\tau$, $N_{max}$, $\gamma_0$)
(1)    for $k := 0$ to $N_{max}$
(2)        $\hat{\gamma}_{(k+1)} = \Pi_{\mathcal{K}}\left(\gamma_{(k)} - \omega \mathbf{B}\left(\mathbf{N}\gamma_{(k)} + \mathbf{r}\right)\right)$
(3)        $\gamma_{(k+1)} = \lambda \hat{\gamma}_{(k+1)} + (1-\lambda)\gamma_{(k)}$
(4)        $r = r\left(\gamma_{(k+1)}\right)$
(5)        if $r < \tau$
(6)            break
(7)    endfor
(8)    return Value at time step $t^{(l+1)}$, $\gamma^{(l+1)} := \gamma_{(k+1)}$ .
```

Projected Gauss-Seidel

```
ALGORITHM GAUSS-SEIDEL($\mathbf{N}$, $\mathbf{r}$, $\tau$, $N_{max}$, $\gamma_0$)
(1)     for $k := 0$ to $N_{max}$
(2)         for $i = 1$ to $n_c$
(3)             $\hat{\gamma}_{i,(k+1)} = \Pi_{\mathcal{K}}\left(\gamma_{i,(k)} - \omega \mathbf{B}_i \left(\mathbf{N}\gamma_k + \mathbf{r}\right)_i\right)$
(4)             $\gamma_{i,(k+1)} = \lambda \hat{\gamma}_{i,(k+1)} + (1-\lambda)\,\gamma_{i,(k)}$
(5)         endfor
(6)         $r = r\left(\gamma_{k+1}\right)$
(7)         if $r < \tau$
(8)             break
(9)     endfor
(10)    return Value at time step $t^{(l+1)}$, $\gamma^{(l+1)} := \gamma_{(k+1)}$ .
```

P-SOR solver for CCP

- Use `SetSolverType()` to change the solver:

```
// change the solver to P-SOR:
my_system.SetSolverType(ChSystem::SOLVER_SOR);

// use high iteration number if contacts interpenetrate:
my_system.SetMaxItersSolverSpeed(90);
```

Nesterov’s Accelerated Projected Gradient Descent

```
ALGORITHM NAPG($\boldsymbol{N}, \boldsymbol{r}, t \leq \frac{1}{\lambda_{max}(\boldsymbol{N})}, \tau, N_{max}$)
(1)   $\gamma_0 = \boldsymbol{0}_{n_c}$
(2)   $\hat{\gamma}_0 = \boldsymbol{1}_{n_c}$
(3)   $\boldsymbol{y}_0 = \gamma_0$
(4)   $\theta_0 = 1$
(5)   for $k := 0$ to $N_{max}$
(6)       $\boldsymbol{g} = \boldsymbol{N}\boldsymbol{y}_k - \boldsymbol{r}$
(7)       $\gamma_{k+1} = \Pi_{\mathcal{K}}\left(\boldsymbol{y}_k - tg\right)$
(8)       $\theta_{k+1} = \frac{-\theta_k^2 + \theta_k\sqrt{\theta_k^2+4}}{2}$
(9)       $\beta_{k+1} = \theta_k \frac{1-\theta_k}{\theta_k^2 + \theta_{k+1}}$
(10)      $\boldsymbol{y}_{k+1} = \gamma_{k+1} + \beta_{k+1}\left(\gamma_{k+1} - \gamma_k\right)$
(11)      $\epsilon = \epsilon\left(\gamma_{k+1}\right)$
(12)      if $\epsilon < \tau$
(13)          break
(14)      endif
(15)  endfor
(16)  return Value at time step $t_{l+1}, \gamma^{l+1} := \hat{\gamma}$ .
```

APGD – Built around Nesterov

- APGD: Accelerated Projected Gradient Descent
  - Idea: instead of descending along the gradient, use a linear combination of previous descent directions

- Proved to be, up to a factor c, the best first order optimization method
  - Convergences like $O(1/k^2)$ as opposed to $O(1/k)$

- Projected version implemented owing to presence of conic constraints

“On the Modeling, Simulation, and Visualization of Many-Body Dynamics Problems with Friction and Contact,” T. Heyn, PhD Thesis, 2013

APGD solver for CCP

- Use `SetSolverType()` to change the solver:

```
// change the solver to Nesterov’ APGD:
my_system.SetSolverType(ChSystem::SOLVER_APGD);

