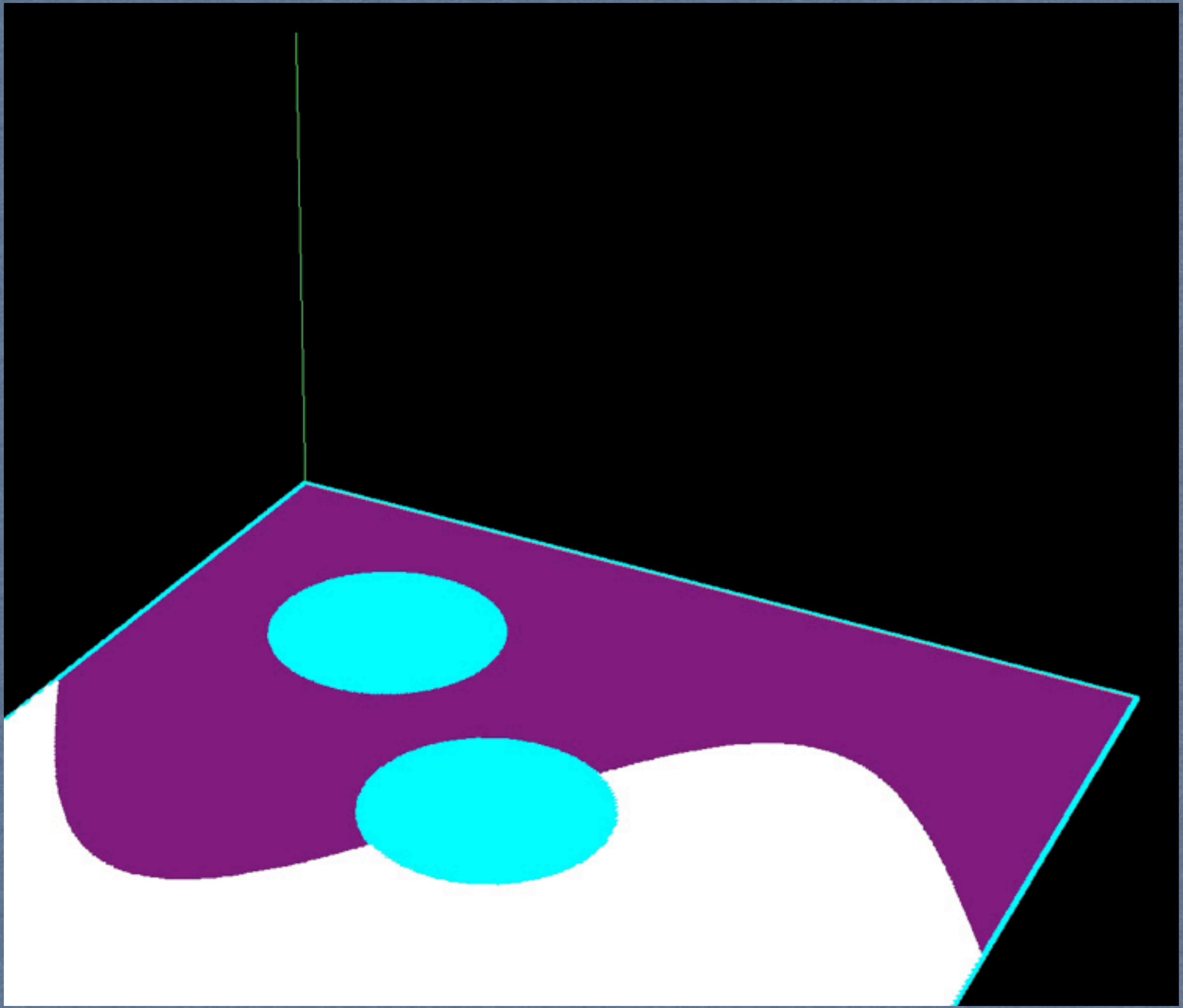
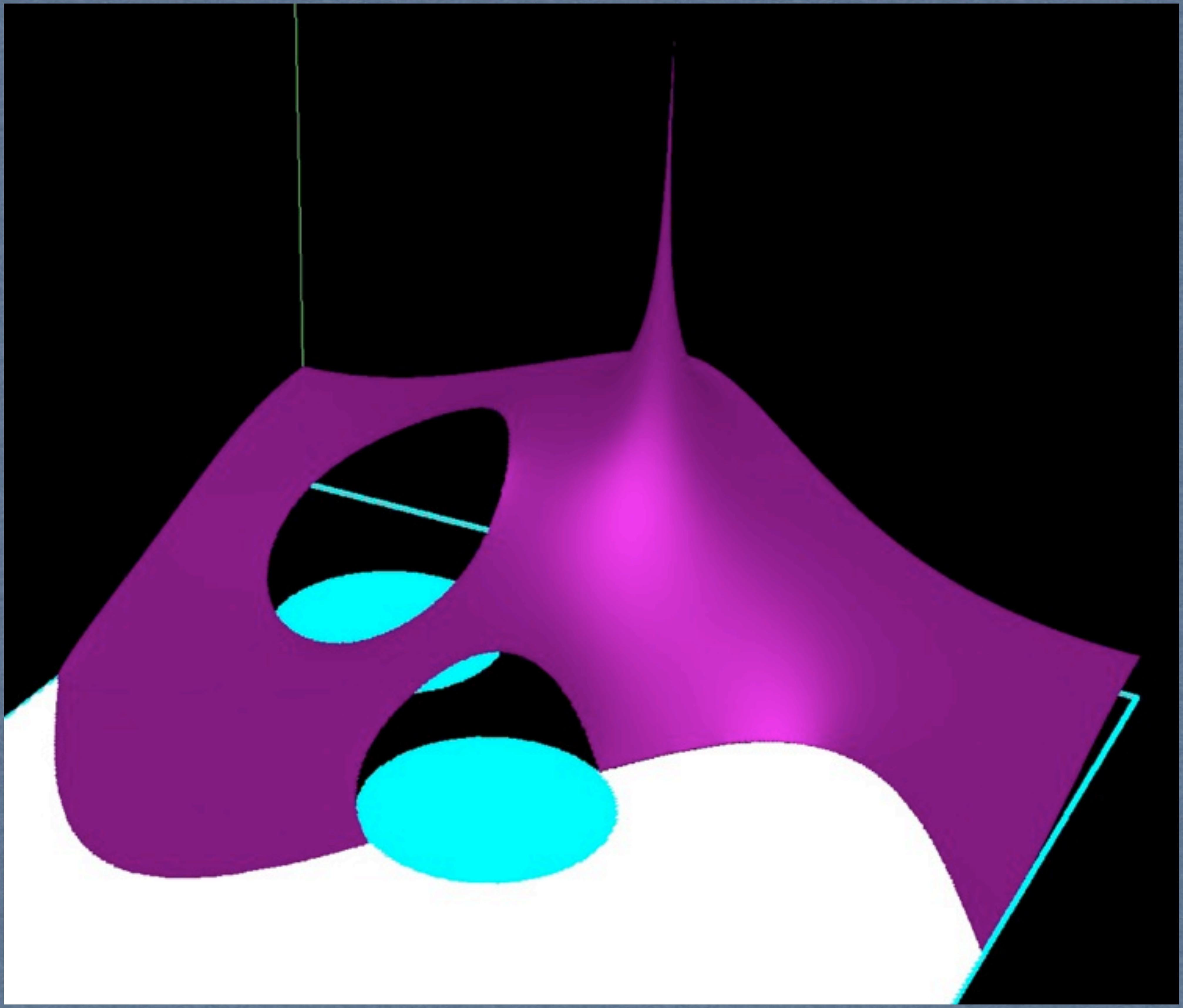
Conjugate Gradients

