DATm: Diderot's Automated Testing Model

Charisee Chiw, Gordon Kindlmann, and John Reppy

THE UNIVERSITY OF CHICAGO
COMPUTER SCIENCE

Diderot

- Diderot; Domain specific language (DSL) for scivis [PLDI'12]

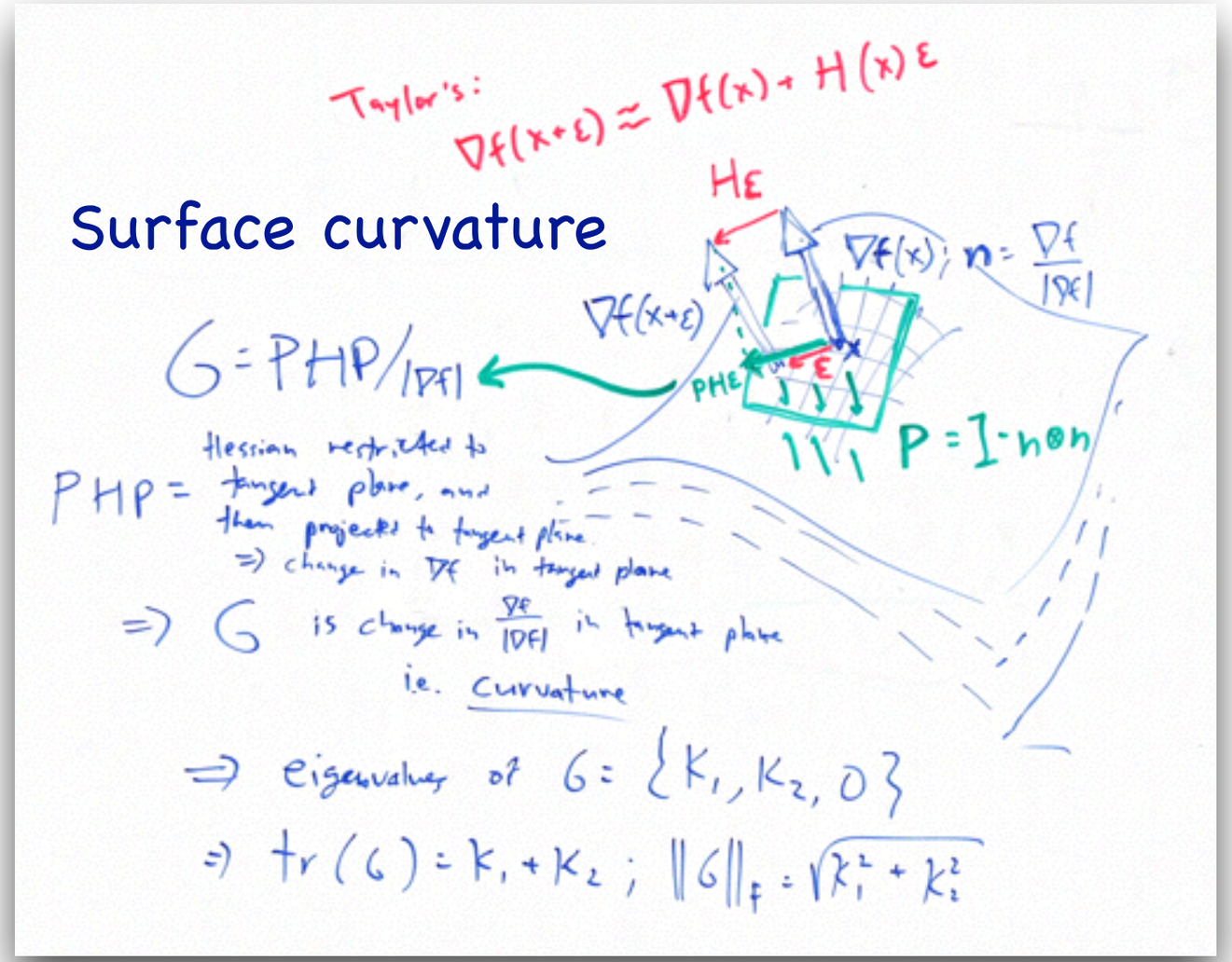
Diderot

- Diderot; Domain specific language (DSL) for scivis [PLDI'12]
- Visually explore data and compute features [VIS'15]

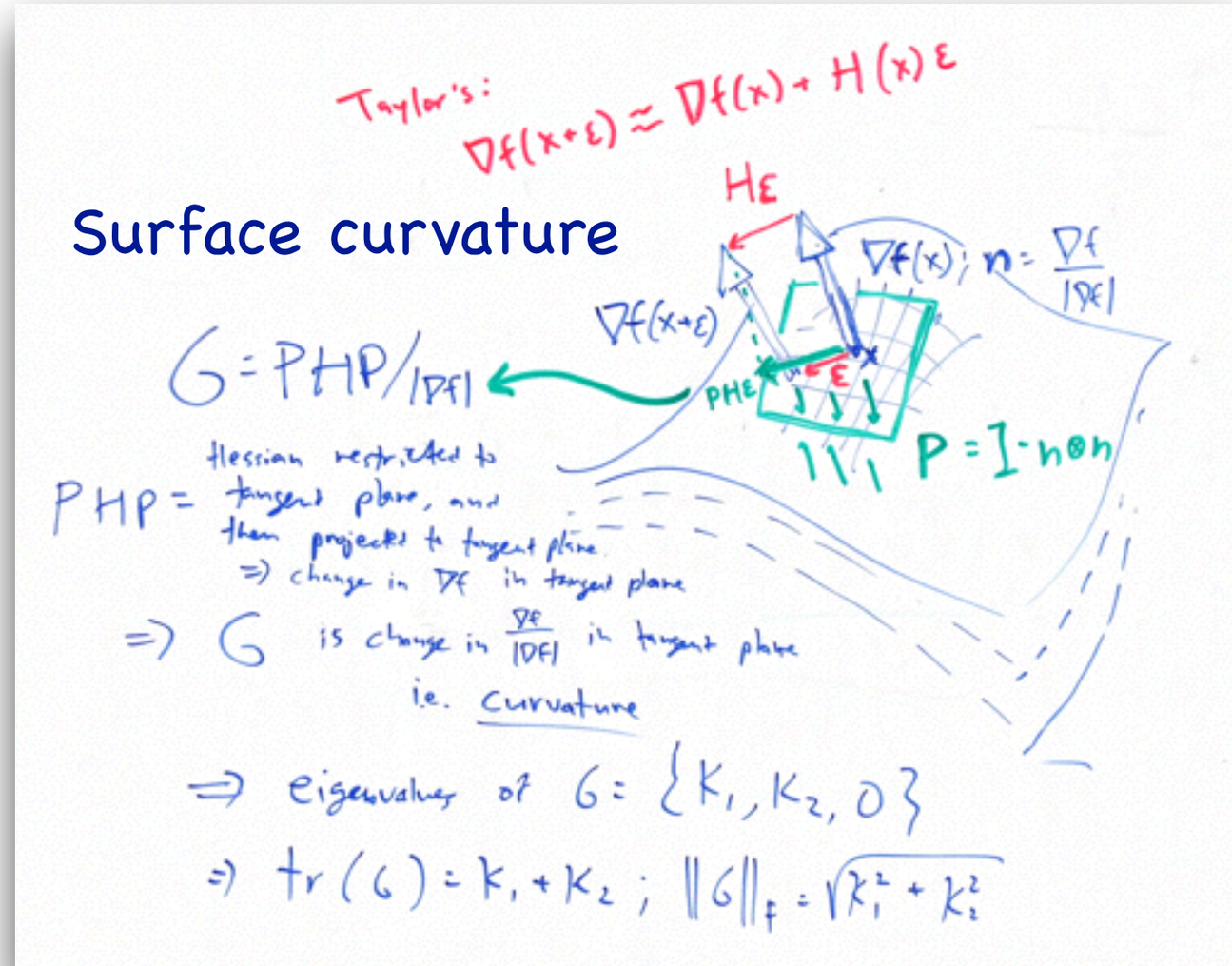
Diderot

- Diderot; Domain specific language (DSL) for scivis [PLDI'12]
- Visually explore data and compute features [VIS'15]
- High-level model of computation with continuous tensor fields

Diderot: from whiteboard to code



Diderot: from whiteboard to code

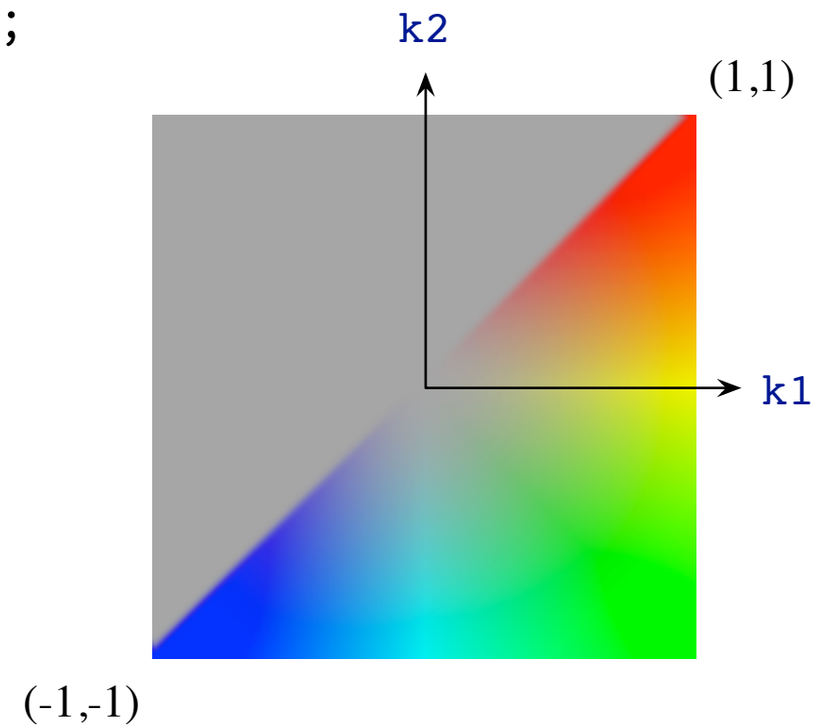


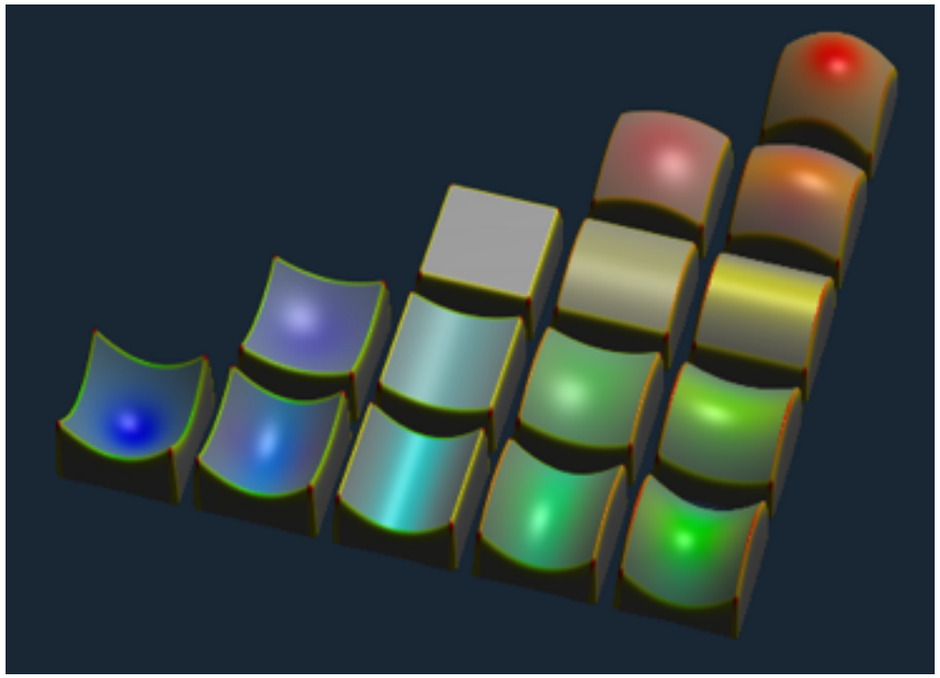
Eases transition of ideas into code

```
vec3 grad = -∇F (pos);
vec3 norm = normalize(grad);
tensor[3,3] H = ∇ ⊗ ∇ F (pos);
tensor[3,3] P = identity[3] - norm⊗norm;
tensor[3,3] G = -(P•H•P)/|grad|;
real disc = sqrt(2.0 * |G|^2 - trace(G)^2);
real k1 = (trace(G) + disc)/2.0;
real k2 = (trace(G) - disc)/2.0;
```

Curvature based volume rendering

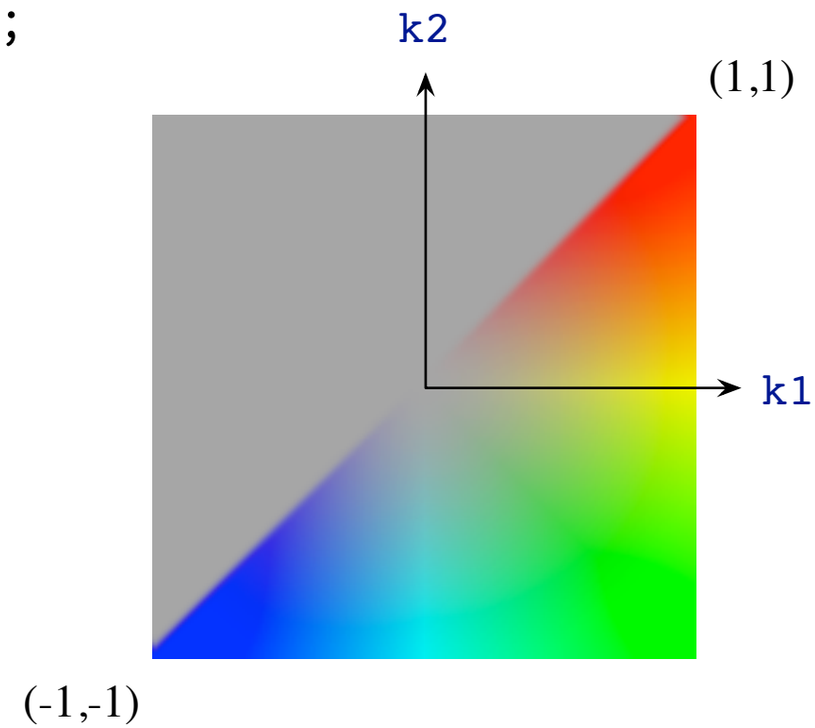
```
field#4(3)[] F = bspln5 ⊛ image("quad-patches.nrrd");
field#0(2)[3] RGB = tent ⊛ image("2d-bow.nrrd");
...
strand RayCast (int ui, int vi) {
  ...
  update
   ...
  vec3 grad = -∇F (pos);
  vec3 norm = normalize(grad);
  tensor[3,3] H = ∇ ⊗ ∇ F (pos);
  tensor[3,3] P = identity[3] - norm⊗norm;
  tensor[3,3] G = -(P•H•P)/|grad|;
  real disc = sqrt(2.0 * |G|^2 - trace(G)^2);
  real k1 = (trace(G) + disc)/2.0;
  real k2 = (trace(G) - disc)/2.0;
  vec3 matRGB = ...RGB([...k1, ...k2]);
  real out = ...
  }
...
}
```