// will terminate iterations when this tolerance is reached:
my_system.SetTolForce(1e-7);

// use high iteration number if constraints tend to ‘dismount’ or contacts interpenetrate:
my_system.SetMaxItersSolverSpeed(110);
```

P-SPG-FB

```
ALGORITHM P-SPG-FB($\mathbf{N}$, $\mathbf{r}$, $\mathbf{x}_0$, $\mathcal{K}$, $\mathbf{P} \mapsto \mathbf{x}$)
(1)   $\mathbf{x}_0 := \Pi_{\mathcal{K}}(\mathbf{x_0})$, $\mathbf{x}_{FB} = \mathbf{x}_0$, $\hat{\alpha}_0 \in [\alpha_{min}, \alpha_{max}]$
(2)   $\mathbf{g}_0 := \mathbf{N}\mathbf{x}_0 + \mathbf{r}$, $f(\mathbf{x}_0) = \frac{1}{2}\mathbf{x}_0^T\mathbf{N}\mathbf{x}_0 + \mathbf{x}_0^T\mathbf{r}$, $w_0 = 10^{29}$
(3)   for $j := 0$ to $N_{max}$
(4)       $\mathbf{p}_j = \mathbf{P^{-1}g_j}$
(5)       $\mathbf{d}_j = \Pi_{\mathcal{K}}(\mathbf{x}_j - \hat{\alpha}_j\mathbf{p}_j) - \mathbf{x}_j$
(6)       if $\langle \mathbf{d}_j, \mathbf{g}_j \rangle \geq 0$
(7)           $\mathbf{d}_j = \Pi_{\mathcal{K}}(\mathbf{x}_j - \hat{\alpha}_j\mathbf{g}_j) - \mathbf{x}_j$
(8)       $\lambda := 1$
(9)       while line search
(10)          $\mathbf{x}_{j+1} := \mathbf{x}_j + \lambda\mathbf{d}_j$
(11)          $\mathbf{g}_{j+1} := \mathbf{N}\mathbf{x}_{j+1} + \mathbf{r}$
(12)          $f(\mathbf{x}_{j+1}) = \frac{1}{2}\mathbf{x}_{j+1}^T\mathbf{N}\mathbf{x}_{j+1} + \mathbf{x}_{j+1}^T\mathbf{r}$
(13)          if $f(\mathbf{x}_{j+1}) > \max_{i=0,..,\min(j,N_{GLL})} f(\mathbf{x}_{j-i}) + \gamma\lambda\langle \mathbf{d}_j, \mathbf{g}_j \rangle$
(14)              define $\lambda_{\text{new}} \in [\sigma_{\min}\lambda, \sigma_{\max}\lambda]$ and repeat line search
(15)          else
(16)              terminate line search
(17)      $\mathbf{s}_j = \mathbf{x}_{j+1} - \mathbf{x}_j$
(18)      $\mathbf{y}_j = \mathbf{g}_{j+1} - \mathbf{g}_j$
(19)      if $j$ is odd
(20)          $\hat{\alpha}_{j+1} = \frac{\langle \mathbf{s}_j, \mathbf{Ps_j} \rangle}{\langle \mathbf{s}_j, \mathbf{y}_j \rangle}$
(21)      else
(22)          $\hat{\alpha}_{j+1} = \frac{\langle \mathbf{s}_j, \mathbf{y}_j \rangle}{\langle \mathbf{y}_j, \mathbf{P^{-1}y_j} \rangle}$
(23)      $\hat{\alpha}_{j+1} = \min(\alpha_{\max}, \max(\alpha_{\min}, \hat{\alpha}_{j+1}))$
(24)      $w_{j+1} = ||[\mathbf{x}_{j+1} - \Pi_{\mathcal{K}}(\mathbf{x}_{j+1} - \tau_g\mathbf{g}_{j+1})]/\tau_g||_2 = ||\epsilon||_2$
(25)      if $w_{j+1} \leq \min_{k=0,..,j} w_k$
(26)          $\mathbf{x}_{FB} = \mathbf{x}_{j+1}$
(27)  return $\mathbf{x}_{FB}$
```

P-SPG-FB solver for CCP

- Use `SetSolverType()` to change the solver:

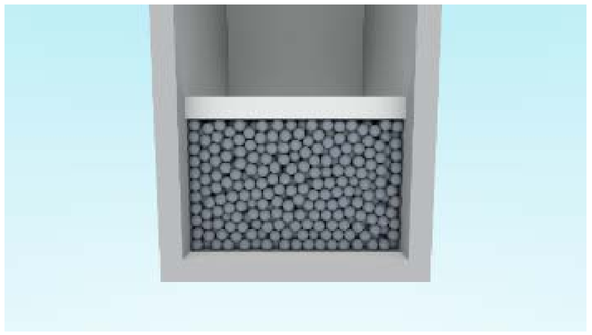
```
// change the solver to Barzilai-Borwein P-SPG-FB:
my_system.SetSolverType(ChSystem::SOLVER_BARZILAIBORWEIN);