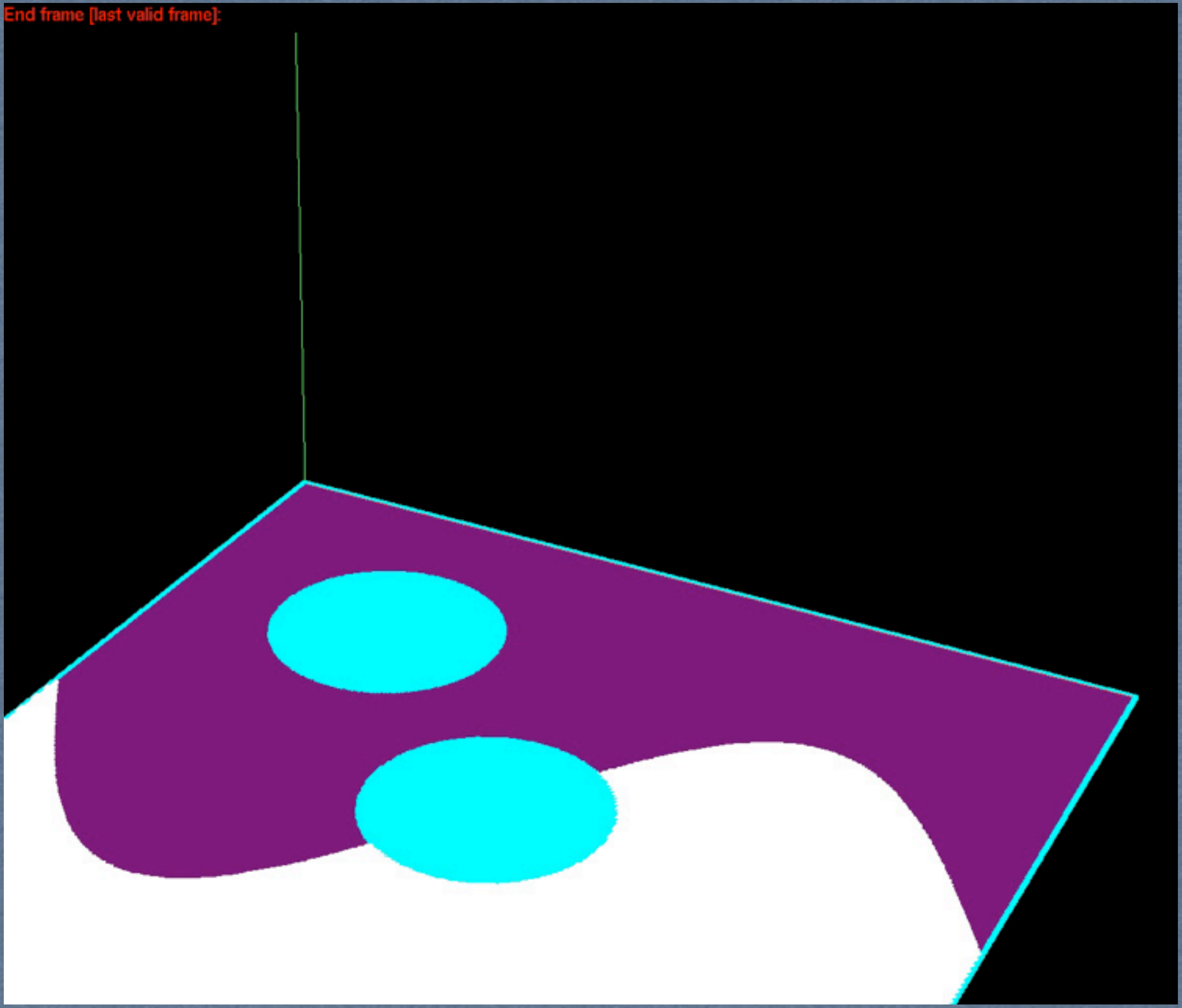
- Solves (or approximates the solution) of Ax=b
  - A must be symmetric and positive definite
- Iterative algorithm
  - Starts with an initial guess x0
  - Generates a sequence of approximations x0, x1, x2, ..., xk
  - Guaranteed to converge in n iterations (if A is nxn)
  - Typically (and hopefully) converges faster than that!
  - Via "preconditioning", convergence can be accelerated



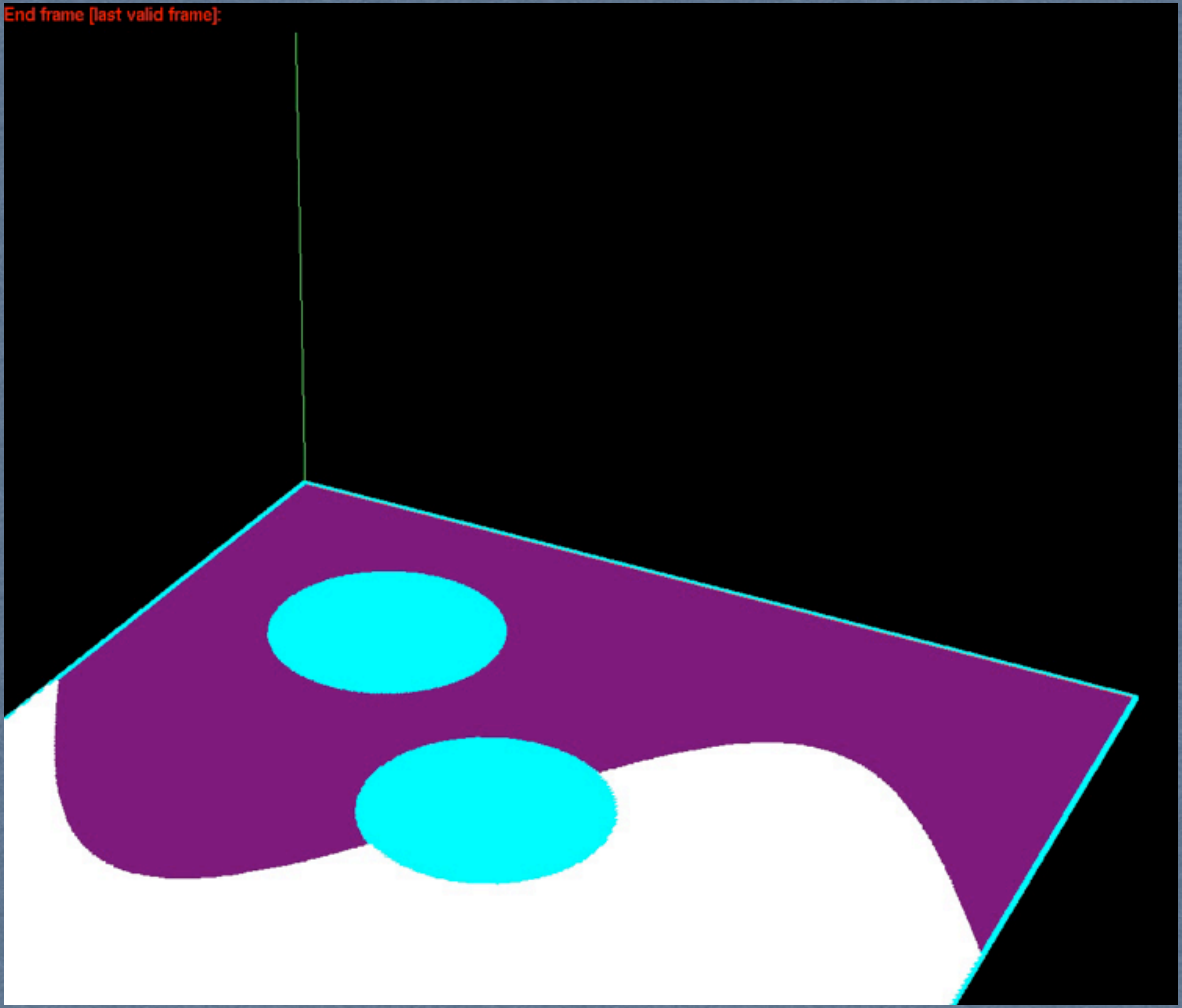
Sample 2D domain (512x512 resolution)