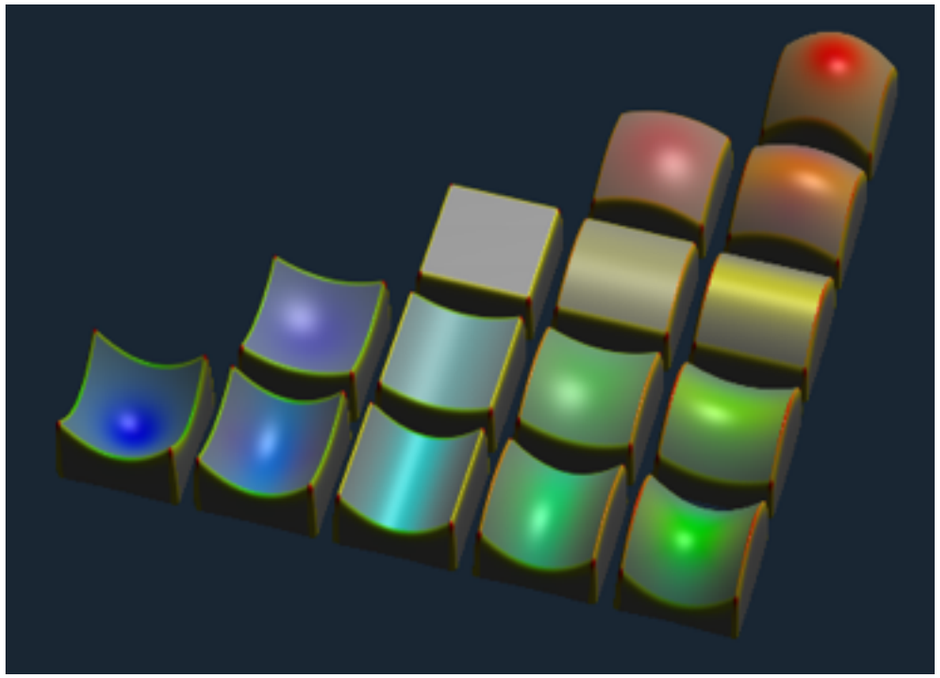




Curvature based volume rendering

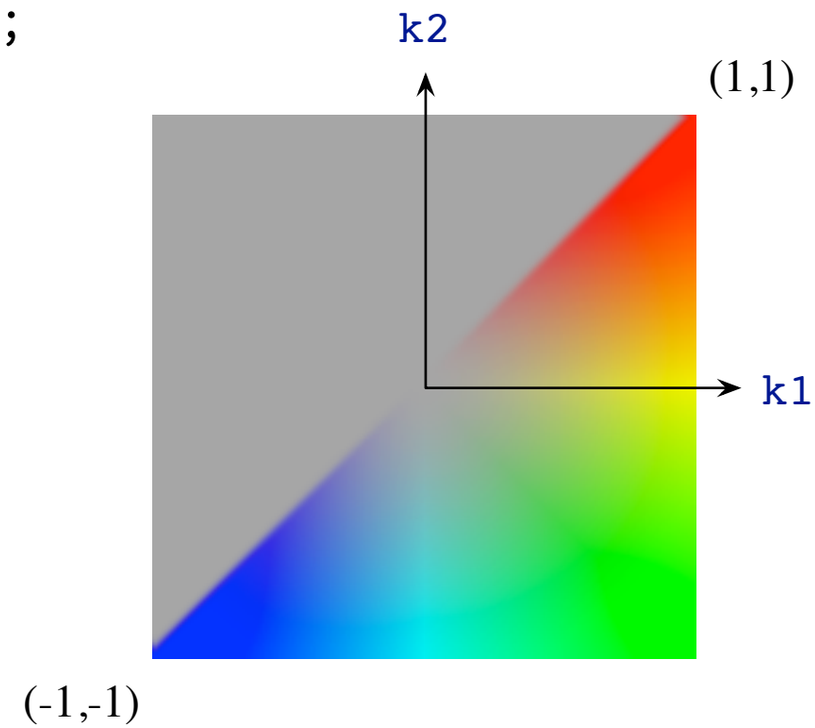
```
field#4(3)[] F = bspln5 ⊛ image("quad-patches.nrrd");
field#0(2)[3] RGB = tent ⊛ image("2d-bow.nrrd");
...
strand RayCast (int ui, int vi) {
  ...
  update
   ...
  vec3 grad = -∇F (pos);
  vec3 norm = normalize(grad);
  tensor[3,3] H = ∇⊗∇ F (pos);
  tensor[3,3] P = identity[3] - norm⊗norm;
  tensor[3,3] G = -(P•H•P)/|grad|;
  real disc = sqrt(2.0 * |G|^2 - trace(G)^2);
  real k1 = (trace(G) + disc)/2.0;
  real k2 = (trace(G) - disc)/2.0;
  vec3 matRGB = ...RGB([...k1, ...k2]);
  real out = ...
  }
...
}
```

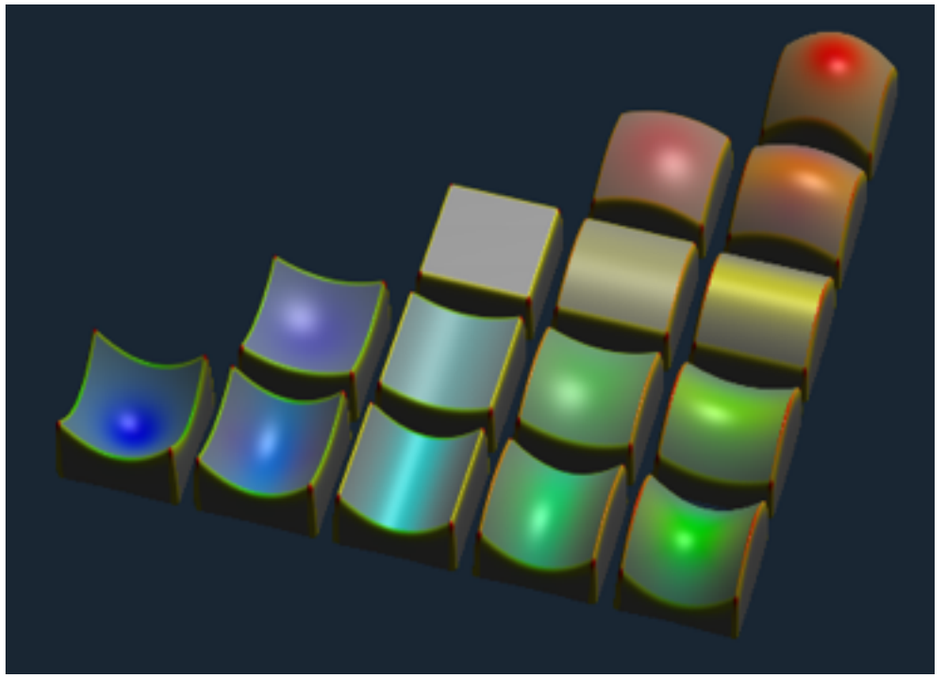




Curvature based volume rendering

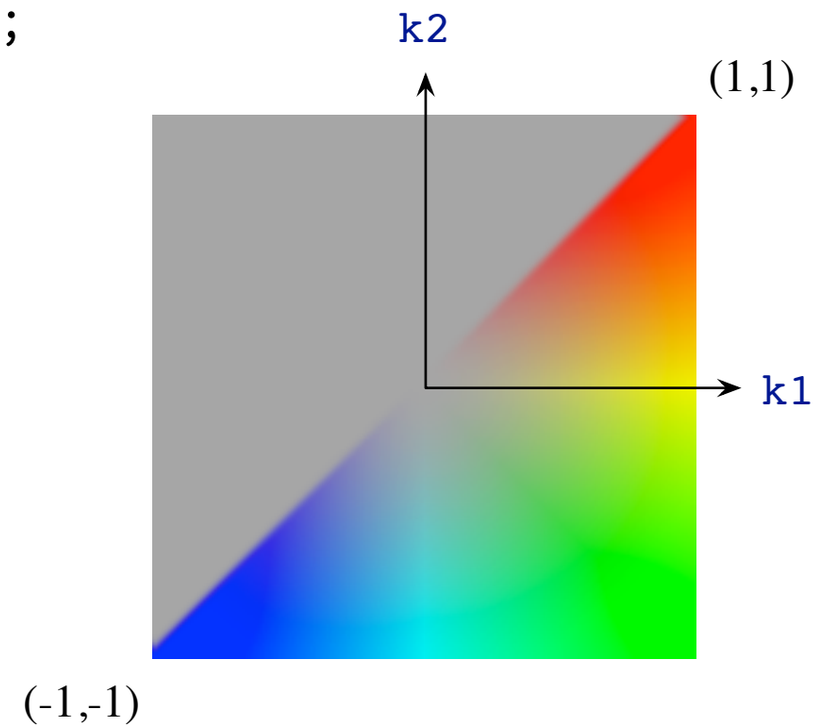
```
field#4(3)[] F = bspln5 ⊛ image("quad-patches.nrrd");
field#0(2)[3] RGB = tent ⊛ image("2d-bow.nrrd");
...
strand RayCast (int ui, int vi) {
  ...
  update
   ...
  vec3 grad = -∇F (pos);
  vec3 norm = normalize(grad);
  tensor[3,3] H = ∇ ⊗ ∇ F (pos);
  tensor[3,3] P = identity[3] - norm⊗norm;
  tensor[3,3] G = -(P•H•P)/|grad|;
  real disc = sqrt(2.0 * |G|^2 - trace(G)^2);
  real k1 = (trace(G) + disc)/2.0;
  real k2 = (trace(G) - disc)/2.0;
  vec3 matRGB = ...RGB([...k1, ...k2]);
  real out = ...
  }
...
}
```

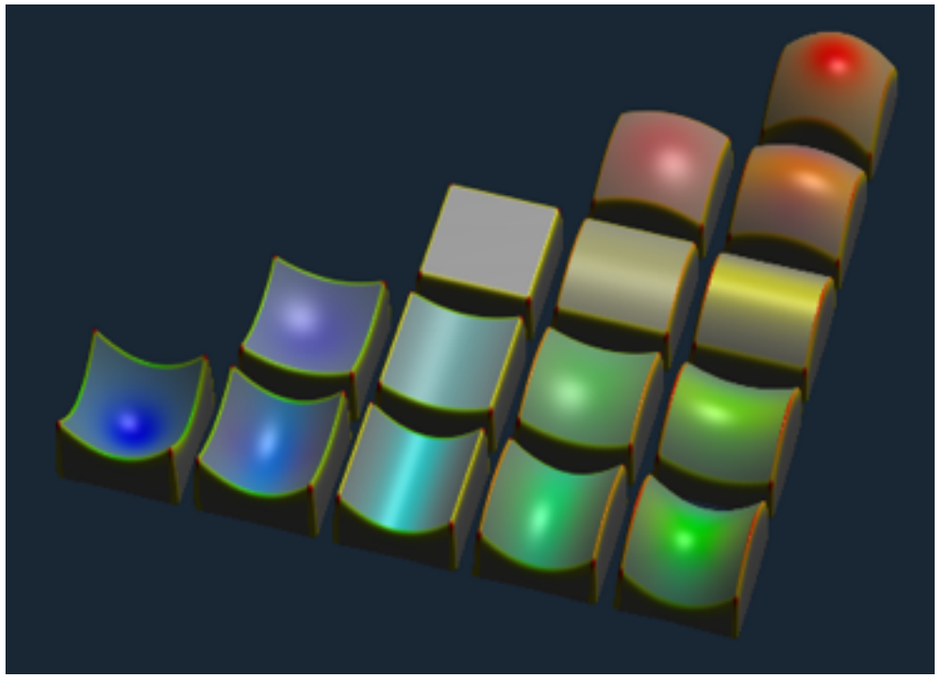




Curvature based volume rendering

```
field#4(3)[] F = bspln5 ⊛ image("quad-patches.nrrd");
field#0(2)[3] RGB = tent ⊛ image("2d-bow.nrrd");
...
strand RayCast (int ui, int vi) {
  ...
  update
   ...
  vec3 grad = -∇F (pos);
  vec3 norm = normalize(grad);
  tensor[3,3] H = ∇⊗∇ F (pos);
  tensor[3,3] P = identity[3] - norm⊗norm;
  tensor[3,3] G = -(P•H•P)/|grad|;
  real disc = sqrt(2.0 * |G|^2 - trace(G)^2);
  real k1 = (trace(G) + disc)/2.0;
  real k2 = (trace(G) - disc)/2.0;
  vec3 matRGB = ...RGB([...k1, ...k2]);
  real out = ...
  }
...
}
```

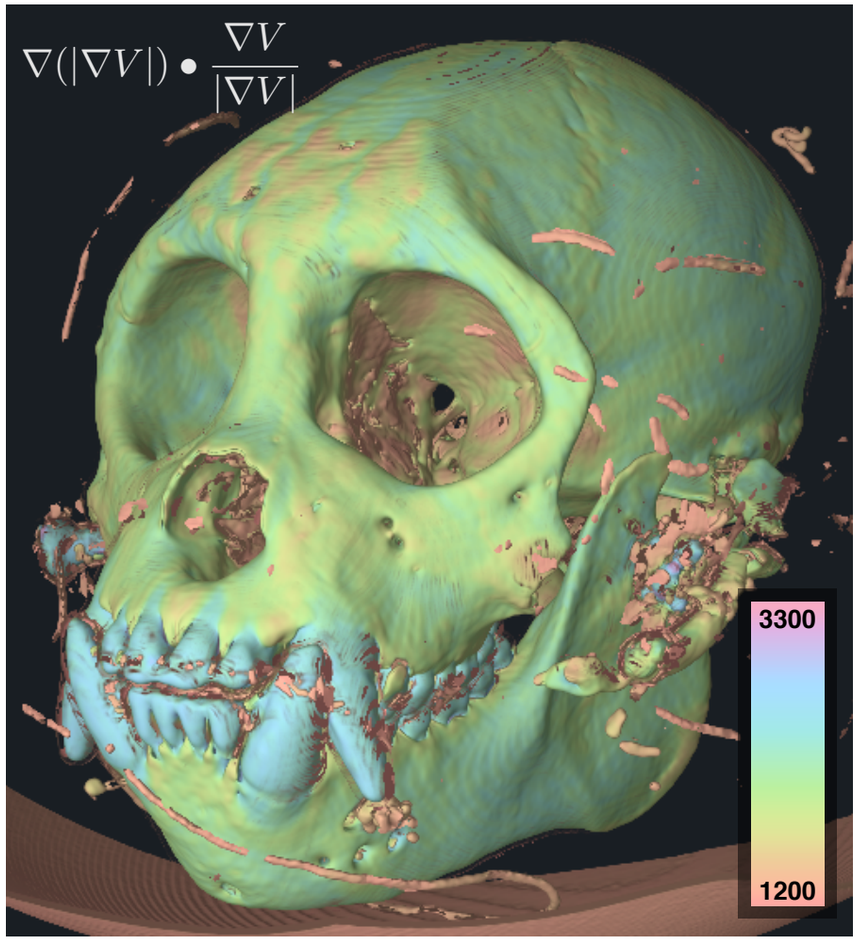




Diderot design and implementation

- Direct use of a high level of math in code

$$\mathbf{field}\,\#2(3)[]\mathrm{C} = -\nabla(|\nabla \mathrm{F}|) \bullet \nabla \mathrm{F}/|\nabla \mathrm{F}|;$$



Diderot design and implementation

- Direct use of a high level of math in code
- EIN IR is used to internally represent tensor and field computations[CPC'16]

e	$::=$	$T_\alpha, A_\alpha, B_\alpha$	Tensor
	$\mid$	F_α, G_α	Field
	$\mid$	δ_{ij}	Kronecker deltas
	$\mid$	$\mathcal{E}_{ij}, \mathcal{E}_{ijk}$	Levi-Civita tensor
	$\mid$	$\sum_\sigma e$	Summation
	$\mid$	$\frac{\partial}{\partial x_\alpha} \diamond e$	Derivative of e
	$\mid$	$e_1 @ e_2$	Probe of a field
	$\mid$	$\mathbf{lift}(e)$	Lift tensor e to a field
	$\mid$	$V_\alpha \circledast H^\beta$	Convolution
	$\mid$	$\sqrt{e}, -e, \exp(e), e^n, \mathrm{sine}(e)$	Unary operators
	$\mid$	$e + e, e - e, \frac{e}{e}, ee$	Binary operators
i, j, k	$\in$	$\textsc{IndexVar}$	Variable index
n	$\in$	$\textsc{ConstVar} = \mathbb{N}$	Constant index
μ	$\in$	$\textsc{IndexVar} \cup \textsc{ConstVar}$	Single index
α, β, γ	$=$	$\bar{\mu}$	Sequence of indices
σ	$=$	$(\mathbb{Z} \times \textsc{IndexVar} \times \mathbb{Z})^*$	index mapping

Diderot design and implementation

- Direct use of a high level of math in code
- EIN IR is used to internally represent tensor and field computations[CPC’16]
- Internally, the IR
  - is dense
  - intricate
  - and goes through complicated transformations