// will terminate iterations when this tolerance is reached:
my_system.SetTolForce(1e-7);

// use high iteration number if constraints tend to ‘dismount’ or contacts interpenetrate:
my_system.SetMaxItersSolverSpeed(110);
```

New Subtopic:
Performance Comparison: Jacobi vs. GS vs. APGD

- Benchmark Problem: 4000 rigid spheres
- Heavy block/slab rests on packed spheres



- Results obtained at one step
- Mass of block varied
  - 10^3 kg to 10^6 kg

- The three-way race: how far can you get in 1000 iterations when solving the QP?

$$\gamma^* = \underset{\substack{\gamma_i \in \Upsilon_i \\ 1 \le i \le N_c}}{\operatorname{argmin}} \left(\frac{1}{2} \gamma^T \mathbf{N} \gamma + \mathbf{r}^T \gamma \right)$$

Performance Comparison: Jacobi vs. GS vs. APGD

$$\gamma^{\star} = \underset{\substack{\gamma_i \in \Upsilon_i \\ 1 \le i \le N_c}}{\operatorname{argmin}} \left(\frac{1}{2} \gamma^T \mathbf{N} \gamma + \mathbf{r}^T \gamma \right)$$

Objective Function $f(\gamma)$			
Mass [kg]	**Jacobi**	**Gauss Seidel**	**APGD**
1×10^3	-28.29	-117.70	-220.14
1×10^4	-35.63	-162.99	-883.54
1×10^5	-37.02	-176.94	-3199.27
1×10^6	-37.15	-210.23	-4696.48

“Using Nesterov’s Method to Accelerate Multibody Dynamics with Friction and Contact,” H. Mazhar, T. Heyn, A. Tasora, D. Negrut, Association for Computing Machinery TOG, 2014

Performance Comparison: Jacobi vs. GS vs. APGD

- Mass of block is 1000 kg
- The three-way race:
  - How much effort does it take to converge the solution within a 7x10^{-6} tolerance

$$\gamma^* = \underset{\substack{\gamma_i \in \Upsilon_i \\ 1 \le i \le N_c}}{\operatorname{argmin}} \left(\frac{1}{2} \gamma^T \mathbf{N} \gamma + \mathbf{r}^T \gamma \right)$$

Solver	Residual	Iterations	Time[s]
Jacobi	7.54×10^{-6} (UtC)	500 000	24 300
Gauss Seidel	6.99×10^{-6}	11 485	494.8
APGD	6.97×10^{-6}	202	10.6

Time Integration & Solvers Cheat Sheet

	LINEAR SYSTEM (DAE)		CCP (DVI)		Iterative	Redundant constraints	Optional Chrono module	Large system	Time integrator compatibility	
		FEA*		FEA*					INT_HHT	INT_EULER_IMPLICIT_LINEARIZED
SOR	🟦	*	●●	*	🟦	●●		●●●	2	●●● DAE, DVI
BARZILAIBORWEIN	🟦	*	●●●	*	🟦	●●		●●●	● DAE	●●● DAE, DVI
APGD	🟦	*	●●●	*	🟦	●●		●●●	● DAE	●●● DAE, DVI
MINRES	🟦	●[1]		*	🟦	●●●		●●●	●● DAE	● DAE
MKL	🟦	●●●		*			🟦	●●	●●● DAE	● DAE
MUMPS	🟦	●●●		*		●	🟦	●●	●●● DAE	● DAE

* For FEA, the solver must support stiffness and damping matrices. Note that FEA in DVI is not yet possible at the moment.

1 The MINRES solver might converge too slow when using finite elements with ill-conditioned stiffness

2 The SOR solver is not precise enough for good HHT convergence, except for simple systems