Exact solution (i.e. Green's function; RHS is a delta impulse)



Convergence of the un-preconditioned CG algorithm



CG with an Incomplete Cholesky preconditioner

Conjugate Gradients

$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$d \Leftarrow r$
$\delta_{new} \Leftarrow r^T r$
$\delta_0 \Leftarrow \delta_{new}$
While $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
    $q \Leftarrow Ad$
    $\alpha \Leftarrow \frac{\delta_{new}}{d^T q}$
    $x \Leftarrow x + \alpha d$
    If i is divisible by 50
        $r \Leftarrow b - Ax$
    else
        $r \Leftarrow r - \alpha q$
    $\delta_{old} \Leftarrow \delta_{new}$
    $\delta_{new} \Leftarrow r^T r$
    $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \Leftarrow r + \beta d$
    $i \Leftarrow i + 1$

Conjugate Gradients

- Implementation option #1
  - Bake out matrix A, and vectors x & b explicitly
    - Potentially costly and inconvenient
  - Perform a straightforward implementation
- Implementation option #2
  - Masquerade your *native* description of A & b as something that *acts* like a matrix & a vector, respectively
  - Use a C++ wrapper to make your native description convey the necessary functionality (dot products, matrix-vector products, etc).

Conjugate Gradients

$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$d \Leftarrow r$
$\delta_{new} \Leftarrow r^T r$
$\delta_0 \Leftarrow \delta_{new}$
While $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
    $q \Leftarrow Ad$
    $\alpha \Leftarrow \frac{\delta_{new}}{d^T q}$
    $x \Leftarrow x + \alpha d$
    If i is divisible by 50
        $r \Leftarrow b - Ax$
    else
        $r \Leftarrow r - \alpha q$
    $\delta_{old} \Leftarrow \delta_{new}$
    $\delta_{new} \Leftarrow r^T r$
    $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \Leftarrow r + \beta d$
    $i \Leftarrow i + 1$

Conjugate Gradients

$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$d \Leftarrow r$
$\delta_{new} \Leftarrow r^T r$
$\delta_0 \Leftarrow \delta_{new}$
While $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
    $q \Leftarrow \boxed{Ad}$
    $\alpha \Leftarrow \frac{\delta_{new}}{d^T q}$
    $x \Leftarrow x + \alpha d$
    If i is divisible by 50
        $r \Leftarrow b - \boxed{Ax}$
    else
        $r \Leftarrow r - \alpha q$
    $\delta_{old} \Leftarrow \delta_{new}$
    $\delta_{new} \Leftarrow r^T r$
    $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \Leftarrow r + \beta d$
    $i \Leftarrow i + 1$

Conjugate Gradients

$i \Leftarrow 0$
$r \Leftarrow b - Ax$
$d \Leftarrow r$
$\delta_{new} \Leftarrow \boxed{r^T r}$
$\delta_0 \Leftarrow \delta_{new}$
While $i < i_{max}$ and $\delta_{new} > \varepsilon^2 \delta_0$ do
    $q \Leftarrow Ad$
    $\alpha \Leftarrow \frac{\delta_{new}}{\boxed{d^T q}}$
    $x \Leftarrow x + \alpha d$
    If i is divisible by 50
        $r \Leftarrow b - Ax$
    else
        $r \Leftarrow r - \alpha q$
    $\delta_{old} \Leftarrow \delta_{new}$
    $\delta_{new} \Leftarrow \boxed{r^T r}$
    $\beta \Leftarrow \frac{\delta_{new}}{\delta_{old}}$
    $d \Leftarrow r + \beta d$
    $i \Leftarrow i + 1$

Conjugate Gradients

- Implementation option #1
  - Bake out matrix A, and vectors x & b explicitly
    - Potentially costly and inconvenient
  - Perform a straightforward implementation
- Implementation option #2
  - Masquerade your *native* description of A & b as something that *acts* like a matrix & a vector, respectively
  - Use a C++ wrapper to make your native description convey the necessary functionality (dot products, matrix-vector products, etc).

```
    template<class T>
class KRYLOV_VECTOR_BASE
{
public:
    // Predefined - NO NEED TO WORRY ABOUT THOSE
    KRYLOV_VECTOR_BASE();
    virtual ~KRYLOV_VECTOR_BASE();
    const KRYLOV_VECTOR_BASE& operator=(const KRYLOV_VECTOR_BASE& bv);
    const T& Raw_Get(int i) const;

    // Pure virtual - MUST OVERLOAD
    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
//#####################################################################
};
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_VECTOR_BASE.h

```
template<class T>
class KRYLOV_VECTOR_BASE
{
public:
    // Predefined - NO NEED TO WORRY ABOUT THOSE
    KRYLOV_VECTOR_BASE();
    virtual ~KRYLOV_VECTOR_BASE();
    const KRYLOV_VECTOR_BASE& operator=(const KRYLOV_VECTOR_BASE& bv);
    const T& Raw_Get(int i) const;

    // Pure virtual - MUST OVERLOAD
    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
//#####################################################################
};
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_VECTOR_BASE.h

```
template<class T>
class KRYLOV_VECTOR_BASE
{
public:
    // Predefined - NO NEED TO WORRY ABOUT THOSE
    KRYLOV_VECTOR_BASE();
    virtual ~KRYLOV_VECTOR_BASE();
    const KRYLOV_VECTOR_BASE& operator=(const KR
    const T& Raw_Get(int i) const;

    // Pure virtual - MUST OVERLOAD
    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
//#####################################################################
};
```

Semantics

```
obj1+=obj2;

means:

forall i
  obj1(i)+=obj2(i);
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_VECTOR_BASE.h

```
Semantics

  obj1.Copy(c,obj2);

  means:

  forall i
    obj1(i)=c*obj2(i);
```

```
template<class
class KRYLOV_V
{
public:
    // Predefi
    KRYLOV_VEC
    virtual ~K
    const KRYL
    const T& Raw_Get(int i) const;

    // Pure virtual - MUST OVERLOAD
    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
//#######################################################################
};
```

```
template<class
class KRYLOV_V
{
public:
    // Predefi
    KRYLOV_VEC
    virtual ~K
    const KRYL
    const T& R

    // Pure virtual - MUST OVERLOAD
    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
//#####################################################################
};
```

Semantics