Diderot design and implementation

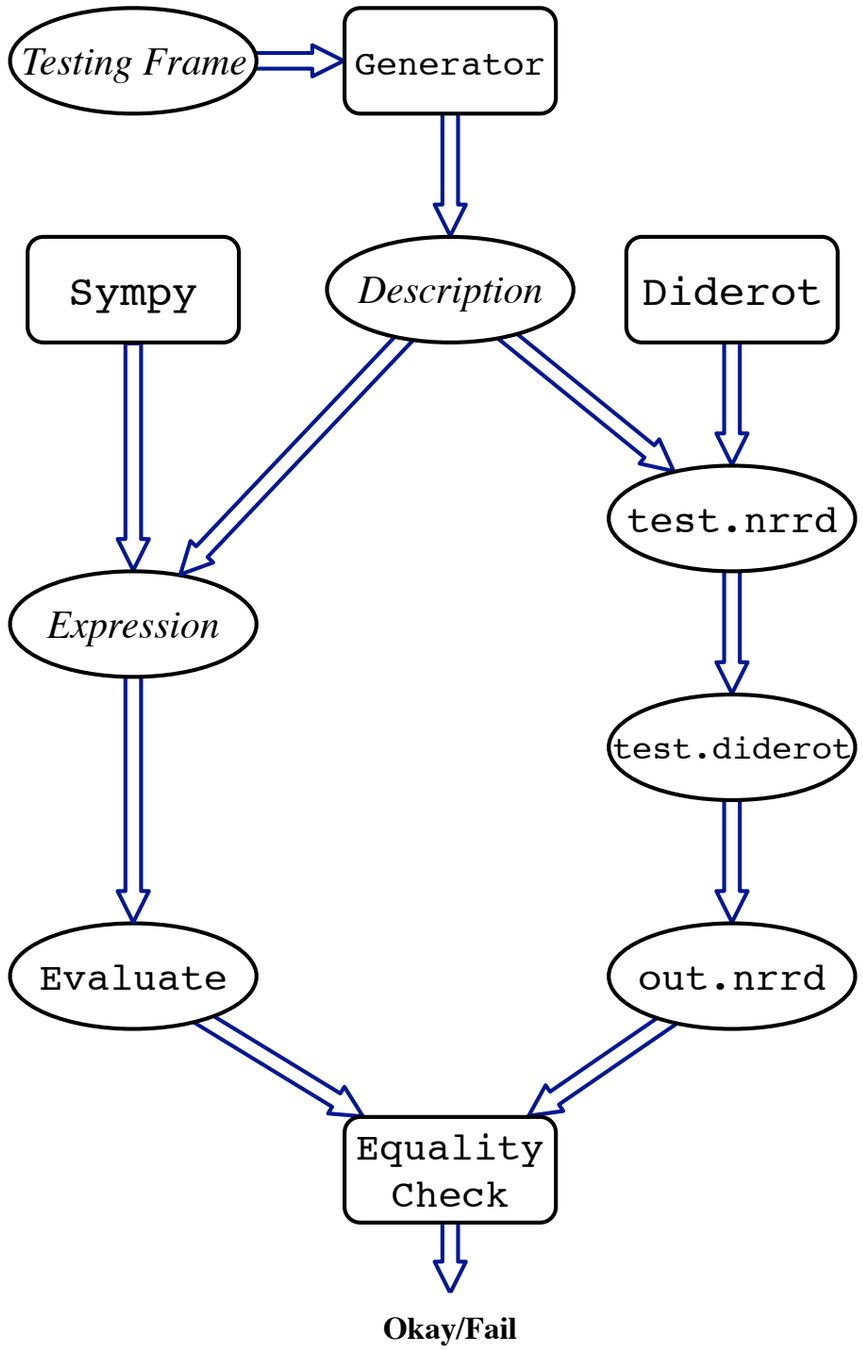
- Direct use of a high level of math in code
- EIN IR is used to internally represent tensor and field computations[CPC’16]
- Internally, the IR
  - is dense
  - intricate
  - and goes through complicated transformations
- Manual testing insufficient

Diderot design and implementation

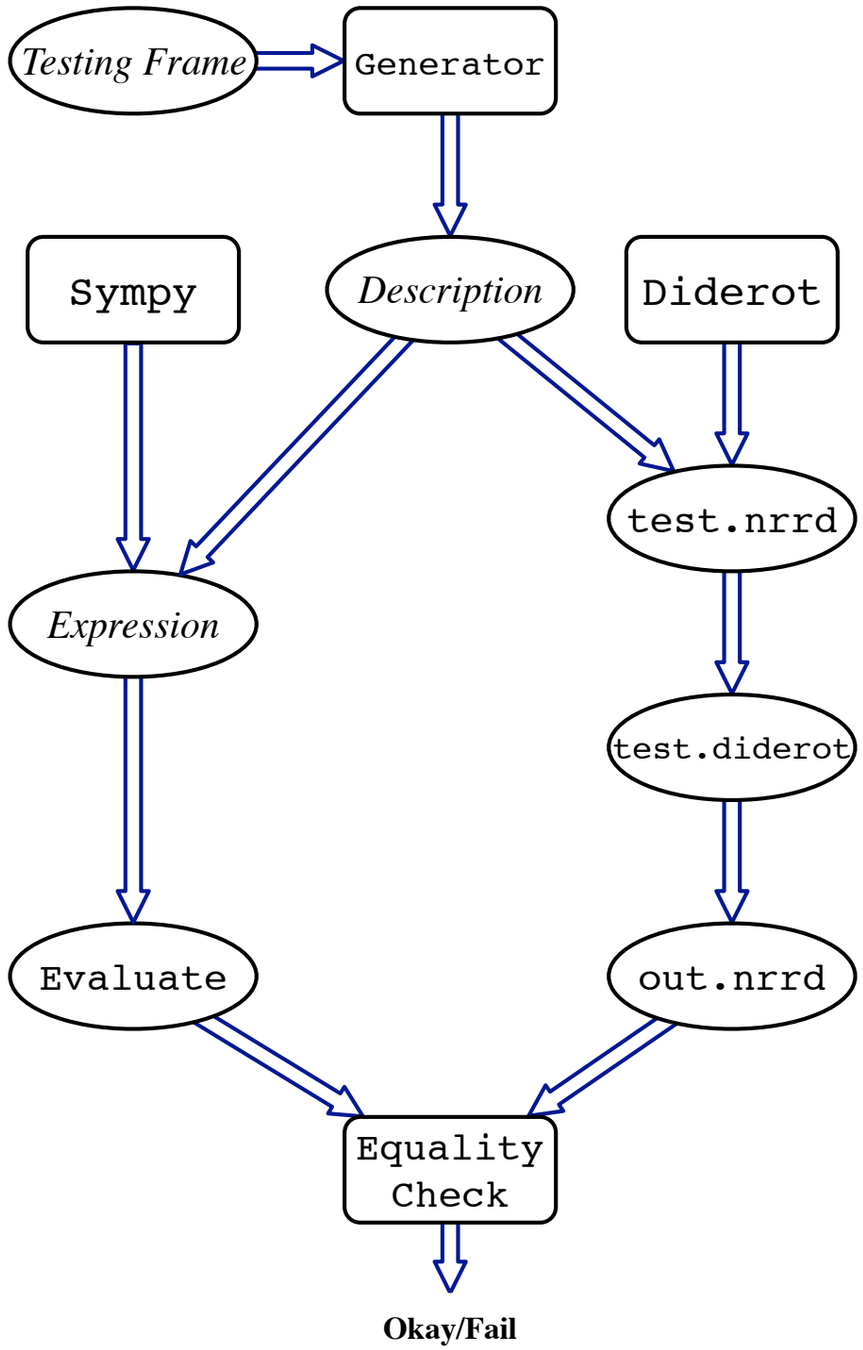
- Direct use of a high level of math in code
- EIN IR is used to internally represent tensor and field computations[CPC’16]
- Internally, the IR
  - is dense
  - intricate
  - and goes through complicated transformations
- Manual testing insufficient
- Adding new operators to the language

Diderot's Automated Testing Model

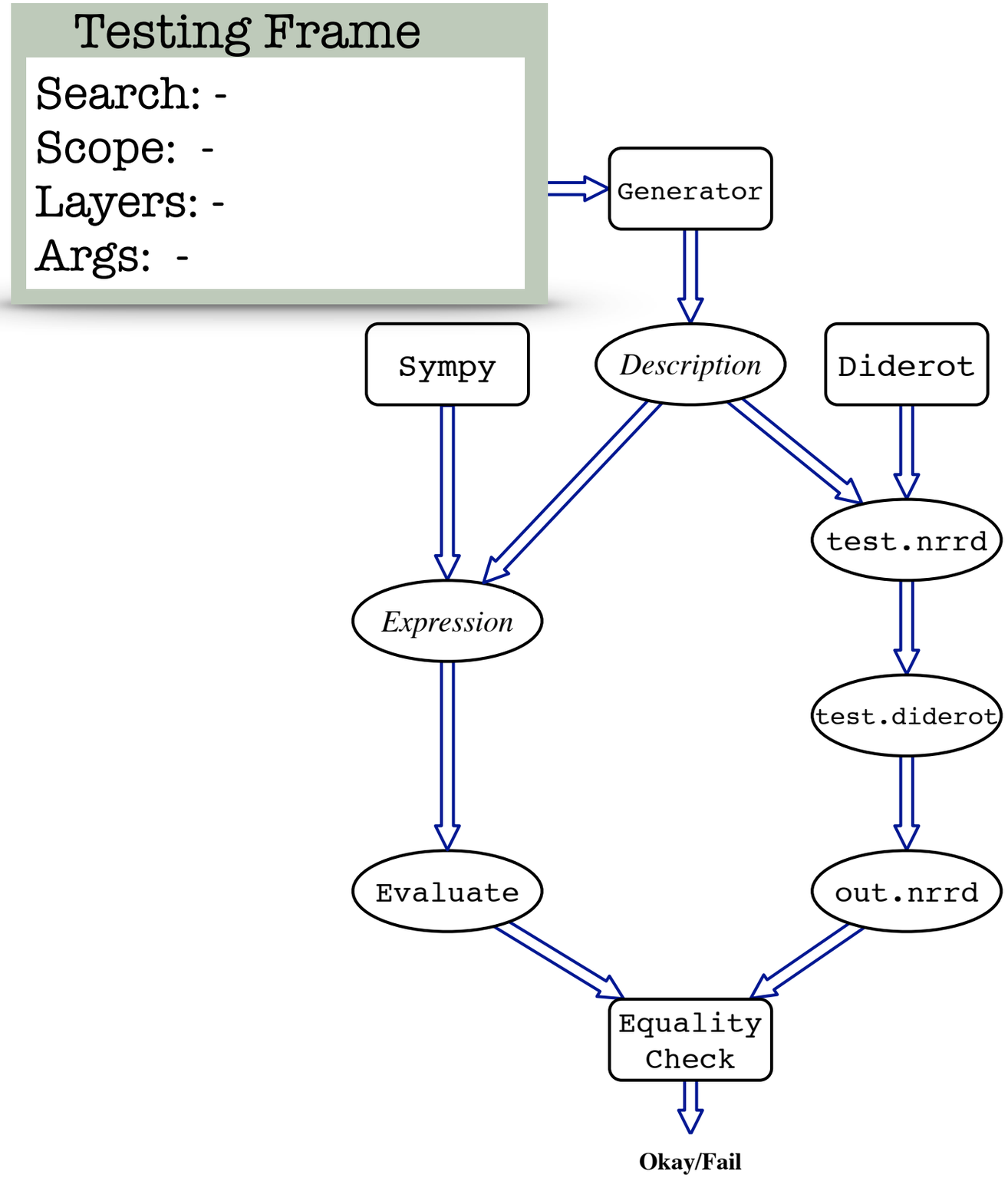
We have developed a test-generation infrastructure for Diderot