```
obj1.Copy(c,obj2,obj3);

means:

forall i
  obj1(i)=c*obj2(i)+obj3(i);
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_VECTOR_BASE.h

```
template<class T>
class KRYLOV_VECTOR_BASE
{
public:
    // Predefined - NO NEED TO WORRY ABOUT THOSE
    KRYLOV_VECTOR_BASE();
    virtual ~KRYLOV_VECTOR_BASE();
    const KRYLOV_VECTOR_BASE& operator=(const KRYLOV_VECTOR_BASE& bv);
    const T& Raw_Get(int i) const;

    // Pure virtual - MUST OVERLOAD
    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
//#####################################################################
};
```

```
   template<class T> class CG_VECTOR:public KRYLOV_VECTOR_BASE<T>
{
    typedef KRYLOV_VECTOR_BASE<T> BASE;

    ARRAY<VECTOR<T,3> >& array;
public:
    CG_VECTOR(ARRAY<VECTOR<T,3> >& array_input) : array(array_input) {}

    static ARRAY<VECTOR<T,3> >& Array(BASE& base_array)
    {return ((CG_VECTOR&)(base_array)).array;}

    static const ARRAY<VECTOR<T,3> >& Array(const BASE& base_array)
    {return ((const CG_VECTOR&)(base_array)).array;}

    BASE& operator+=(const BASE& bv)
    {array+=Array(bv);return *this;}

    BASE& operator-=(const BASE& bv)
    {array-=Array(bv);return *this;}

    BASE& operator*=(const T a)
    {array*=a;return *this;}

    void Copy(const T c,const BASE& bv)
    {ARRAY<VECTOR<T,3> >::Copy(c,Array(bv),array);}

    void Copy(const T c1,const BASE& bv1,const BASE& bv2)
    {ARRAY<VECTOR<T,3> >::Copy(c1,Array(bv1),Array(bv2),array);}

    int Raw_Size() const
    {return array.Flattened().m;}

    T& Raw_Get(int i)
    {return array.Flattened()(i);}
};
```

```
template<class T> class CG_VECTOR:public KRYLOV_VECTOR_BASE<T>
{
    typedef KRYLOV_VECTOR_BASE<T> BASE;

    ARRAY<VECTOR<T,3> >& array;
public:
    CG_VECTOR(ARRAY<VECTOR<T,3> >& array_input) : array(array_input) {}

    static ARRAY<VECTOR<T,3> >& Array(BASE& base_array)
    {return ((CG_VECTOR&)(base_array)).array;}

    static const ARRAY<VECTOR<T,3> >& Array(const BASE& base_array)
    {return ((const CG_VECTOR&)(base_array)).array;}

    BASE& operator+=(const BASE& bv)
    {array+=Array(bv);return *this;}

    BASE& operator-=(const BASE& bv)
    {array-=Array(bv);return *this;}

    BASE& operator*=(const T a)
    {array*=a;return *this;}

    void Copy(const T c,const BASE& bv)
    {ARRAY<VECTOR<T,3> >::Copy(c,Array(bv),array);}

    void Copy(const T c1,const BASE& bv1,const BASE& bv2)
    {ARRAY<VECTOR<T,3> >::Copy(c1,Array(bv1),Array(bv2),array);}

    int Raw_Size() const
    {return array.Flattened().m;}

    T& Raw_Get(int i)
    {return array.Flattened()(i);}
};
```

```
template<class T> class CG_VECTOR:public KRYLOV_VECTOR_BASE<T>
{
    typedef KRYLOV_VECTOR_BASE<T> BASE;

    ARRAY<VECTOR<T,3> >& array;
public:
    CG_VECTOR(ARRAY<VECTOR<T,3> >& array_input) : array(array_input) {}

    static ARRAY<VECTOR<T,3> >& Array(BASE& base_array)
    {return ((CG_VECTOR&)(base_array)).array;}

    static const ARRAY<VECTOR<T,3> >& Array(const BASE& base_array)
    {return ((const CG_VECTOR&)(base_array)).array;}

    BASE& operator+=(const BASE& bv)
    {array+=Array(bv);return *this;}

    BASE& operator-=(const BASE& bv)
    {array-=Array(bv);return *this;}

    BASE& operator*=(const T a)
    {array*=a;return *this;}

    void Copy(const T c,const BASE& bv)
    {ARRAY<VECTOR<T,3> >::Copy(c,Array(bv),array);}

    void Copy(const T c1,const BASE& bv1,const BASE& bv2)
    {ARRAY<VECTOR<T,3> >::Copy(c1,Array(bv1),Array(bv2),array);}

    int Raw_Size() const
    {return array.Flattened().m;}

    T& Raw_Get(int i)
    {return array.Flattened()(i);}
};
```

```
template<class T> class CG_VECTOR:public KRYLOV_VECTOR_BASE<T>
{
    typedef KRYLOV_VECTOR_BASE<T> BASE;

    ARRAY<VECTOR<T,3> >& array;
public:
    CG_VECTOR(ARRAY<VECTOR<T,3> >& array_input) : array(array_input) {}

    static ARRAY<VECTOR<T,3> >& Array(BASE& base_array)
    {return ((CG_VECTOR&)(base_array)).array;}

    static const ARRAY<VECTOR<T,3> >& Array(const BASE& base_array)
    {return ((const CG_VECTOR&)(base_array)).array;}

    BASE& operator+=(const BASE& bv)
    {array+=Array(bv);return *this;}

    BASE& operator-=(const BASE& bv)
    {array-=Array(bv);return *this;}

    BASE& operator*=(const T a)
    {array*=a;return *this;}

    void Copy(const T c,const BASE& bv)
    {ARRAY<VECTOR<T,3> >::Copy(c,Array(bv),array);}

    void Copy(const T c1,const BASE& bv1,const BASE& bv2)
    {ARRAY<VECTOR<T,3> >::Copy(c1,Array(bv1),Array(bv2),array);}

    int Raw_Size() const
    {return array.Flattened().m;}

    T& Raw_Get(int i)
    {return array.Flattened()(i);}
};
```

```
template<class T> class CG_VECTOR:public KRYLOV_VECTOR_BASE<T>
{
    typedef KRYLOV

    ARRAY<VECTOR<T
public:
    CG_VECTOR(ARRA

    static ARRAY<V
    {return ((CG_V

    static const A
    {return ((cons
```

From KRYLOV_VECTOR_BASE<T> :

```
    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,
        const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
```

```
    BASE& operator+=(const BASE& bv)
    {array+=Array(bv);return *this;}

    BASE& operator-=(const BASE& bv)
    {array-=Array(bv);return *this;}

    BASE& operator*=(const T a)
    {array*=a;return *this;}

    void Copy(const T c,const BASE& bv)
    {ARRAY<VECTOR<T,3> >::Copy(c,Array(bv),array);}

    void Copy(const T c1,const BASE& bv1,const BASE& bv2)
    {ARRAY<VECTOR<T,3> >::Copy(c1,Array(bv1),Array(bv2),array);}

    int Raw_Size() const
    {return array.Flattened().m;}

    T& Raw_Get(int i)
    {return array.Flattened()(i);}
};
```

```
template<class T> class CG_VECTOR:public KRYLOV_VECTOR_BASE<T>
{
    typedef KRYLOV_

    ARRAY<VECTOR<T
public:
    CG_VECTOR(ARRA

    static ARRAY<V
    {return ((CG_V

    static const A
    {return ((cons
```

```
From KRYLOV_VECTOR_BASE<T> :

    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,
        const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
```

```
    BASE& operator+=(const BASE& bv)
    {array+=Array(bv);return *this;}

    BASE& operator-=(const BASE& bv)
    {array-=Array(bv);return *this;}

    BASE& operator*=(const T a)
    {array*=a;return *this;}

    void Copy(const T c,const BASE& bv)
    {ARRAY<VECTOR<T,3> >::Copy(c,Array(bv),array);}

    void Copy(const T c1,const BASE& bv1,const BASE& bv2)
    {ARRAY<VECTOR<T,3> >::Copy(c1,Array(bv1),Array(bv2),array);}

    int Raw_Size() const
    {return array.Flattened().m;}

    T& Raw_Get(int i)
    {return array.Flattened()(i);}
};
```

```
template<class T> class CG_VECTOR:public KRYLOV_VECTOR_BASE<T>
{
    typedef KRYLOV

    ARRAY<VECTOR<T
public:
    CG_VECTOR(ARRA

    static ARRAY<V
    {return ((CG_V

    static const A
    {return ((cons

    BASE& operator+=(const BASE& bv)
    {array+=Array(bv);return *this;}

    BASE& operator-=(const BASE& bv)
    {array-=Array(bv);return *this;}

    BASE& operator*=(const T a)
    {array*=a;return *this;}

    void Copy(const T c,const BASE& bv)
    {ARRAY<VECTOR<T,3> >::Copy(c,Array(bv),array);}

    void Copy(const T c1,const BASE& bv1,const BASE& bv2)
    {ARRAY<VECTOR<T,3> >::Copy(c1,Array(bv1),Array(bv2),array);}

    int Raw_Size() const
    {return array.Flattened().m;}

    T& Raw_Get(int i)
    {return array.Flattened()(i);}
};
```

```
From KRYLOV_VECTOR_BASE<T> :

    virtual KRYLOV_VECTOR_BASE& operator+=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator-=(const KRYLOV_VECTOR_BASE& bv)=0;
    virtual KRYLOV_VECTOR_BASE& operator*=(const T a)=0;
    virtual void Copy(const T c,const KRYLOV_VECTOR_BASE& bv)=0;
    virtual void Copy(const T c1,const KRYLOV_VECTOR_BASE& bv1,
        const KRYLOV_VECTOR_BASE& bv2)=0;
    virtual int Raw_Size() const=0;
    virtual T& Raw_Get(int i)=0;
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_SYSTEM_BASE.h

```
    template<class T>
class KRYLOV_SYSTEM_BASE
{

    // Predefined - NO NEED TO WORRY ABOUT THOSE
public:
    KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
    virtual ~KRYLOV_SYSTEM_BASE();
    void Test_System(KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& y,KRYLOV_VECTOR_BASE<T>& z) const;
    const KRYLOV_VECTOR_BASE<T>& Precondition(const KRYLOV_VECTOR_BASE<T>& r,
        KRYLOV_VECTOR_BASE<T>& z) const;

    // Pure virtual - MUST OVERLOAD
public:
    virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
    virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
    virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;

    // Pure virtual - OVERLOAD WITH BLANK BODY (for solids)
    virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
protected:
    virtual void Apply_Preconditioner(const KRYLOV_VECTOR_BASE<T>& r,KRYLOV_VECTOR_BASE<T>& z) const;

};
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_SYSTEM_BASE.h

Initialize with (false,false) if using unpreconditioned CG

```
template<class T>
class KRYLOV_SYSTEM_BASE
{

    // Predefined - NO NEED TO WORRY ABOUT THOSE
public:
    KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
    virtual ~KRYLOV_SYSTEM_BASE();
    void Test_System(KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& y,KRYLOV_VECTOR_BASE<T>& z) const;
    const KRYLOV_VECTOR_BASE<T>& Precondition(const KRYLOV_VECTOR_BASE<T>& r,
        KRYLOV_VECTOR_BASE<T>& z) const;

    // Pure virtual - MUST OVERLOAD
public:
    virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
    virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
    virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;

    // Pure virtual - OVERLOAD WITH BLANK BODY (for solids)
    virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
protected:
    virtual void Apply_Preconditioner(const KRYLOV_VECTOR_BASE<T>& r,KRYLOV_VECTOR_BASE<T>& z) const;

};
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_SYSTEM_BASE.h

```
template<class T>
class KRYLOV_SYSTEM_BASE
{

    // Predefined - NO N
public:
    KRYLOV_SYSTEM_BASE(bo
    virtual ~KRYLOV_SYSTE
    void Test_System(KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& y,KRYLOV_VECTOR_BASE<T>& z) const;
    const KRYLOV_VECTOR_BASE<T>& Precondition(const KRYLOV_VECTOR_BASE<T>& r,
        KRYLOV_VECTOR_BASE<T>& z) const;

    // Pure virtual - MUST OVERLOAD
public:
    virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
    virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
    virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;

    // Pure virtual - OVERLOAD WITH BLANK BODY (for solids)
    virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
protected:
    virtual void Apply_Preconditioner(const KRYLOV_VECTOR_BASE<T>& r,KRYLOV_VECTOR_BASE<T>& z) const;

};
```

Semantics

```
Multiply(x,y);

means:

y:=Ax; (A is the matrix “implied” by this class, x&y are vectors)
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_SYSTEM_BASE.h

Semantics

```
a=Inner_Product(x,y);

means:

a=Sum(x(i)*y(i)) [for all i]
```

```
template<class T>
class KRYLOV_SYSTEM_BASE
{

    // Predefined - NO N
public:
    KRYLOV_SYSTEM_BASE(bo
    virtual ~KRYLOV_SYSTE
    void Test_System(KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& y,KRYLOV_VECTOR_BASE<T>& z) const;
    const KRYLOV_VECTOR_BASE<T>& Precondition(const KRYLOV_VECTOR_BASE<T>& r,
        KRYLOV_VECTOR_BASE<T>& z) const;

    // Pure virtual - MUST OVERLOAD
public:
    virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
    virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
    virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;

    // Pure virtual - OVERLOAD WITH BLANK BODY (for solids)
    virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
protected:
    virtual void Apply_Preconditioner(const KRYLOV_VECTOR_BASE<T>& r,KRYLOV_VECTOR_BASE<T>& z) const;

};
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_SYSTEM_BASE.h

```
template<class T>
class KRYLOV_SYSTEM_BASE
{

    // Predefined - NO N
public:
    KRYLOV_SYSTEM_BASE(bo
    virtual ~KRYLOV_SYSTE
    void Test_System(KRYI                                                    st;
    const KRYLOV_VECTOR_E
        KRYLOV_VECTOR_BASE<T>& z) const;

    // Pure virtual - MUST OVERLOAD
public:
    virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
    virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
    virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;

    // Pure virtual - OVERLOAD WITH BLANK BODY (for solids)
    virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
protected:
    virtual void Apply_Preconditioner(const KRYLOV_VECTOR_BASE<T>& r,KRYLOV_VECTOR_BASE<T>& z) const;

};
```

Semantics

```
a=Convergence_Norm(x);

means:

a=MAX(Norm(x(i))) [for all i]
Norm() is any desired norm function, e.g. max, Euclidean, etc.
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_SYSTEM_BASE.h

Semantics

```
Project(x);

means:

For all constrained i : x(i):=0;
```

```
template<class T>
class KRYLOV_SYSTEM_BASE
{

    // Predefined - NO N
public:
    KRYLOV_SYSTEM_BASE(bo
    virtual ~KRYLOV_SYSTE
    void Test_System(KRYL                                                              const;
    const KRYLOV_VECTOR_BASE<T>& Precondition(const KRYLOV_VECTOR_BASE<T>& r,
        KRYLOV_VECTOR_BASE<T>& z) const;

    // Pure virtual - MUST OVERLOAD
public:
    virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
    virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
    virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;

    // Pure virtual - OVERLOAD WITH BLANK BODY (for solids)
    virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
protected:
    virtual void Apply_Preconditioner(const KRYLOV_VECTOR_BASE<T>& r,KRYLOV_VECTOR_BASE<T>& z) const;

};
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_SYSTEM_BASE.h