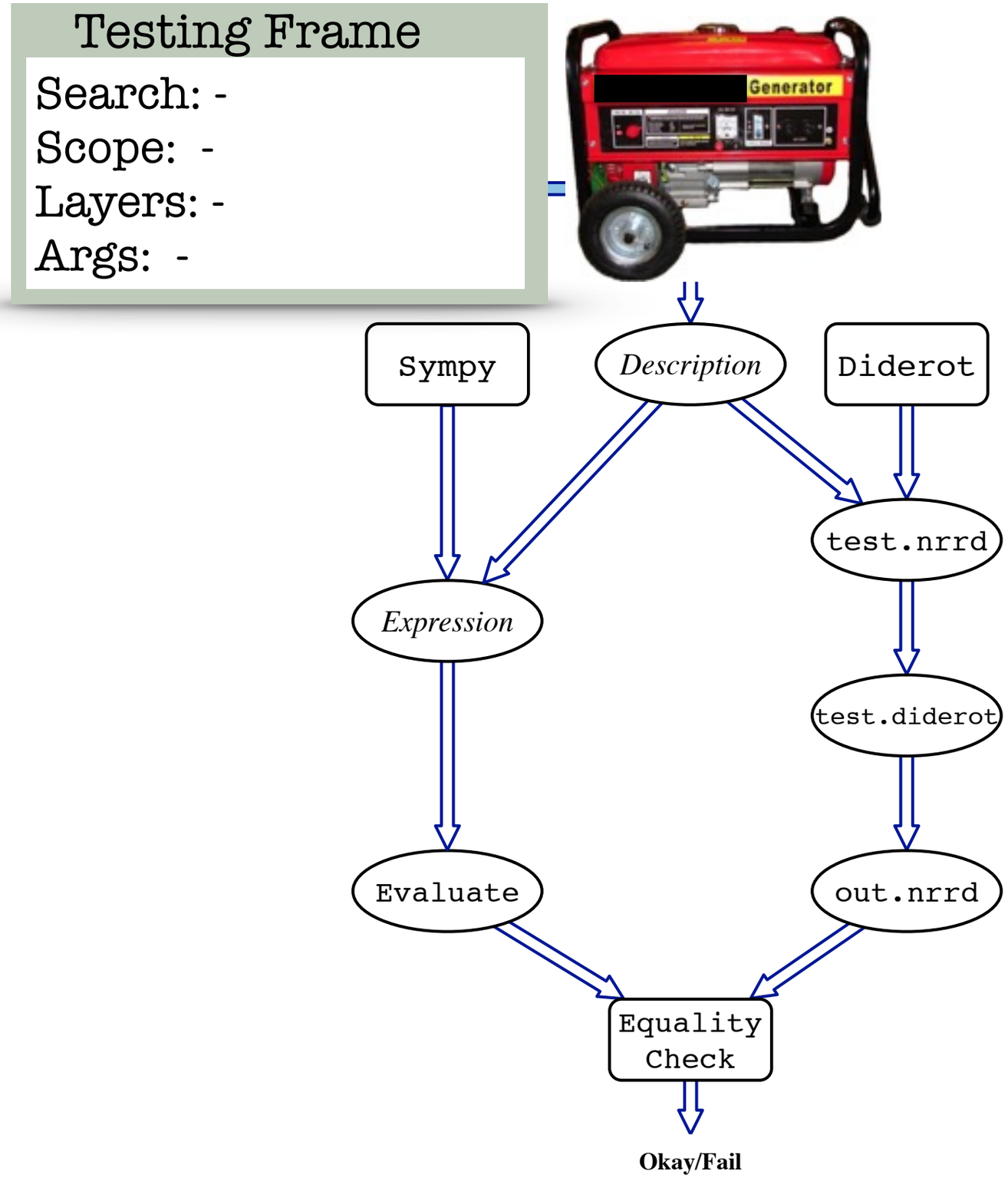
Diderot's Automated Testing Model



Diderot's Automated Testing Model



Diderot's Automated Testing Model



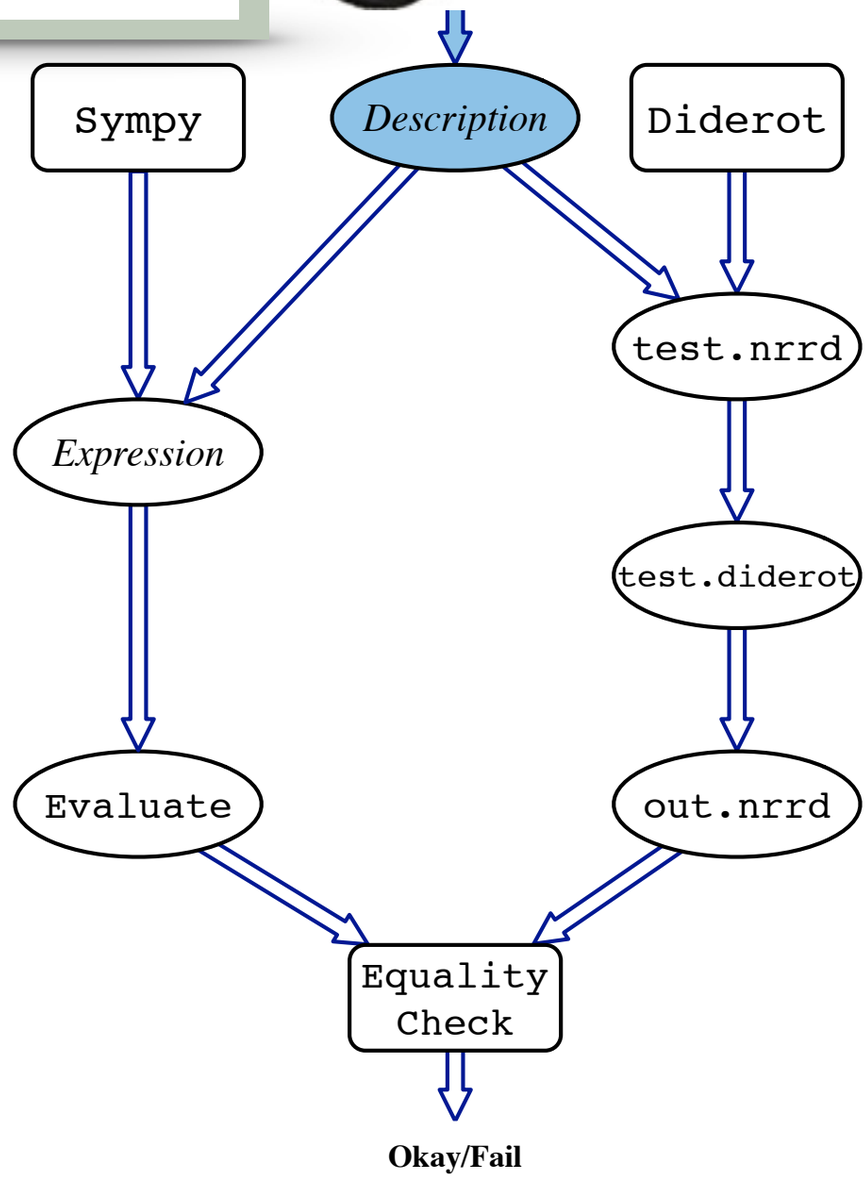
Diderot's Automated Testing Model

Testing Frame

Search: -
Scope: -
Layers: -
Args: -

Details

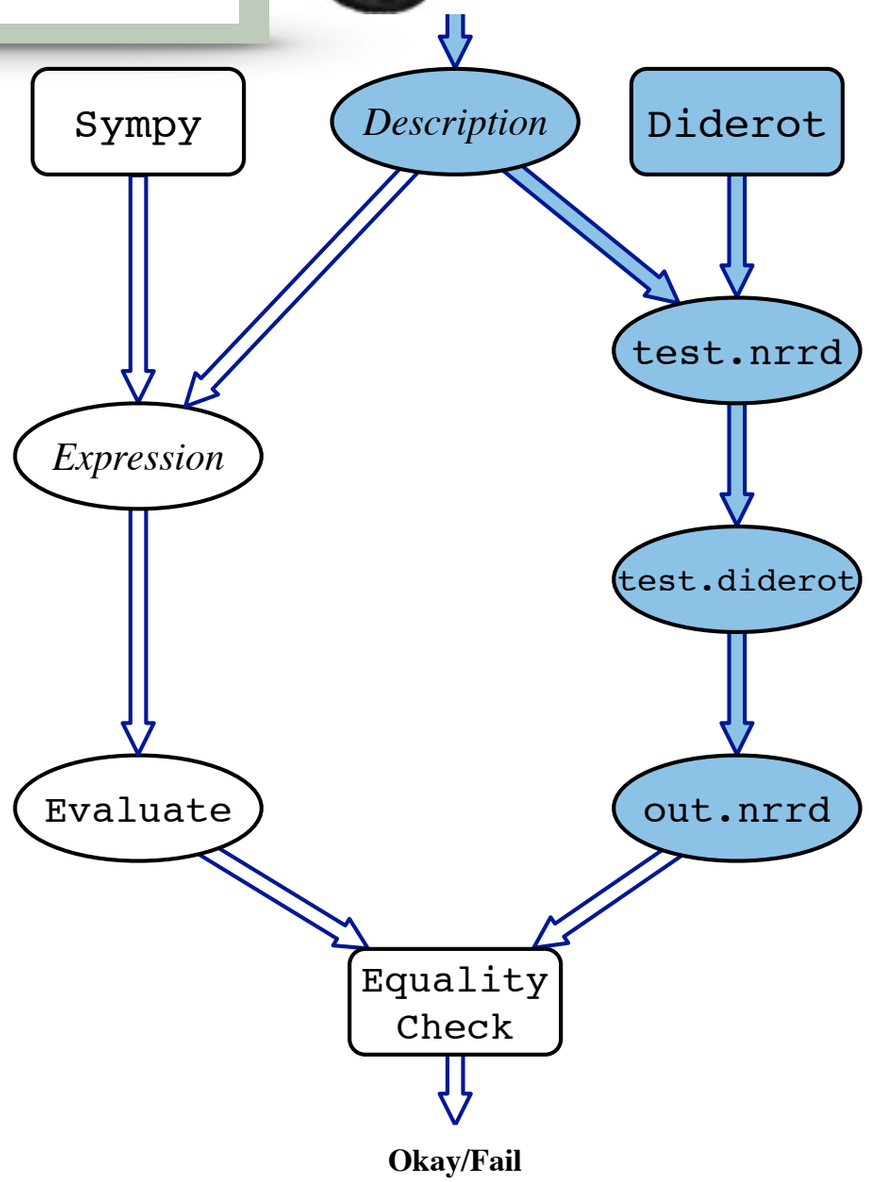
args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F



Diderot's Automated Testing Model

Testing Frame

Search: -
Scope: -
Layers: -
Args: -



Details

args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F

Diderot

```
tensor[3] T;
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
int n;
strand sample (int i) {
 tensor[3,3] out;
 update {
  tensor[3] pos;
   if(i==1) { pos = pos1; }
   ...
   else if(i==n) { pos = posn; }
   out = G(pos);
  }
}initially [sample(i)| i in 0..n];
```

Diderot's Automated Testing Model

Testing Frame

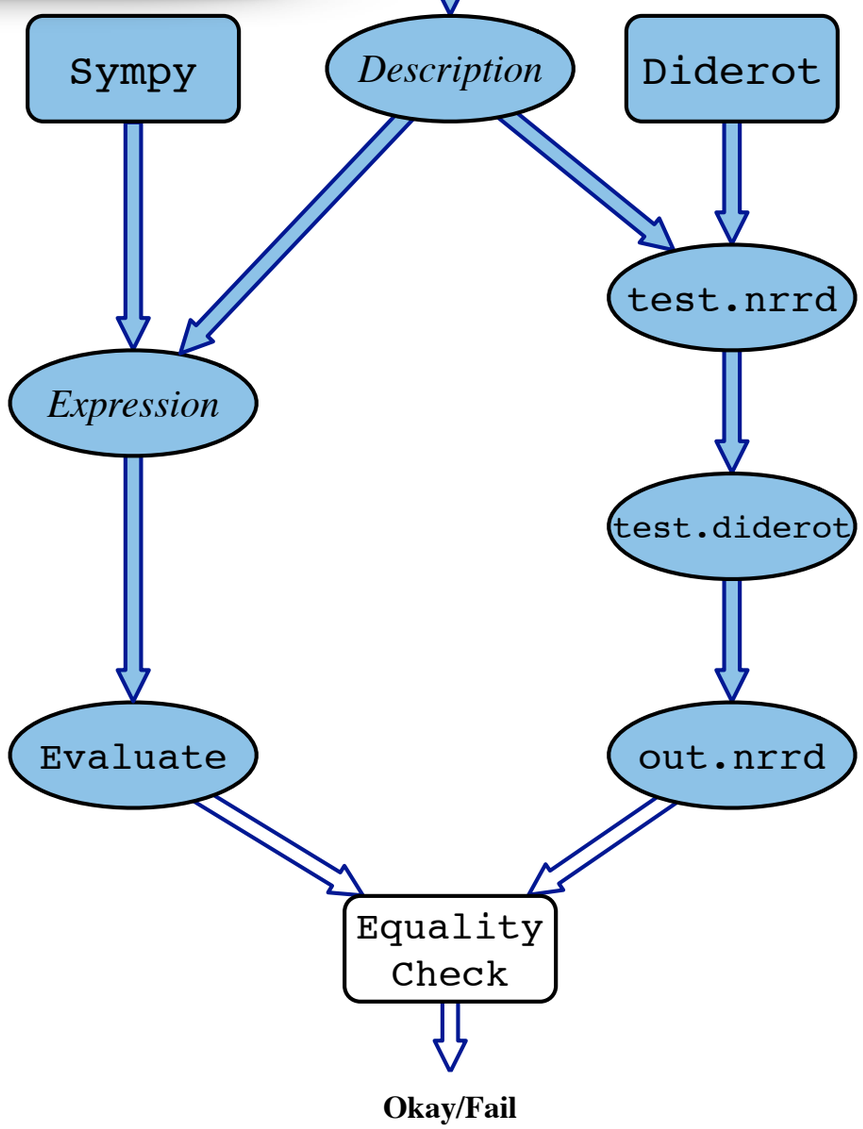
Search: -
Scope: -
Layers: -
Args: -

Details

args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F

Sympy

```
import sympy
x,y,z =symbols('x y z')
T = [3, 9, 11]
F = 2 + 5x+9xy
tmp = deriv(F) # ∇F=[5+9y, 9x, 0]
exp = outer(T, tmp) #T⊗ ∇F
for pos in positions:
  [posx, posy, posz] = pos
  exp = exp.subs(x, posx)
  exp = exp.subs(y, posy)
  exp = exp.subs(z, posz)
  out.append(exp)
return out
```



Diderot

```
tensor[3] T;
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
int n;
strand sample (int i) {
 tensor[3,3] out;
 update {
  tensor[3] pos;
   if(i==1) { pos = pos1; }
   ...
   else if(i==n) { pos = posn; }
   out = G(pos);
  }
}initially [sample(i)| i in 0..n];
```

Diderot's Automated Testing Model

Testing Frame

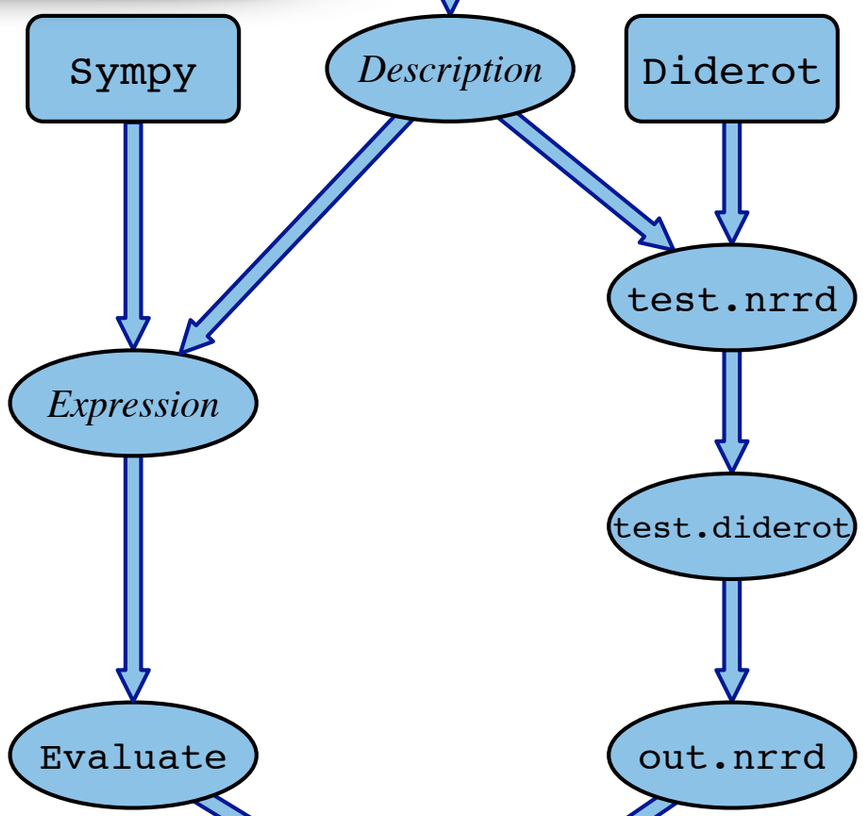
Search: -
Scope: -
Layers: -
Args: -

Details

args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F

Sympy

```
import sympy
x,y,z =symbols('x y z')
T = [3, 9, 11]
F = 2 + 5x+9xy
tmp = deriv(F) # ∇F=[5+9y, 9x, 0]
exp = outer(T, tmp) #T⊗ ∇F
for pos in positions:
  [posx, posy, posz] = pos
  exp = exp.subs(x, posx)
  exp = exp.subs(y, posy)
  exp = exp.subs(z, posz)
  out.append(exp)
return out
```



Diderot

```
tensor[3] T;
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
int n;
strand sample (int i) {
 tensor[3,3] out;
 update {
  tensor[3] pos;
   if(i==1) { pos = pos1; }
   ...
   else if(i==n) { pos = posn; }
   out = G(pos);
  }
}initially [sample(i)| i in 0..n];
```

- ☐ OK
- ☐ Type error
- ☐ Compiler error
- ☐ Numerical error

Infrastructure

- Represents Diderot language

τ	$::=$	$\textbf{tensor}[\varsigma]$	tensor with shape ς
		$\textbf{field}\#k(d)[\kappa]$	tensor field with continuity k, shape κ, and dimension d $(1 \leq d \leq 3)$
		$\textbf{image}(d)[\kappa]$	image data with shape κ, and dimension d $(1 \leq d \leq 3)$
γ	$::=$	$\tau \mid \textbf{field}\#k(d)[\varsigma]$	broader range of shape
d_n	$::=$	$2 \mid 3$	dimension
v_n	$::=$	$2 \mid 3 \mid 4$	extended dimension
ς	$::=$	nil'$\mid v_1, \cdots, v_n$	$n \leq 3$ Tensor shape
κ	$::=$	nil'$\mid v_1 \mid d_1, d_1$	Field shape
erators	$::=$	$-$, $\lVert \cdot \rVert$, @, $\sqrt{}$, ∇, $\nabla\otimes$, $\nabla\times$, $\nabla\bullet$, inverse normalize, trace, transpose	
	$\mid$	$+$, $-$, $*$, $/$, $\bullet$, $:$, $\times$, $\otimes$, [:,0], [1,:], [0], [1], [:,1,:], [1,0,:],	
	$\mid$	det, sin, cos, tan, arccos, arcsin, arctan, $\circledast$, modulate, ...	
kernel	$::=$	c4hexic, tent, ctmr, bspln3,	

Infrastructure

- Represents Diderot language

Operator | Kernel | Type

τ	$::=$	**tensor**$[\varsigma]$	tensor with shape ς
		field$\#k(d)[\kappa]$	tensor field with continuity k, shape κ, and dimension d $(1 \leq d \leq 3)$
		image$(d)[\kappa]$	image data with shape κ, and dimension d $(1 \leq d \leq 3)$
γ	$::=$	$\tau \mid$ **field**$\#k(d)[\varsigma]$	broader range of shape
d_n	$::=$	$2 \mid 3$	dimension
v_n	$::=$	$2 \mid 3 \mid 4$	extended dimension
ς	$::=$	nil'$\mid v_1, \cdots, v_n$	$n \leq 3$ Tensor shape
κ	$::=$	nil'$\mid v_1 \mid d_1, d_1$	Field shape
erators	$::=$	$-$, $\lVert \cdot \rVert$, @, $\sqrt{}$, ∇, $\nabla\otimes$, $\nabla\times$, $\nabla\bullet$, inverse normalize, trace, transpose	
	$\mid$	$+$, $-$, $*$, $/$, $\bullet$, $:$, $\times$, $\otimes$, [:,0], [1,:], [0], [1], [:,1,:], [1,0,:],	
	$\mid$	det, sin, cos, tan, arccos, arcsin, arctan, $\circledast$, modulate, ...	
kernel	$::=$	c4hexic, tent, ctmr, bspln3,	

Operators
- -out
- -placement
- -arity
- -limit

Kernels
- -str
- -continuity
- -order

Types
- -dim
- -shape
- -k

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: -
Layers: -
Args: -

Infrastructure

- Represents Diderot language

Operator | Kernel | Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: -
Layers: -
Args: -

variable	description
coeff_style	The coefficients attribute indicate the polynomial order of synthetic data. The data can be linear (x), quadratic (x^2), or cubic (x^3).
samples	A diderot program is used to create a nrrd file to represent a tensor field. The program evaluates a polynomial expression at a number of samples.
space	Input to the Diderot programs include the number of samples to take and if we randomize the "shear" and "angle" attributes in the Diderot program. Using fewer samples and changing the orientation of the samples creates a stronger field reconstruction test.
num_pos	Generated test programs probe a tensor field at a set number of positions.
krn	The test program uses a specified reconstruction kernel
layer	A computation is made up of a set number of nested operators, which can range from one to three.
in_tys, *rst_tys*	Limitations to argument types and result types, such as only tensor fields.
random_limit	We support the random or exhaustive exploration of test cases. The developer specifies the probability of a single test program being executed, with exhaustive being 100%.

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: -
Layers: -
Args: -

- Test case

args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F

Infrastructure

- Represents Diderot language

Operator | Kernel | Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: -
Layers: -
Args: -

- Test case

args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F

Computation: $T \otimes \nabla$ F

Test Description

Argument

name, Type, Kernel, data

Operator | Operator

Argument

name, Type, Kernel, data

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: -
Layers: -
Args: -

- Test case

args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F

- Generate test cases

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: -
Layers: -
Args: -

- Test case

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

- Generate test cases

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

```
args: vec2 A and 2-d scalar H
operators: A ⊗(∇ H)
```

```
args: vec2 A and 2-d vector field V
operators: A ⊗ V
```

```
args: scalar field F
operators: inv(∇ ⊗ ∇F)
```

```
args: vector field V and 2-d scalar H
operators: V ⊗(∇ H)
```

```
args: vec3 B and 3-d vector field G
operators: B •(∇× G)
```

```
args: vec3 B and 3-d vector field G
operators: B • G
```

```
args: vector field U and 3-d scalar F
operators: U ⊗(∇ F)
```

```
args: matrix M and 3-d vector field G
operators: det(M) ∗ G
```

```
args: vector A, vector B and 3-d vector field G
operators: (A-B)• G
```

```
args: second-order tensor field N and matrix M
operators: inv(N) :M
```

```
args: vec2 A and vec2 B
operators: A ⊗ B
```

```
args: vec2 A and 2-d vector field V
operators: A • V
```

```
args: matrix M and 3-d scalar field F
operators: ∇ ⊗ ∇ F • M
```

```
args: scalar field F
operators: tranpose(∇ ⊗ ∇F)
```

```
args: matrix M and 3-d vector field G
operators: M • G
```

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: -
Layers: 1
Args: -

- Test case

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

- Generate test cases

```
args: vec2 A and 2-d vector field V
operators: A ⊗ V
```

```
args: vec3 B and 3-d vector field G
operators: B • G
```

```
args: vec2 A and vec2 B
operators: A ⊗ B
```

```
args: vec2 A and 2-d vector field V
operators: A • V
```

```
args: matrix M and 3-d vector field G
operators: M • G
```

advantage: smaller number of tests

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: -
Layers: -
Args: tensor fields

- Test case

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

- Generate test cases

```
args: scalar field F
operators: inv(∇ ⊗ ∇F)
```

```
args: vector field V and 2-d scalar H
operators: V ⊗(∇ H)
```

```
args: vector field U and 3-d scalar F
operators: U ⊗(∇ F)
```

```
args: second-order tensor field N and matrix M
operators: inv(N) :M
```

```
args: scalar field F
operators: tranpose(∇ ⊗ ∇F)
```

advantage: focus on tensor fields

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: $\bullet$ inner product
Layers: -
Args: -

- Test case

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

- Generate test cases

```
args: vec3 B and 3-d vector field G
operators: B •(∇× G)
```

```
args: vec3 B and 3-d vector field G
operators: B • G
```

```
args: vector A, vector B and 3-d vector field G
operators: (A-B)• G
```

```
args: vec2 A and 2-d vector field V
operators: A • V
```

```
args: matrix M and 3-d scalar field F
operators: ∇ ⊗ ∇ F • M
```

```
args: matrix M and 3-d vector field G
operators: M • G
```

advantage: testing new operator

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: - $e \otimes (\nabla e)$
Layers: -
Args: -

- Test case

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

- Generate test cases

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

```
args: vec2 A and 2-d scalar H
operators: A ⊗(∇ H)
```

```
args: vector field V and 2-d scalar H
operators: V ⊗(∇ H)
```

```
args: vector field U and 3-d scalar F
operators: U ⊗(∇ F)
```

advantage: targetted testing

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

- Frame initializes testing variables

Testing Frame

Search: -
Scope: - $e \otimes (\nabla e)$
Layers: -
Args: -

- Test case

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

- Generate test cases

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

Infrastructure

- Represents Diderot language

Operator	Kernel	Type

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

- Frame initializes testing variables

Testing Frame

Search: -
Scope: - $e \otimes (\nabla e)$
Layers: -
Args: -

- Test case

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

- Generate test cases

Single test case

Test Description

args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F

Single test case

Test Description

args: A vec3 T and a 3-d scalar field F
operators: $T \otimes \nabla$ F

Python code

```
import sympy
x,y,z =symbols('x y z')
T = [3, 9, 11]
F = 2 + 5x+9xy
```

Single test case

Test Description

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

Python code

```
import sympy
x,y,z =symbols('x y z')
T = [3, 9, 11]
F = 2 + 5x+9xy
```

Diderot program

```
tensor[3] T;
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
```

Single test case

Test Description

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

Python code

```
import sympy
x,y,z =symbols('x y z')
T = [3, 9, 11]
F = 2 + 5x+9xy
```

Diderot program

```
tensor[3] T=[3, 9, 11];
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
int n;
strand sample (int i) {
 tensor[3,3] out;
 update {
  tensor[3] pos;
   if(i==1) { pos = pos_1; }
   ...
   else if(i==n) { pos = pos_n; }
   out = G(pos);
  }
}initially [sample(i)| i in 0..n];
```

Single test case

Test Description

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

Python code

```
import sympy
x,y,z =symbols('x y z')
T = [3, 9, 11]
F = 2 + 5x+9xy
tmp = deriv(F) # ∇F=[5+9y, 9x, 0]
exp = outer(T, tmp) #T⊗ ∇F
for pos in positions:
  [posx, posy, posz] = pos
  exp = exp.subs(x, posx)
  exp = exp.subs(y, posy)
  exp = exp.subs(z, posz)
  out.append(exp)
return out
```

Diderot program

```
tensor[3] '=[3, 9, 11];
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
int n;
strand sample (int i) {
 tensor[3,3] out;
 update {
  tensor[3] pos;
   if(i==1) { pos = pos1; }
   ...
   else if(i==n) { pos = posn; }
   out = G(pos);
  }
}initially [sample(i)| i in 0..n];
```

Single test case

Test Description

```
args: A vec3 T and a 3-d scalar field F
operators: T⊗∇ F
```

Python code

```
import sympy
x,y,z =symbols('x y z')
T = [3, 9, 11]
F = 2 + 5x+9xy
tmp = deriv(F) # ∇F=[5+9y, 9x, 0]
exp = outer(T, tmp) #T⊗ ∇F
for pos in positions:
  [posx, posy, posz] = pos
  exp = exp.subs(x, posx)
  exp = exp.subs(y, posy)
  exp = exp.subs(z, posz)
  out.append(exp)
return out
```

Diderot program

```
tensor[3] '=[3, 9, 11];
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
int n;
strand sample (int i) {
 tensor[3,3] out;
 update {
  tensor[3] pos;
   if(i==1) { pos = pos1; }
   ...
   else if(i==n) { pos = posn; }
   out = G(pos);
  }
}initially [sample(i)| i in 0..n];
```

- ☐ OK
- ☐ Type error
- ☐ Compiler error
- ☐ Numerical error

Compare Diderot with analytically derived solution

Enable other types of testing

Enable other types of testing

- MIP program in Diderot

```
field#k(3)[3] F1 = "V1.nrrd" ⊛ h;
field#k(3)[3] F2 = "V2.nrrd" ⊛ h;
field#k(3)[] G= trace(F1 ⊗ F2);
tensor[3] pos;
int res;
strand raycast (real ui, real vi)
   output real out;
   update {
     ...
     out = max (out,G(pos));
   }
} initially [raycast(ui,vi)|ui in 0 ..res-1, vi in 0..res-1];
```

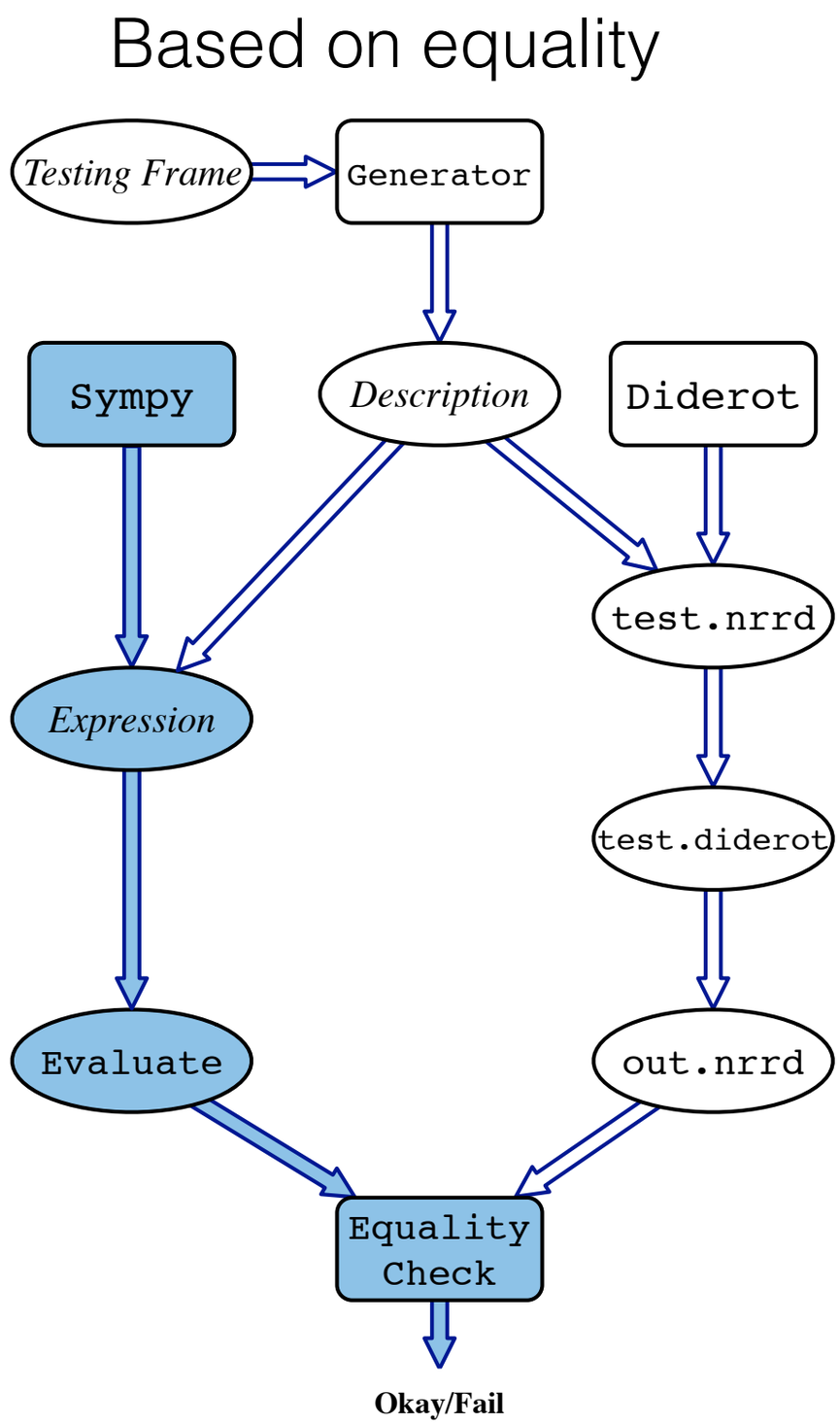
Enable other types of testing

- MIP program in Diderot

```
field#k(3)[3] F1 = "V1.nrrd" ⊛ h;
field#k(3)[3] F2 = "V2.nrrd" ⊛ h;
field#k(3)[] G= trace(F1 ⊗ F2);
tensor[3] pos;
int res;
strand raycast (real ui, real vi)
   output real out;
   update {
     ...
     out = max (out,G(pos));
   }
} initially [raycast(ui,vi)|ui in 0 ..res-1, vi in 0..res-1];
```