```
template<class T>
class KRYLOV_SYSTEM_BASE
{

    // Predefined - NO N
public:
    KRYLOV_SYSTEM_BASE(bo
    virtual ~KRYLOV_SYSTE
    void Test_System(KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& y,KRYLOV_VECTOR_BASE<T>& z) const;
    const KRYLOV_VECTOR_BASE<T>& Precondition(const KRYLOV_VECTOR_BASE<T>& r,
        KRYLOV_VECTOR_BASE<T>& z) const;

    // Pure virtual - MUST OVERLOAD
public:
    virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
    virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
    virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;

    // Pure virtual - OVERLOAD WITH BLANK BODY (for solids)
    virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
protected:
    virtual void Apply_Preconditioner(const KRYLOV_VECTOR_BASE<T>& r,KRYLOV_VECTOR_BASE<T>& z) const;

};
```

Semantics

```
Set_Boundary_Conditions(x);

means:

For all constrained i : x(i):=<respective constrained value>;
```

PhysBAM_Tools/Krylov_Solvers/KRYLOV_SYSTEM_BASE.h

```
template<class T>
class KRYLOV_SYSTEM_BASE
{

    // Predefined - NO NEED TO WORRY ABOUT THOSE
public:
    KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
    virtual ~KRYLOV_SYSTEM_BASE();
    void Test_System(KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& y,KRYLOV_VECTOR_BASE<T>& z) const;
    const KRYLOV_VECTOR_BASE<T>& Precondition(const KRYLOV_VECTOR_BASE<T>& r,
        KRYLOV_VECTOR_BASE<T>& z) const;

    // Pure virtual - MUST OVERLOAD
public:
    virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
    virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
    virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
    virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;

    // Pure virtual - OVERLOAD WITH BLANK BODY (for solids)
    virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
protected:
    virtual void Apply_Preconditioner(const KRYLOV_VECTOR_BASE<T>& r,KRYLOV_VECTOR_BASE<T>& z) const;

};
```

```
template<class T>
class CG_SYSTEM:public KRYLOV_SYSTEM_BASE<T>
{
    typedef KRYLOV_SYSTEM_BASE<T> BASE;
    typedef KRYLOV_VECTOR_BASE<T> VECTOR_BASE;

    SIMULATION_LAYOUT<T>& layout;
    const T time;
    const T dt;

public:
    CG_SYSTEM(SIMULATION_LAYOUT<T>& layout_input,const T time_input,const T dt_input)
        :BASE(false,false),layout(layout_input),time(time_input),dt(dt_input) {}

    void Multiply(const VECTOR_BASE& v,VECTOR_BASE& result) const
    {
        const ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);
        ARRAY<VECTOR<T,3> >& result_array=CG_VECTOR<T>::Array(result);

        result_array.Fill(VECTOR<T,3>());
        layout.Add_Damping_Forces(layout.particles.X,v_array,result_array);
        for(int p=1;p<=v_array.m;p++)
            result_array(p)=layout.mass(p)*v_array(p)-dt*result_array(p);
    }

    double Inner_Product(const VECTOR_BASE& x,const VECTOR_BASE& y) const
    {
        const ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);
        const ARRAY<VECTOR<T,3> >& y_array=CG_VECTOR<T>::Array(y);

        double result=0.;
        for(int i=1;i<=x_array.m;i++)
            result+=VECTOR<T,3>::Dot_Product(x_array(i),y_array(i));
        return result;
    }
```

```
template<class T>
class CG_SYSTEM:public KRYLOV_SYSTEM_BASE<T>
{
    typedef KRYLOV_SYSTEM_BASE<T> BASE;
    typedef KRYLOV_VECTOR_BASE<T> VECTOR_BASE;

    SIMULATION_LAYOUT<T>& layout;
    const T time;
    const T dt;

public:
    CG_SYSTEM(SIMULATION_LAYOUT<T>& layout_input,const T time_input,const T dt_input)
        :BASE(false,false),layout(layout_input),time(time_input),dt(dt_input) {}

    void Multiply(const VECTOR_BASE& v,VECTOR_BASE& result) const
    {
        const ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);
        ARRAY<VECTOR<T,3> >& result_array=CG_VECTOR<T>::Array(result);

        result_array.Fill(VECTOR<T,3>());
        layout.Add_Damping_Forces(layout.particles.X,v_array,result_array);
        for(int p=1;p<=v_array.m;p++)
            result_array(p)=layout.mass(p)*v_array(p)-dt*result_array(p);
    }

    double Inner_Product(const VECTOR_BASE& x,const VECTOR_BASE& y) const
    {
        const ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);
        const ARRAY<VECTOR<T,3> >& y_array=CG_VECTOR<T>::Array(y);

        double result=0.;
        for(int i=1;i<=x_array.m;i++)
            result+=VECTOR<T,3>::Dot_Product(x_array(i),y_array(i));
        return result;
    }
```

```
template<class T>
class CG_SYSTEM:public KRYLOV_SYSTEM_BASE<T>
{
    typedef KRYLOV_SYSTEM_BASE<T> BASE;
    typedef KRYLOV_VECTOR_BASE<T> VECTOR_BASE;

    SIMULATION_LAYOUT<T>& layout;
    const T time;
    const T dt;

public:
    CG_SYSTEM(SIMULATION_LAYOUT<T>& layout_input,const T time_input,const T dt_input)
        :BASE(false,false),layout(layout_input),time(time_input),dt(dt_input) {}

    void Multiply(const VECTOR_BASE& v,VECTOR_BASE& result) const
    {
        const ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);
        ARRAY<VECTOR<T,3> >& result_array=CG_VECTOR<T>::Array(result);

        result_array.Fill(VECTOR<T,3>());
        layout.Add_Damping_Forces(layout.particles.X,v_array,result_array);
        for(int p=1;p<=v_array.m;p++)
            result_array(p)=layout.mass(p)*v_array(p)-dt*result_array(p);
    }
```

```
From KRYLOV_VECTOR_BASE<T> :

KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
```

```
template<class T>
class CG_SYSTEM:public KRYLOV_SYSTEM_BASE<T>
{
    typedef KRYLOV_SYSTEM_BASE<T> BASE;
    typedef KRYLOV_VECTOR_BASE<T> VECTOR_BASE;

    SIMULATION_LAYOUT<T>& layout;
    const T time;
    const T dt;

public:
    CG_SYSTEM(SIMULATION_LAYOUT<T>& layout_input,const T time_input,const T dt_input)
        :BASE(false,false),layout(layout_input),time(time_input),dt(dt_input) {}

    void Multiply(const VECTOR_BASE& v,VECTOR_BASE& result) const
    {
        const ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);
        ARRAY<VECTOR<T,3> >& result_array=CG_VECTOR<T>::Array(result);

        result_array.Fill(VECTOR<T,3>());
        layout.Add_Damping_Forces(layout.particles.X,v_array,result_array);
        for(int p=1;p<=v_array.m;p++)
            result_array(p)=layout.mass(p)*v_array(p)-dt*result_array(p);
    }
```

From KRYLOV_VECTOR_BASE<T> :

```
KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
```

```
From KRYLOV_VECTOR_BASE<T> :

KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
```

```
        :BASE(false,false),layout(layout_input),time(time_input),dt(dt_input) {}

    void Multiply(const VECTOR_BASE& v,VECTOR_BASE& result) const
    {
        const ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);
        ARRAY<VECTOR<T,3> >& result_array=CG_VECTOR<T>::Array(result);

        result_array.Fill(VECTOR<T,3>());
        layout.Add_Damping_Forces(layout.particles.X,v_array,result_array);
        for(int p=1;p<=v_array.m;p++)
            result_array(p)=layout.mass(p)*v_array(p)-dt*result_array(p);
    }

    double Inner_Product(const VECTOR_BASE& x,const VECTOR_BASE& y) const
    {
        const ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);
        const ARRAY<VECTOR<T,3> >& y_array=CG_VECTOR<T>::Array(y);

        double result=0.;
        for(int i=1;i<=x_array.m;i++)
            result+=VECTOR<T,3>::Dot_Product(x_array(i),y_array(i));
        return result;
    }
```