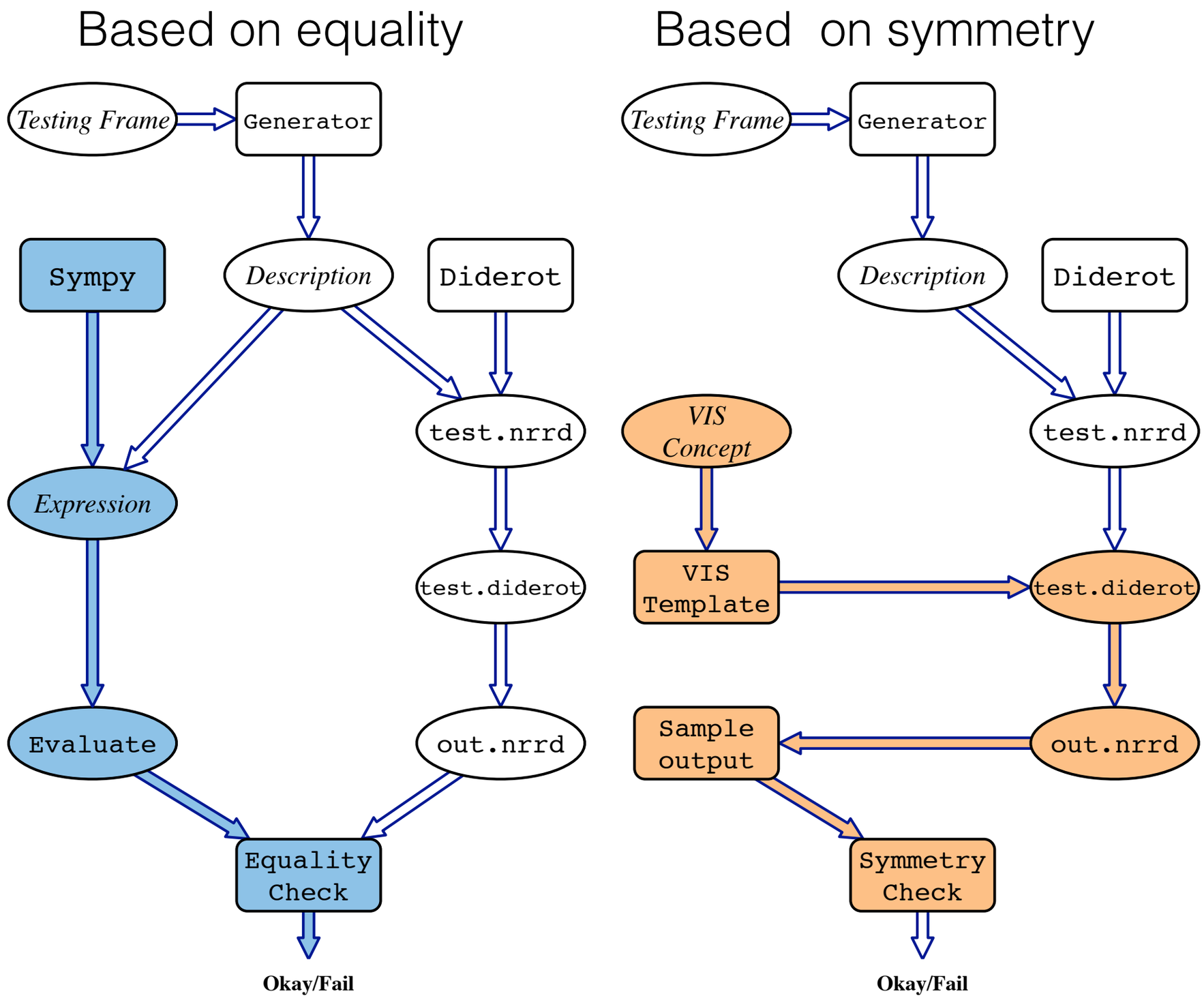
Enable other types of testing

- MIP program in Diderot
- goal: automated testing for visualization programs

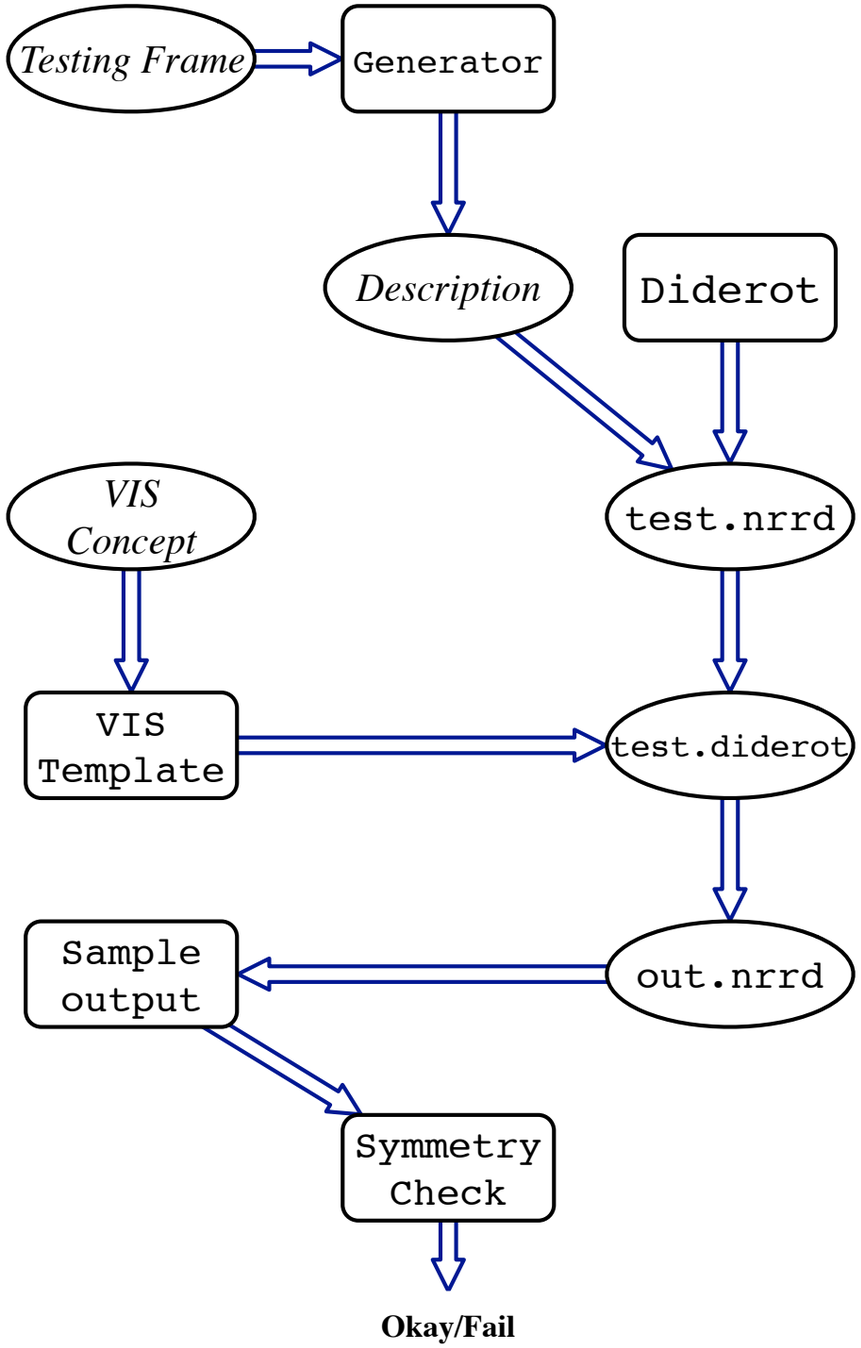


Enable other types of testing

- MIP program in Diderot
- goal: automated testing for visualization programs
- VATm: visualization automated testing model

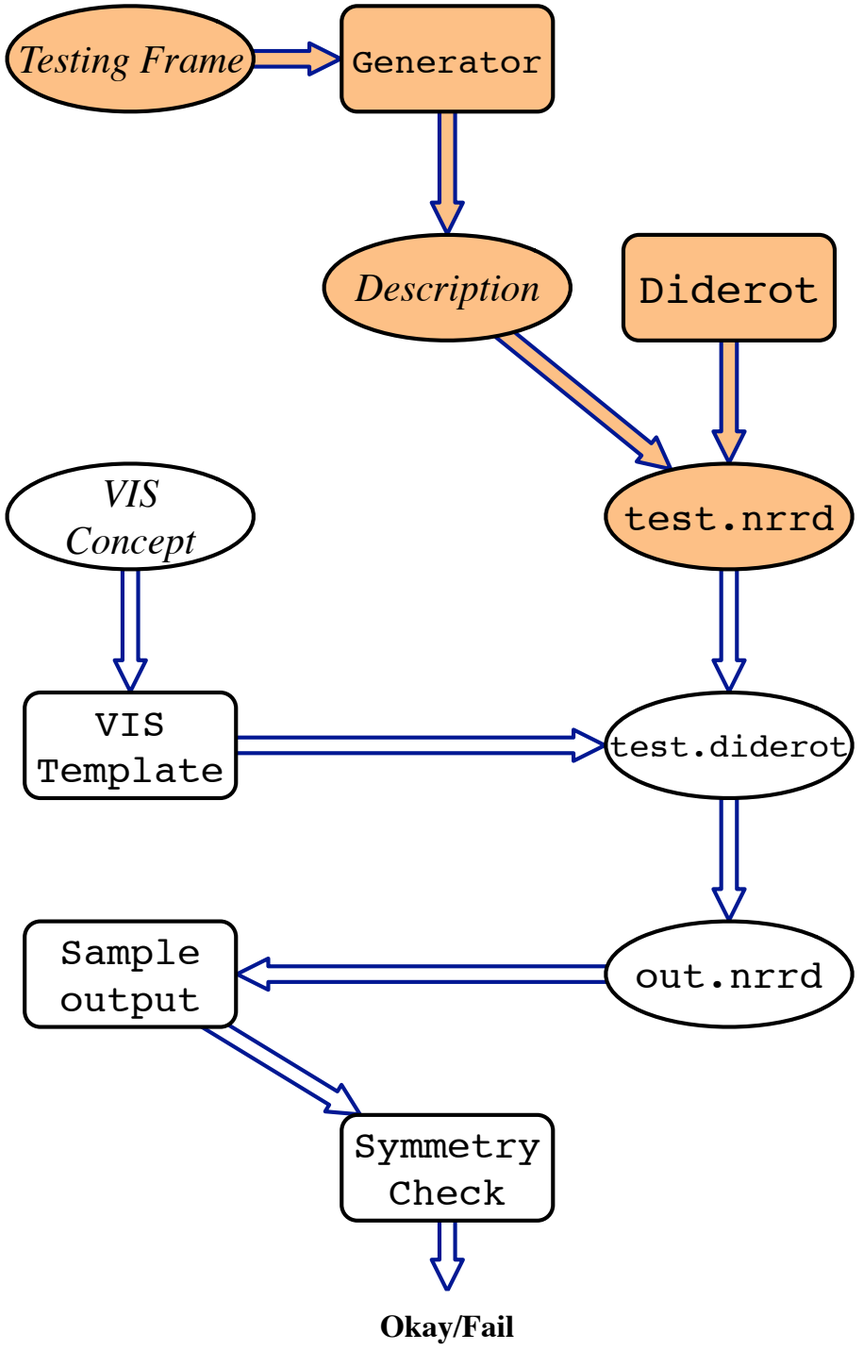


Visualization Automated Testing model



Visualization Automated Testing model

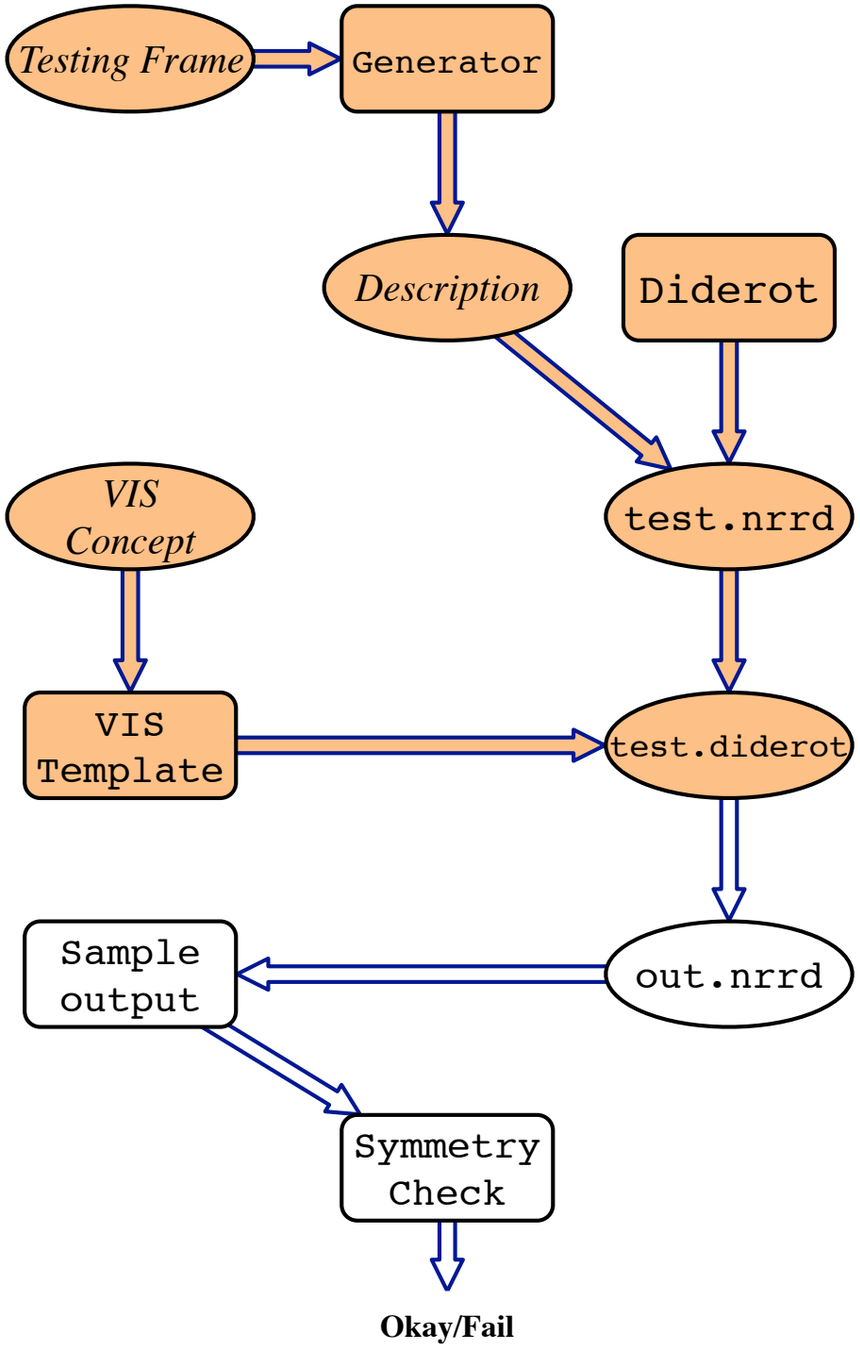
args: 3-d vec3 fields F1 and F2
operators: trace(F1 $\otimes$ F2)



Visualization Automated Testing model

args: 3-d vec3 fields F1 and F2
operators: trace(F1 $\otimes$ F2)

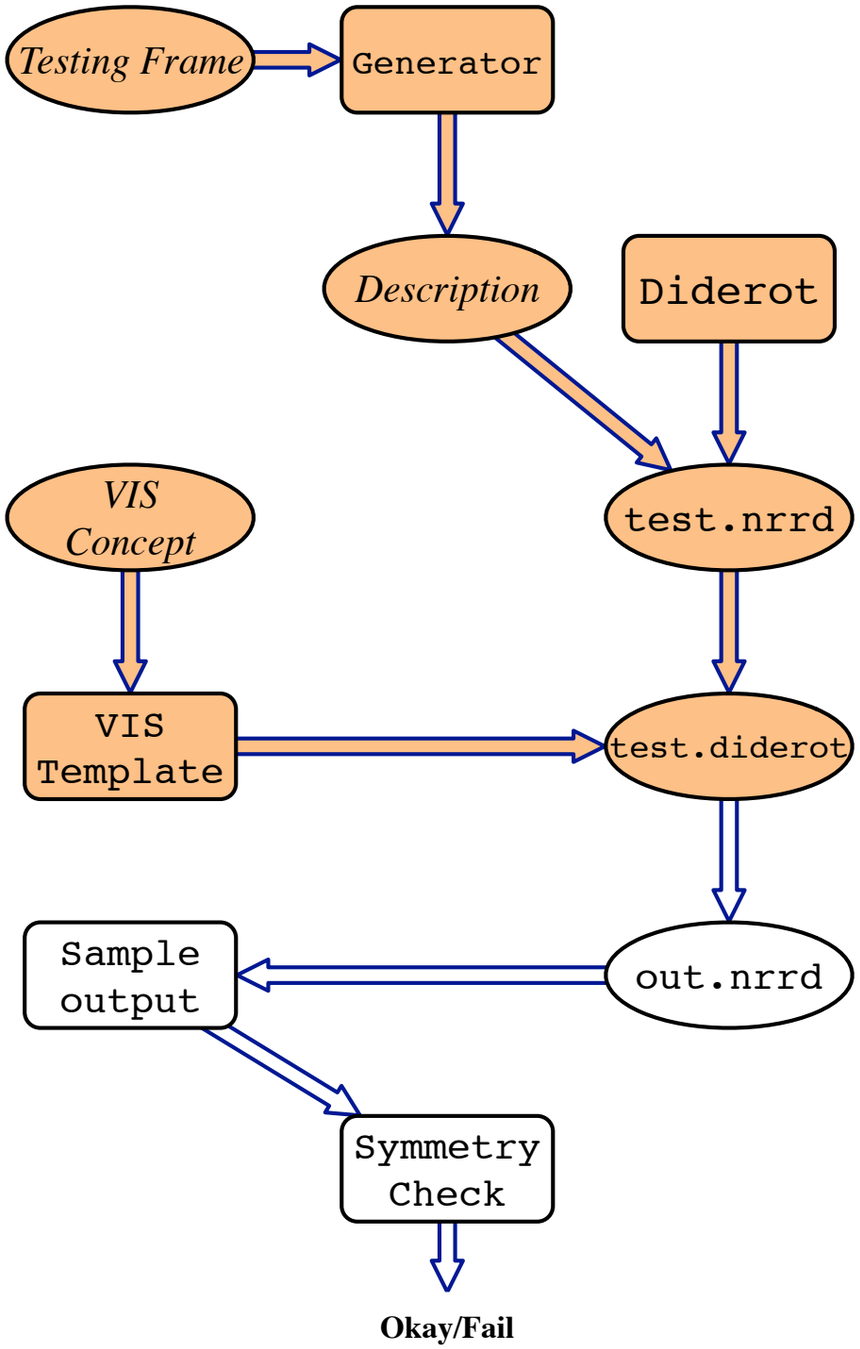
```
int res;
strand raycast (real ui, real vi)
   output real out;
   update {
     ...
     out = max (out,G(pos));
   }
} initially [raycast(ui,vi)|ui in 0 ..res-1, vi in 0..res-1];
```



Visualization Automated Testing model

args: 3-d vec3 fields F1 and F2
operators: trace(F1 $\otimes$ F2)