```
T Convergence_Norm(const VECTOR_BASE& x) const
{
    const ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);

    T result=0.;
    for(int i=1;i<=x_array.m;i++)
        result=std::max(result,x_array(i).Magnitude());
    return result;
}

void Project(VECTOR_BASE& x) const
{
    ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);

    layout.Clear_Values_Of_Kinematic_Particles(x_array);
}

void Set_Boundary_Conditions(VECTOR_BASE& v) const
{
    ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);

    layout.Set_Kinematic_Velocities(time+dt,v_array);
}

void Project_Nullspace(VECTOR_BASE& x) const {}
};
```

```
From KRYLOV_VECTOR_BASE<T> :

KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
```

```
    T Convergence_Norm(const VECTOR_BASE& x) const
    {
        const ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);

        T result=0.;
        for(int i=1;i<=x_array.m;i++)
            result=std::max(result,x_array(i).Magnitude());
        return result;
    }

    void Project(VECTOR_BASE& x) const
    {
        ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);

        layout.Clear_Values_Of_Kinematic_Particles(x_array);
    }

    void Set_Boundary_Conditions(VECTOR_BASE& v) const
    {
        ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);

        layout.Set_Kinematic_Velocities(time+dt,v_array);
    }

    void Project_Nullspace(VECTOR_BASE& x) const {}
};
```

```
From KRYLOV_VECTOR_BASE<T> :

KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
```

```
    T Convergence_Norm(const VECTOR_BASE& x) const
    {
        const ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);

        T result=0.;
        for(int i=1;i<=x_array.m;i++)
            result=std::max(result,x_array(i).Magnitude());
        return result;
    }

    void Project(VECTOR_BASE& x) const
    {
        ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);

        layout.Clear_Values_Of_Kinematic_Particles(x_array);
    }

    void Set_Boundary_Conditions(VECTOR_BASE& v) const
    {
        ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);

        layout.Set_Kinematic_Velocities(time+dt,v_array);
    }

    void Project_Nullspace(VECTOR_BASE& x) const {}
};
```

From KRYLOV_VECTOR_BASE<T> :

```
KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
```

```
    T Convergence_Norm(const VECTOR_BASE& x) const
    {
        const ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);

        T result=0.;
        for(int i=1;i<=x_array.m;i++)
            result=std::max(result,x_array(i).Magnitude());
        return result;
    }

    void Project(VECTOR_BASE& x) const
    {
        ARRAY<VECTOR<T,3> >& x_array=CG_VECTOR<T>::Array(x);

        layout.Clear_Values_Of_Kinematic_Particles(x_array);
    }

    void Set_Boundary_Conditions(VECTOR_BASE& v) const
    {
        ARRAY<VECTOR<T,3> >& v_array=CG_VECTOR<T>::Array(v);

        layout.Set_Kinematic_Velocities(time+dt,v_array);
    }

    void Project_Nullspace(VECTOR_BASE& x) const {}
};
```

From KRYLOV_VECTOR_BASE<T> :

```
KRYLOV_SYSTEM_BASE(bool use_preconditioner,bool preconditioner_commutes_with_projection);
virtual void Multiply(const KRYLOV_VECTOR_BASE<T>& x,KRYLOV_VECTOR_BASE<T>& result) const=0;
virtual double Inner_Product(const KRYLOV_VECTOR_BASE<T>& x,const KRYLOV_VECTOR_BASE<T>& y) const=0;
virtual T Convergence_Norm(const KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Set_Boundary_Conditions(KRYLOV_VECTOR_BASE<T>& x) const=0;
virtual void Project_Nullspace(KRYLOV_VECTOR_BASE<T>& x) const=0;
```

```
template<class T>
T SIMULATION_LAYOUT<T>::Maximum_Dt()
{
    T maximum_dt=FLT_MAX;
    for(int s=1;s<=wire_curve->mesh.elements.m;s++)
        maximum_dt=std::min(maximum_dt,2*damping_coefficient*restlength(s)/youngs_modulus);

    return CFL_number*maximum_dt;
}
```

$$dt < \frac{2bl_0}{k}$$

```
template<class T>
void SIMULATION_DRIVER<T>::Simulate_Time_Step(const T time,const T dt)
{
    layout.Set_Kinematic_Positions(time,layout.particles.X);
    layout.Set_Kinematic_Velocities(time,layout.particles.V);

    // Construct right-hand-side
    ARRAY<TV> rhs(layout.n);
    layout.Add_Elastic_Forces(layout.particles.X,rhs);
    layout.Add_External_Forces(rhs);
    for(int p=1;p<=layout.n;p++)
        rhs(p)=layout.mass(p)*layout.particles.V(p)+dt*rhs(p);

    // Use previous velocities as initial guess for next velocities
    ARRAY<TV> V_next(layout.particles.V);

    // Temporary vectors, required by Conjugate Gradients
    ARRAY<TV> temp_q(layout.n),temp_s(layout.n),temp_r(layout.n),
        temp_k(layout.n),temp_z(layout.n);

    // Encapsulate all vectors in CG-mandated format
    CG_VECTOR<T> cg_x(V_next),cg_b(rhs),
        cg_q(temp_q),cg_s(temp_s),cg_r(temp_r),cg_k(temp_k),cg_z(temp_z);

    // Generate CG-formatted system object
    CG_SYSTEM<T> cg_system(layout,time,dt);

    // Generate Conjugate Gradients solver object
    CONJUGATE_GRADIENT<T> cg;
    cg.print_residuals=true;
    cg.print_diagnostics=true;
```

```
template<class T>
void SIMULATION_DRIVER<T>::Simulate_Time_Step(const T time,const T dt)
{
    layout.Set_Kinematic_Positions(time,layout.particles.X);
    layout.Set_Kinematic_Velocities(time,layout.particles.V);

    // Construct right-hand-side
    ARRAY<TV> rhs(layout.n);
    layout.Add_Elastic_Forces(layout.particles.X,rhs);
    layout.Add_External_Forces(rhs);
    for(int p=1;p<=layout.n;p++)
        rhs(p)=layout.mass(p)*layout.particles.V(p)+dt*rhs(p);

    // Use previous velocities as initial guess for next velocities
    ARRAY<TV> V_next(layout.particles.V);

    // Temporary vectors, required by Conjugate Gradients
    ARRAY<TV> temp_q(layout.n),temp_s(layout.n),temp_r(layout.n),
        temp_k(layout.n),temp_z(layout.n);

    // Encapsulate all vectors in CG-mandated format
    CG_VECTOR<T> cg_x(V_next),cg_b(rhs),
        cg_q(temp_q),cg_s(temp_s),cg_r(temp_r),cg_k(temp_k),cg_z(temp_z);

    // Generate CG-formatted system object
    CG_SYSTEM<T> cg_system(layout,time,dt);

    // Generate Conjugate Gradients solver object
    CONJUGATE_GRADIENT<T> cg;
    cg.print_residuals=true;
    cg.print_diagnostics=true;
```

```
template<class T>
void SIMULATION_DRIVER<T>::Simulate_Time_Step(const T time,const T dt)
{
    layout.Set_Kinematic_Positions(time,layout.particles.X);
    layout.Set_Kinematic_Velocities(time,layout.particles.V);