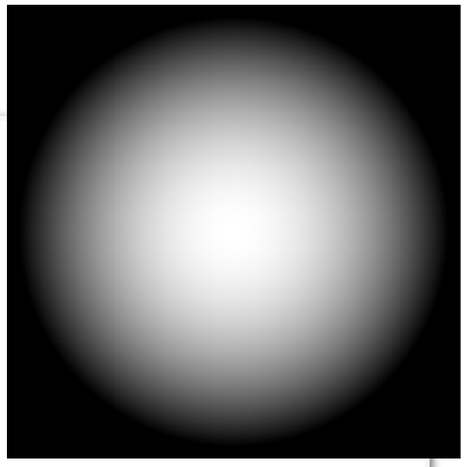
```
field#k(3)[3] F1 = "V1.nrrd" ⊛ h;
field#k(3)[3] F2 = "V2.nrrd" ⊛ h;
field#k(3)[] G= trace(F1 ⊗ F2);
tensor[3] pos;
int res;
strand raycast (real ui, real vi)
   output real out;
   update {
     ...
     out = max (out,G(pos));
   }
} initially [raycast(ui,vi)|ui in 0 ..res-1, vi in 0..res-1];
```

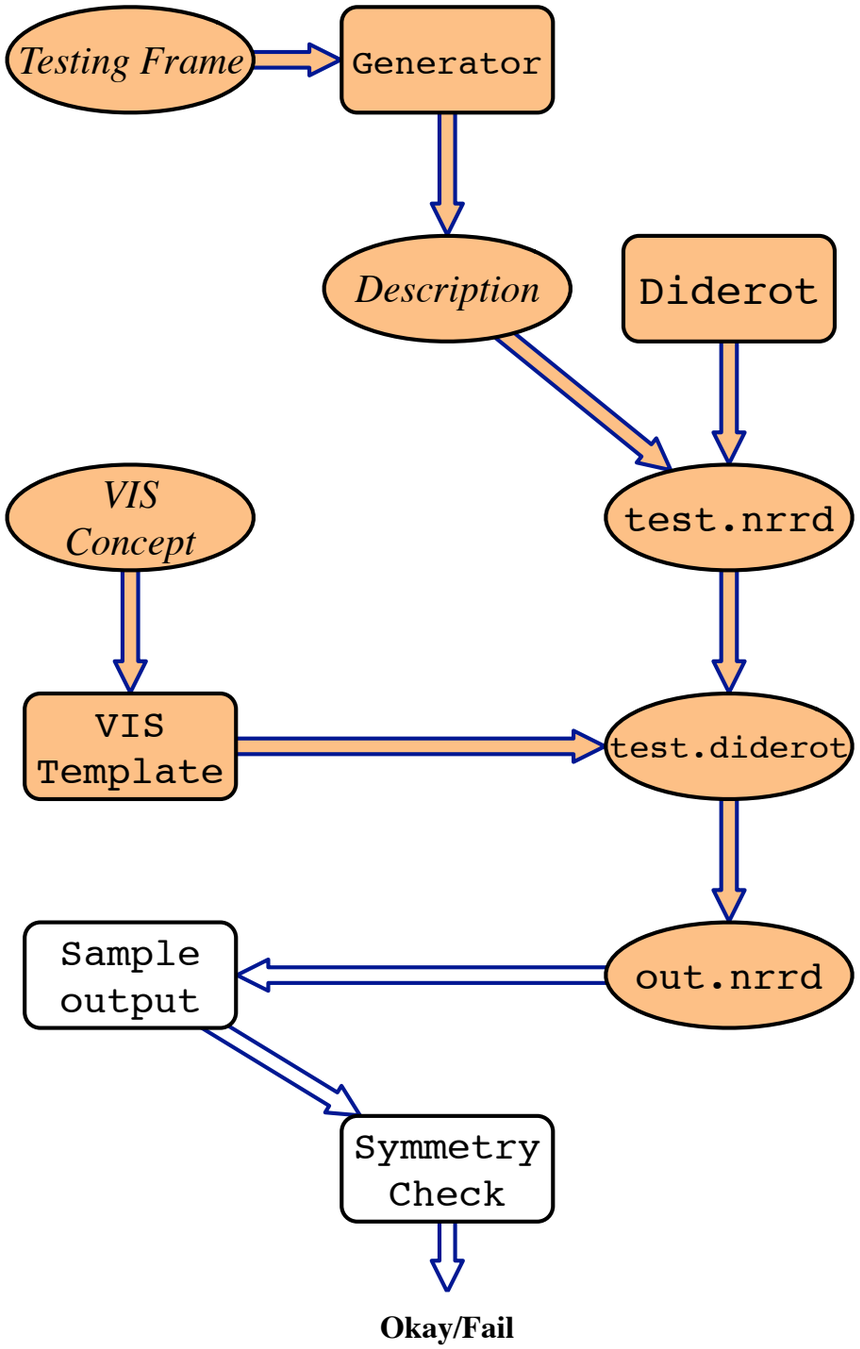


Visualization Automated Testing model

args: 3-d vec3 fields F1 and F2
operators: trace(F1 ⊗ F2)

```
field#k(3)[3] F1 = "V1.nrrd" ⊛ h;
field#k(3)[3] F2 = "V2.nrrd" ⊛ h;
field#k(3)[] G= trace(F1 ⊗ F2);
tensor[3] pos;
int res;
strand raycast (real ui, real vi)
   output real out;
   update {
     ...
     out = max (out,G(pos));
   }
} initially [raycast(ui,vi)|ui in 0 ..res-1, vi in 0..res-1];
```

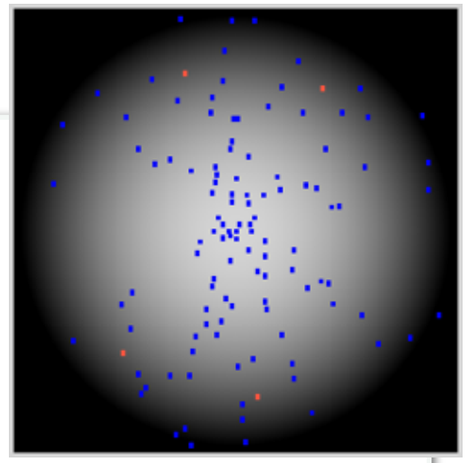


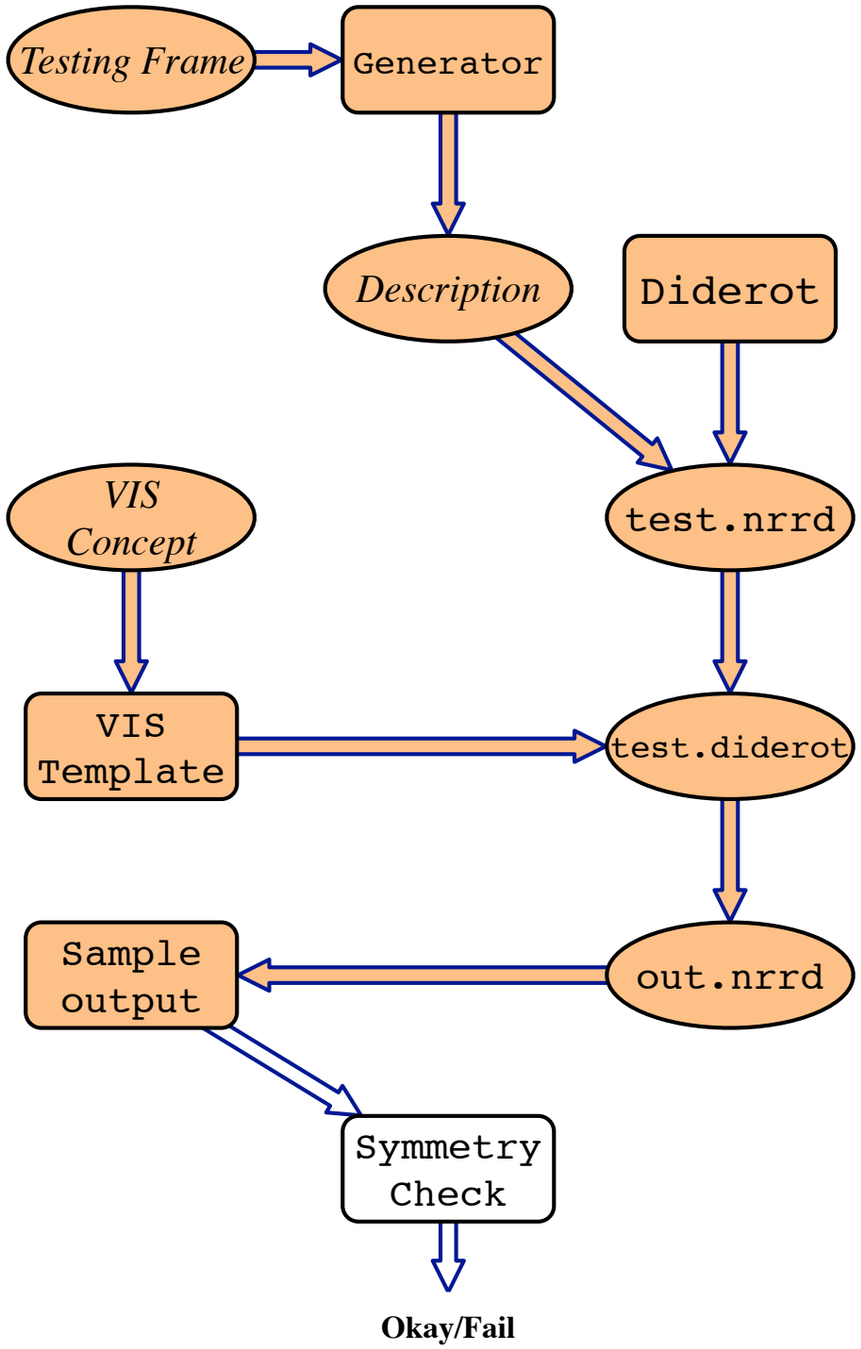


Visualization Automated Testing model

args: 3-d vec3 fields F1 and F2
operators: trace(F1 $\otimes$ F2)

```
field#k(3)[3] F1 = "V1.nrrd" ⊛ h;
field#k(3)[3] F2 = "V2.nrrd" ⊛ h;
field#k(3)[] G= trace(F1 ⊗ F2);
tensor[3] pos;
int res;
strand raycast (real ui, real vi)
   output real out;
   update {
     ...
     out = max (out,G(pos));
   }
} initially [raycast(ui,vi)|ui in 0 ..res-1, vi in 0..res-1];
```

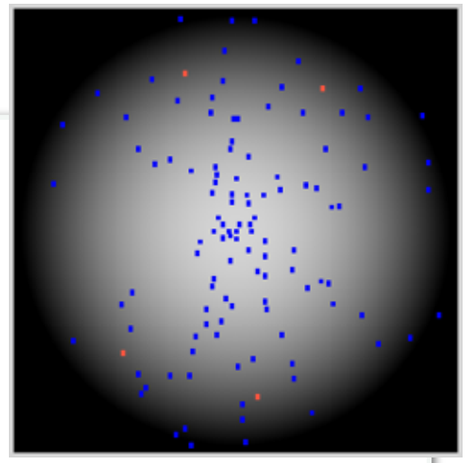




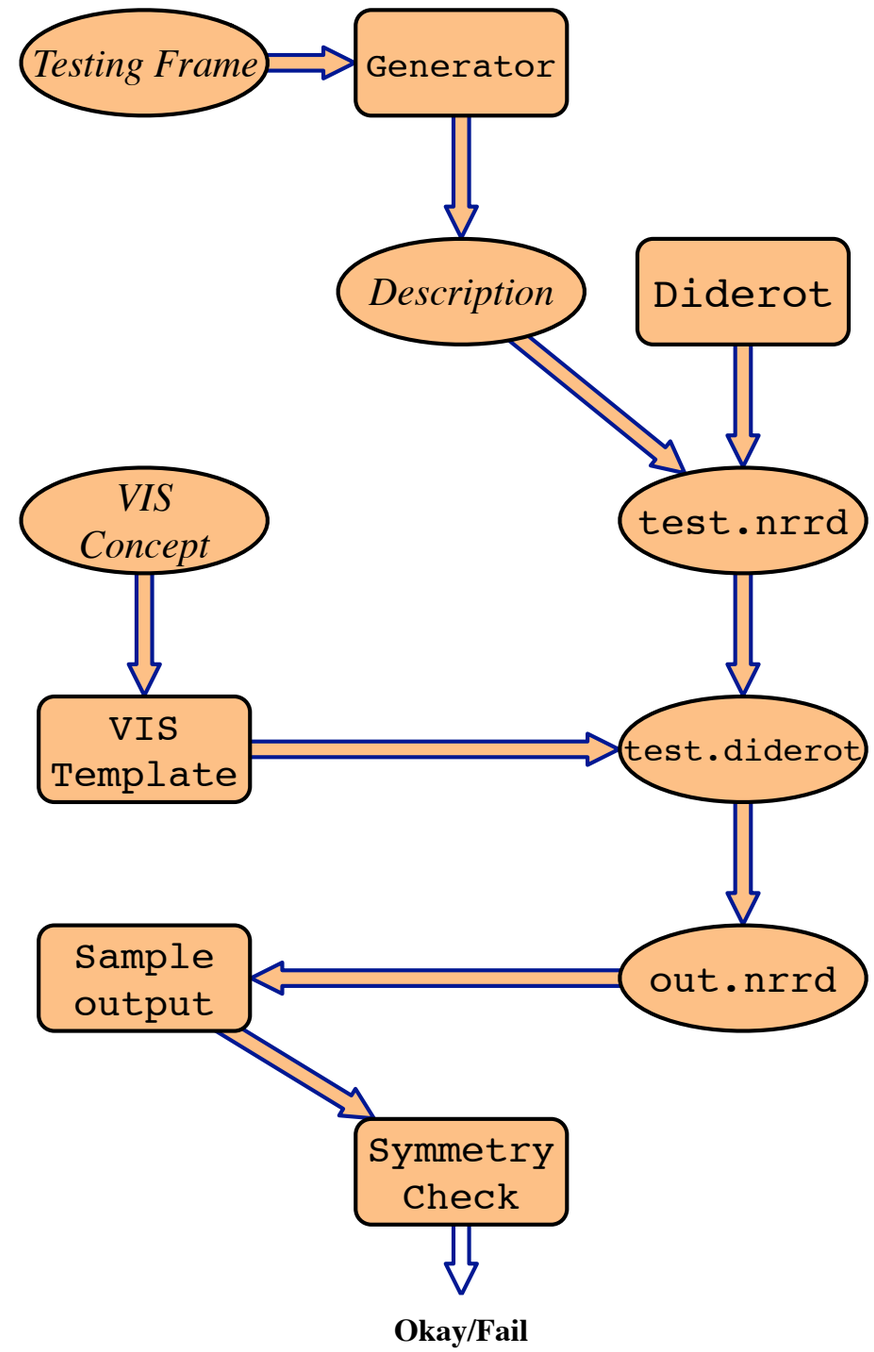
Visualization Automated Testing model

args: 3-d vec3 fields F1 and F2
operators: trace(F1 $\otimes$ F2)

```
field#k(3)[3] F1 = "V1.nrrd" ⊛ h;
field#k(3)[3] F2 = "V2.nrrd" ⊛ h;
field#k(3)[] G= trace(F1 ⊗ F2);
tensor[3] pos;
int res;
strand raycast (real ui, real vi)
   output real out;
   update {
     ...
     out = max (out,G(pos));
   }
} initially [raycast(ui,vi)|ui in 0 ..res-1, vi in 0..res-1];
```



- ☐ OK
- ☐ Type error
- ☐ Compiler error
- ☐ Numerical error



Found Bugs with DATm

Nested	Mode	Cause	Description
1	T	O	Restriction on trace
1	T	O	Restriction on transpose
1	T	O	Restriction on slice
2	T	O	Layout in using computations inline
1	T	E	Restriction on determinant
1	T	E	Restriction on modulate
1	T	E	Missing $\times$ between tensor and field
1	T	E	Generality $\otimes$ between tensor and field
2	C	O	Design in EIN IR indices
			Ex. $(U \times V)[1]$
2	C	R	Rewrite Parameter Id
			Ex.$\nabla(T + F)$
2	C	R	Wrapping summation in rewrite rule
2	C	R	Check dimension in Optimization
2	C	R	Translating to vector code
			Vectorize EIN without variable index
2	C	R	optimization not generalized
3	C	R	redundant indices in a single term
			Ex. $trace(modulate(-U, V))$
2	N	R	Rewrite field reconstruction indices
			Ex. transpose$(\nabla \otimes V)$
2	N	R	Data accessed incorrectly
			Reverse order for second order F
1	N	O	Algebraic error
			creating EIN operator. Ex. $\|\varphi\|$
1	N	O	creating the concatenation operator
2	N	R	determinant(concat(F,G))