    // Construct right-hand-side
    ARRAY<TV> rhs(layout.n);
    layout.Add_Elastic_Forces(layout.particles.X,rhs);
    layout.Add_External_Forces(rhs);
    for(int p=1;p<=layout.n;p++)
        rhs(p)=layout.mass(p)*layout.particles.V(p)+dt*rhs(p);

    // Use previous velocities as initial guess for next velocities
    ARRAY<TV> V_next(layout.particles.V);

    // Temporary vectors, required by Conjugate Gradients
    ARRAY<TV> temp_q(layout.n),temp_s(layout.n),temp_r(layout.n),
        temp_k(layout.n),temp_z(layout.n);

    // Encapsulate all vectors in CG-mandated format
    CG_VECTOR<T> cg_x(V_next),cg_b(rhs),
        cg_q(temp_q),cg_s(temp_s),cg_r(temp_r),cg_k(temp_k),cg_z(temp_z);

    // Generate CG-formatted system object
    CG_SYSTEM<T> cg_system(layout,time,dt);

    // Generate Conjugate Gradients solver object
    CONJUGATE_GRADIENT<T> cg;
    cg.print_residuals=true;
    cg.print_diagnostics=true;
```

```
template<class T>
void SIMULATION_DRIVER<T>::Simulate_Time_Step(const T time,const T dt)
{
    layout.Set_Kinematic_Positions(time,layout.particles.X);
    layout.Set_Kinematic_Velocities(time,layout.particles.V);

    // Construct right-hand-side
    ARRAY<TV> rhs(layout.n);
    layout.Add_Elastic_Forces(layout.particles.X,rhs);
    layout.Add_External_Forces(rhs);
    for(int p=1;p<=layout.n;p++)
        rhs(p)=layout.mass(p)*layout.particles.V(p)+dt*rhs(p);

    // Use previous velocities as initial guess for next velocities
    ARRAY<TV> V_next(layout.particles.V);

    // Temporary vectors, required by Conjugate Gradients
    ARRAY<TV> temp_q(layout.n),temp_s(layout.n),temp_r(layout.n),
        temp_k(layout.n),temp_z(layout.n);

    // Encapsulate all vectors in CG-mandated format
    CG_VECTOR<T> cg_x(V_next),cg_b(rhs),
        cg_q(temp_q),cg_s(temp_s),cg_r(temp_r),cg_k(temp_k),cg_z(temp_z);

    // Generate CG-formatted system object
    CG_SYSTEM<T> cg_system(layout,time,dt);

    // Generate Conjugate Gradients solver object
    CONJUGATE_GRADIENT<T> cg;
    cg.print_residuals=true;
    cg.print_diagnostics=true;
```

```
template<class T>
void SIMULATION_DRIVER<T>::Simulate_Time_Step(const T time,const T dt)
{
    layout.Set_Kinematic_Positions(time,layout.particles.X);
    layout.Set_Kinematic_Velocities(time,layout.particles.V);

    // Construct right-hand-side
    ARRAY<TV> rhs(layout.n);
    layout.Add_Elastic_Forces(layout.particles.X,rhs);
    layout.Add_External_Forces(rhs);
    for(int p=1;p<=layout.n;p++)
        rhs(p)=layout.mass(p)*layout.particles.V(p)+dt*rhs(p);

    // Use previous velocities as initial guess for next velocities
    ARRAY<TV> V_next(layout.particles.V);

    // Temporary vectors, required by Conjugate Gradients
    ARRAY<TV> temp_q(layout.n),temp_s(layout.n),temp_r(layout.n),
        temp_k(layout.n),temp_z(layout.n);

    // Encapsulate all vectors in CG-mandated format
    CG_VECTOR<T> cg_x(V_next),cg_b(rhs),
        cg_q(temp_q),cg_s(temp_s),cg_r(temp_r),cg_k(temp_k),cg_z(temp_z);

    // Generate CG-formatted system object
    CG_SYSTEM<T> cg_system(layout,time,dt);

    // Generate Conjugate Gradients solver object
    CONJUGATE_GRADIENT<T> cg;
    cg.print_residuals=true;
    cg.print_diagnostics=true;
```

```
template<class T>
void SIMULATION_DRIVER<T>::Simulate_Time_Step(const T time,const T dt)
{
    layout.Set_Kinematic_Positions(time,layout.particles.X);
    layout.Set_Kinematic_Velocities(time,layout.particles.V);

    // Construct right-hand-side
    ARRAY<TV> rhs(layout.n);
    layout.Add_Elastic_Forces(layout.particles.X,rhs);
    layout.Add_External_Forces(rhs);
    for(int p=1;p<=layout.n;p++)
        rhs(p)=layout.mass(p)*layout.particles.V(p)+dt*rhs(p);

    // Use previous velocities as initial guess for next velocities
    ARRAY<TV> V_next(layout.particles.V);

    // Temporary vectors, required by Conjugate Gradients
    ARRAY<TV> temp_q(layout.n),temp_s(layout.n),temp_r(layout.n),
        temp_k(layout.n),temp_z(layout.n);

    // Encapsulate all vectors in CG-mandated format
    CG_VECTOR<T> cg_x(V_next),cg_b(rhs),
        cg_q(temp_q),cg_s(temp_s),cg_r(temp_r),cg_k(temp_k),cg_z(temp_z);

    // Generate CG-formatted system object
    CG_SYSTEM<T> cg_system(layout,time,dt);

    // Generate Conjugate Gradients solver object
    CONJUGATE_GRADIENT<T> cg;
    cg.print_residuals=true;
    cg.print_diagnostics=true;
```

```
    // Solve linear system using CG
    cg.Solve(cg_system,
        cg_x,cg_b,cg_q,cg_s,cg_r,cg_k,cg_z,
        1e-6,0,100);

    // Trapezoidal rule for positions
    ARRAY<TV> dX(layout.n);
    dX =(dt/2)*layout.particles.V;
    dX+=(dt/2)*V_next;
    layout.Clear_Values_Of_Kinematic_Particles(dX);

    layout.particles.X+=dX;                // Update particle positions and velocities
    layout.particles.V=V_next;

    layout.Set_Kinematic_Positions(time+dt,layout.particles.X);
    layout.Set_Kinematic_Velocities(time+dt,layout.particles.V);
}
```

```
    // Solve linear system using CG
    cg.Solve(cg_system,
        cg_x,cg_b,cg_q,cg_s,cg_r,cg_k,cg_z,
        1e-6,0,100);

    // Trapezoidal rule for positions
    ARRAY<TV> dX(layout.n);
    dX =(dt/2)*layout.particles.V;
    dX+=(dt/2)*V_next;
    layout.Clear_Values_Of_Kinematic_Particles(dX);

    layout.particles.X+=dX;                // Update particle positions and velocities
    layout.particles.V=V_next;

    layout.Set_Kinematic_Positions(time+dt,layout.particles.X);
    layout.Set_Kinematic_Velocities(time+dt,layout.particles.V);
}
```

```
    // Solve linear system using CG
    cg.Solve(cg_system,
        cg_x,cg_b,cg_q,cg_s,cg_r,cg_k,cg_z,
        1e-6,0,100);

    // Trapezoidal rule for positions
    ARRAY<TV> dX(layout.n);
    dX =(dt/2)*layout.particles.V;
    dX+=(dt/2)*V_next;
    layout.Clear_Values_Of_Kinematic_Particles(dX);

    layout.particles.X+=dX;                 // Update particle positions and velocities
    layout.particles.V=V_next;

    layout.Set_Kinematic_Positions(time+dt,layout.particles.X);
    layout.Set_Kinematic_Velocities(time+dt,layout.particles.V);
}
```