- OK
- Type error
- Compiler error
- Numerical error

Found Bugs with DATm

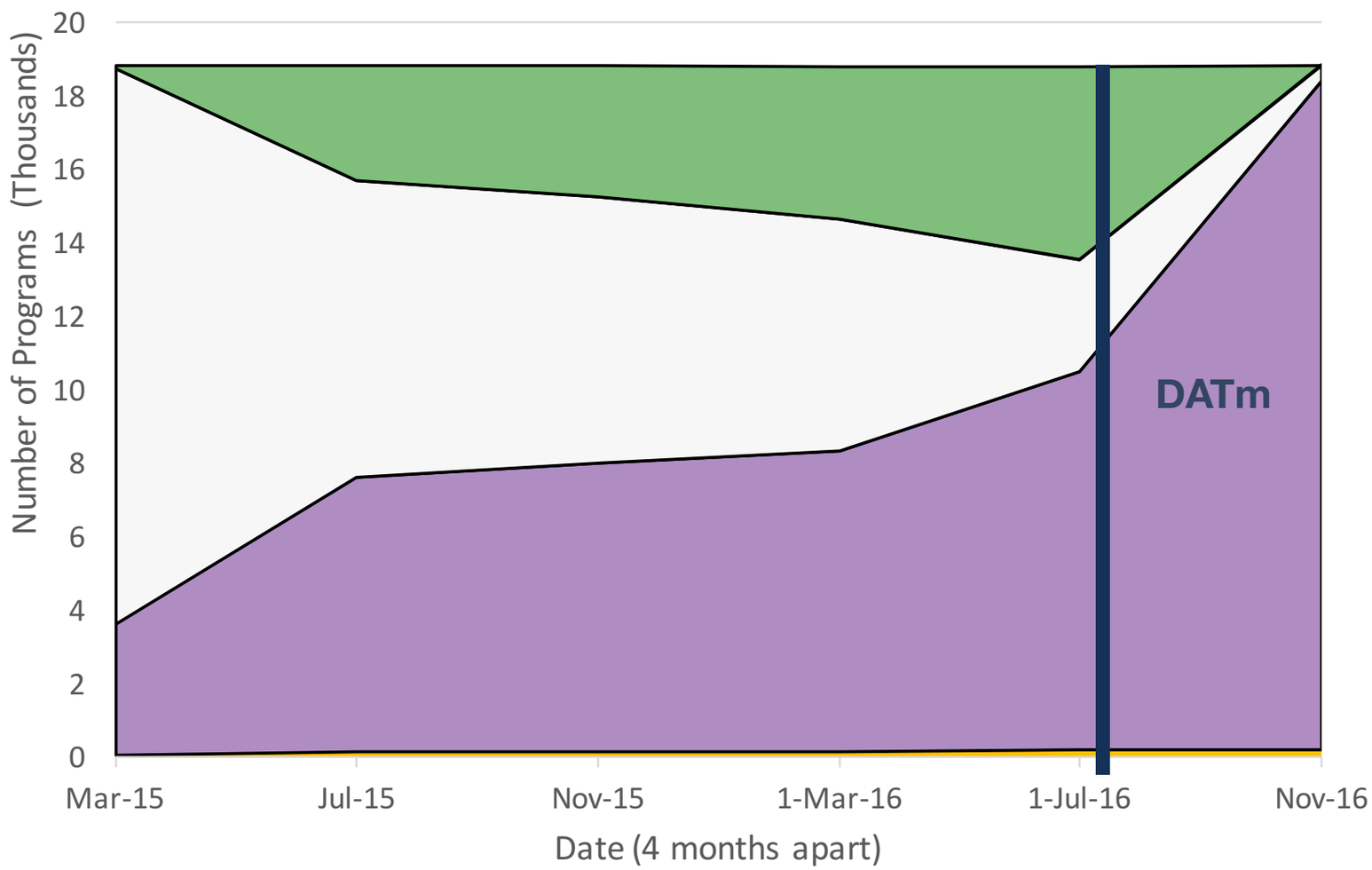
Nested	Mode	Cause	Description
1	T	O	Restriction on trace
1	T	O	Restriction on transpose
1	T	O	Restriction on slice
2	T	O	Layout in using computations inline
1	T	E	Restriction on determinant
1	T	E	Restriction on modulate
1	T	E	Missing $\times$ between tensor and field
1	T	E	Generality $\otimes$ between tensor and field
2	C	O	Design in EIN IR indices
			Ex. $(U \times V)[1]$
2	C	R	Rewrite Parameter Id
			Ex.$\nabla(T + F)$
2	C	R	Wrapping summation in rewrite rule
2	C	R	Check dimension in Optimization
2	C	R	Translating to vector code
			Vectorize EIN without variable index
2	C	R	optimization not generalized
3	C	R	redundant indices in a single term
			Ex. $trace(modulate(-U, V))$
2	N	R	Rewrite field reconstruction indices
			Ex. transpose$(\nabla \otimes V)$
2	N	R	Data accessed incorrectly
			Reverse order for second order F
1	N	O	Algebraic error
			creating EIN operator. Ex. $\|\varphi\|$
1	N	O	creating the concatenation operator
2	N	R	determinant(concat(F,G))

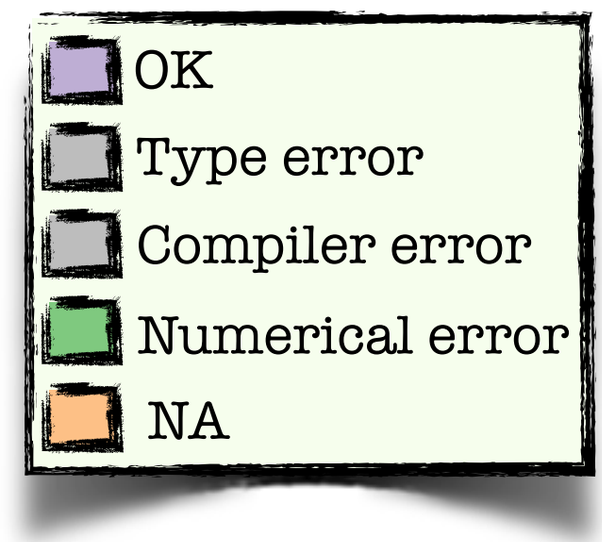
Bug exposed with

- different arguments
- combination of operators
- nested operators

- OK
- Type error
- Compiler error
- Numerical error

Snapshots of the Diderot compiler





- six different snapshots of the compiler
- generated almost 19,000 tests
- compilation errors may due to lack of feature support
- measures number of test programs not the number of bugs

Bug exposed with nested operators

Two nested operators

```
field#k(d)[d',d'] A,B;
field#k(d)[d',d'] M = modulate(A,B);
field#k(d)[] G = trace(M);
```

- [ ] OK
- [ ] Type error
- [x] Compiler error
- [ ] Numerical error

Bug exposed with nested operators

Two nested operators

```
field#k(d)[d',d'] A,B;
field#k(d)[d',d'] M = modulate(A,B);
field#k(d)[] G = trace(M);
```

Three nested operators

```
field#k(d)[d',d'] A,B;
field#k(d)[d',d'] M = modulate(-A,B);
field#k(d)[] G = trace(M);
```

- [ ] OK
- [ ] Type error
- [x] Compiler error
- [ ] Numerical error

Bug exposed with nested operators

Two nested operators

```
field#k(d)[d',d'] A,B;
field#k(d)[d',d'] M = modulate(A,B);
field#k(d)[] G = trace(M);
```

Three nested operators

```
field#k(d)[d',d'] A,B;
field#k(d)[d',d'] M = modulate(-A,B);
field#k(d)[] G = trace(M);
```

Creates an EIN form that triggers a different part of the compiler

- [ ] OK
- [ ] Type error
- [x] Compiler error
- [ ] Numerical error

Bug exposed with nested operators

Two nested operators

```
field#k(d)[d',d'] A,B;
field#k(d)[d',d'] M = modulate(A,B);
field#k(d)[] G = trace(M);
```

Three nested operators

```
field#k(d)[d',d'] A,B;
field#k(d)[d',d'] M = modulate(-A,B);
field#k(d)[] G = trace(M);
```

Creates an EIN form that triggers a different part of the compiler

$$\lambda(A,B)\langle\textstyle\sum_i(-A_{ii} * B_{ii})\rangle(A,B)$$

- [ ] OK
- [ ] Type error
- [x] Compiler error
- [ ] Numerical error

Bug exposed with different arguments

Checking a single operator applied to different argument types

- ☐ OK
- ■ Type error
- ☐ Compiler error
- ☐ Numerical error

Bug exposed with different arguments

Checking a single operator applied to different argument types

```
vec3 A,B;
vec3 C = A × B;
```

- ☐ OK
- ■ Type error
- ☐ Compiler error
- ☐ Numerical error

Bug exposed with different arguments

Checking a single operator applied to different argument types

```
vec3 A,B;
vec3 C = A × B;
field#k(d)[3] F,G;
field#k(d)[3] H = F × G;
```

- [ ] OK
- [x] Type error
- [ ] Compiler error
- [ ] Numerical error

Bug exposed with different arguments

Checking a single operator applied to different argument types

```
vec3 A,B;
vec3 C = A × B;
field#k(d)[3] F,G;
field#k(d)[3] H = F × G;
field#k(d)[3] H = G × B;
```

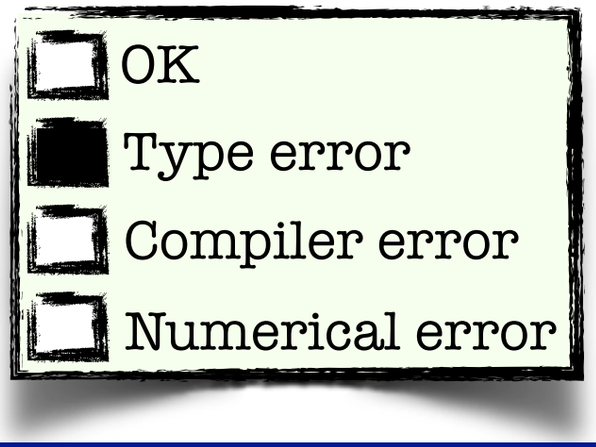
- ☐ OK
- ■ Type error
- ☐ Compiler error
- ☐ Numerical error

Bug exposed with different arguments

Checking a single operator applied to different argument types

```
vec3 A,B;
vec3 C = A × B;
field#k(d)[3] F,G;
field#k(d)[3] H = F × G;
field#k(d)[3] H = G × B;
```

Fixing creates a richer and more complete language



Exhaustive and random testing

		Timing to run $DATm$ with given probability.					
No.of Operators	No. of Programs	0 %	0.5 %	1%	5%	10%	100%
1	695	0.25	0.3	0.55	2.6	4.65	32.53
2	18,819	7.62	14.43	18.35	64.2	121.23	1099.16
3	495,626	58.83	246.02	393.2	-	-	-

- Exhaustive search: Offers full coverage
- Random search: Deeper search space

Related Work

- Optimizations and Techniques: Spiral, TCE, COFFEE, UFL
- VIS DSLs: Vivaldi, Scout, ViSLang
- "Verification is the process of assessing software correctness and numerical accuracy of the solution to a given mathematical model." [Silva et.al'15]

- Comparison of different techniques [Chen et.al'09]
- Randomized differential testing [Yang et.al'11, Lindig'05, McKeeman'98]
- Metamorphic testing [Donaldson et.al'16, Tao et.al.' 10]
- DUFT test DSL programs but not implementation[Wu et.al.' 09]

Conclusion

Conclusion

- Demonstrated a way to test a high level language based on correctness
- Showed bugs and results

Conclusion

- Demonstrated a way to test a high level language based on correctness
- Showed bugs and results

Future Work

- Testing coverage
- Smarter test generation
- Clear bug reporting
- Types of testing
- Parallel testing

Thank you!

DATm
https://github.com/cchiw/DATm

Diderot
http://diderot-language.cs.uchicago.edu/

contact:
cchiw@uchicago.edu

Acknowledgement: This material is based upon work supported by the NSF under Grants CCF-1446412 and CCF-1564298

Benefits

- Adding new operators is easy
- Targeted testing helps with debugging
- Can create uncommon programs

Limitations

The limitation attribute refers to the value restriction on an argument

$$\mathrm{op}_{sqrt} = \{out : \mathrm{sqrt}, placement : \mathrm{split}, arity : 1, limit : \mathrm{positive}\}$$

```
//check limitations
if(e(pos) > 0){observed = √e(pos);//include item}
else{observed=null//do not include item}
```

Limitations

The limitation attribute refers to the value restriction on an argument

Operators

- -out
- -placement
- -arity
- -limit

$$\mathrm{op}_{sqrt} = \{out : \mathrm{sqrt}, placement : \mathrm{split}, arity : 1, limit : \mathrm{positive}\}$$

```
//check limitations
if(e(pos) > 0){observed = √e(pos);//include item}
else{observed=null//do not include item}
```

```
tensor[3] T;
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
int n;
strand sample (int i) {
 tensor[3,3] out;
 update {
  tensor[3] pos;
   if(i==1) { pos = pos1; }
   ...
   else if(i==n) { pos = posn; }
   out = G(pos);
  }
}initially [sample(i)| i in 0..n];
```

Limitations

The limitation attribute refers to the value restriction on an argument

Operators

- -out
- -placement
- -arity
- -limit

$$\mathrm{op}_{sqrt} = \{out : \mathrm{sqrt}, placement : \mathrm{split}, arity : 1, limit : \mathrm{positive}\}$$

```
//check limitations
if(e(pos) > 0){observed = √e(pos);//include item}
else{observed=null//do not include item}
```

```
tensor[3] T;
field#k(3)[] F = "test.nrrd";
field#k(3)[] G = T⊗∇F;
int n;
strand sample (int i) {
 tensor[3,3] out;
 update {
  tensor[3] pos;
   if(i==1) { pos = pos1; }
   ...
   else if(i==n) { pos = posn; }
   out = G(pos);
  }
}initially [sample(i)| i in 0..n];
```

Language Expressivity

- [ ] OK
- [x] Type error
- [ ] Compiler error
- [ ] Numerical error

Language Expressivity

The type specification for the trace operation in DATm

$$\mathtt{trace} \ : \ \mathbf{field}\#k(d)[i,i] \ \text{->} \ \mathbf{field}\#k(d)[]$$

- [ ] OK
- [x] Type error
- [ ] Compiler error
- [ ] Numerical error

Language Expressivity

The type specification for the trace operation in DATm

$$\mathtt{trace} \ : \ \mathbf{field}\#k(d)[i,i] \ \text{->} \ \mathbf{field}\#k(d)[]$$

Implementation in Diderot compiler

$$\mathtt{trace} \ : \ \mathbf{field}\#k(d)[d,d] \ \text{->} \ \mathbf{field}\#k(d)[]$$

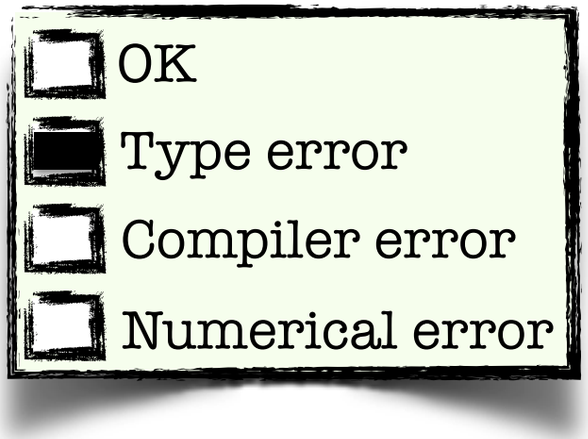
- [ ] OK
- [x] Type error
- [ ] Compiler error
- [ ] Numerical error

Language Expressivity

The type specification for the trace operation in DATm

$$\texttt{trace} \;:\; \mathbf{field}\#k(d)[i,i] \;\texttt{->}\; \mathbf{field}\#k(d)[]$$

Implementation in Diderot compiler

$$\texttt{trace} \;:\; \mathbf{field}\#k(d)[d,d] \;\texttt{->}\; \mathbf{field}\#k(d)[]$$

Fixing creates a richer and more complete language