Matrix Expressions and BLAS/LAPACK

Matthew Rocklin

June 27th, 2013

Need For High Level Compilers

Math is Important

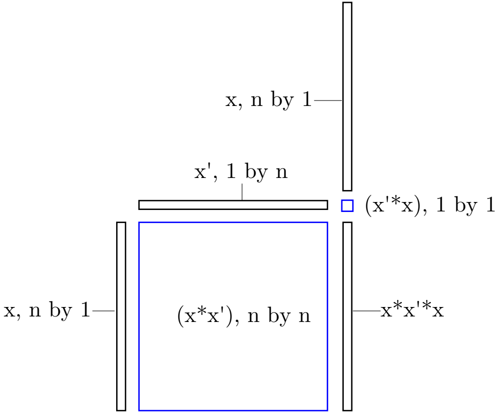
```
// A: Naive
int fact(int n){
    if (n == 0)
        return 1;
    return n*fact(n-1);
}
```

```
// B: Programmer
int fact(int n){
    int prod = n;
    while(n--)
        prod *= n;
    return prod;
}
```

```
// C: Mathematician
int fact(int n){
    // n! = Gamma(n+1)
    return lround(exp(lgamma(n+1)))
}
```

Evan Miller http://www.evanmiller.org/mathematical-hacker.html

```
x = matrix(ones(10000, 1))

x*x.T*x                 Elapsed time is    ?     seconds.
(x*x.T)*x               Elapsed time is 0.337711 seconds.
x*(x.T*x)               Elapsed time is 0.000956 seconds.
```



If $\mathbf{A}$ is symmetric positive-definite and $\mathbf{B}$ is orthogonal:

Question: is $\mathbf{B} \cdot \mathbf{A} \cdot \mathbf{B}^\top$ symmetric and positive-definite?

Answer: Yes.

Question: Could a computer have told us this?

Answer: Probably.

`http://scicomp.stackexchange.com/questions/74/symbolic-software-packages-for-matrix-expressions/`

$$\beta = (X^T X)^{-1} X^T y$$

Python/NumPy

```
beta = (X.T*X).I * X.T*y
```

MatLab

```
beta = inv(X'*X) * X'*y
```

$$\beta = (X^T X)^{-1} X^T y$$

Python/NumPy

```
beta = solve(X.T*X, X.T*y)
```

MatLab

```
beta = X'*X \ X'*y
```

$$\beta = (X^T X)^{-1} X^T y$$

Python/NumPy

```
beta = solve(X.T*X, X.T*y, sym_pos=True)
```

MatLab

```
beta = (X'*X) \ (X'*y)
```

BLAS/LAPACK

Numeric libraries for dense linear algebra

- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$

BLAS/LAPACK

Numeric libraries for dense linear algebra

- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`

BLAS/LAPACK

Numeric libraries for dense linear algebra

- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- `DSYMM` - **D**ouble precision **SY**mmetric **M**atrix **M**ultiply – $\alpha AB + \beta C$

BLAS/LAPACK

Numeric libraries for dense linear algebra

- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- `DSYMM` - **D**ouble precision **SY**mmetric **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`

BLAS/LAPACK

Numeric libraries for dense linear algebra

- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- `DSYMM` - **D**ouble precision **SY**mmetric **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- ...

BLAS/LAPACK

Numeric libraries for dense linear algebra

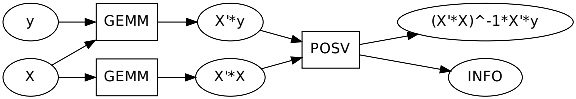
- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- `DSYMM` - **D**ouble precision **SY**mmetric **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- ...
- `DPOSV` - **D**ouble symmetric **PO**sitive definite matrix **S**ol**V**e – $A^{-1}y$

BLAS/LAPACK

Numeric libraries for dense linear algebra

- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- `DSYMM` - **D**ouble precision **SY**mmetric **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- ...
- `DPOSV` - **D**ouble symmetric **PO**sitive definite matrix **S**ol**V**e – $A^{-1}y$
  - `SUBROUTINE DPOSV( UPLO, N, NRHS, A, LDA, B, LDB, INFO )`

```
Naive  :     (X.T*X).I * X.T*y

Expert :     solve(X.T*X, X.T*y, sym_pos=True)
```




```
subroutine least_squares(X, y)
 ...
endsubroutine
```

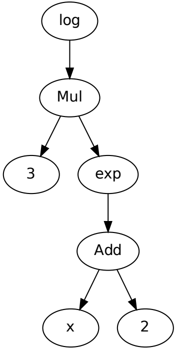
Matrix Expressions and SymPy

SymPy Expressions

Operators (`Add`, `log`, `exp`, `sin`, `integral`, `derivative`, ...) are Python classes

Terms (`3`, `x`, `log(3*x)`, `integral(x**2)`, ...) are Python objects

```
>>> x = Symbol('x')
>>> expr = log(3*exp(x + 2))

>>> simplify(expr)
x + 2 + log(3)
```

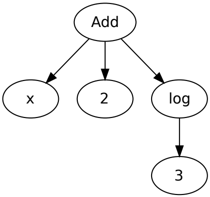


SymPy Expressions

Operators (`Add`, `log`, `exp`, `sin`, `integral`, `derivative`, ...) are Python classes

Terms (`3`, `x`, `log(3*x)`, `integral(x**2)`, ...) are Python objects

```
>>> x = Symbol('x')
>>> expr = log(3*exp(x + 2))

>>> simplify(expr)
x + 2 + log(3)
```

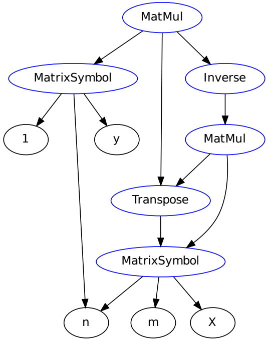


Inference

```
>>> x = Symbol('x')
>>> y = Symbol('y')

>>> facts = Q.real(x) & Q.positive(y)
>>> query = Q.positive(x**2 + y)

>>> ask(query, facts)
True
```

Matrix Expressions

```
>>> X = MatrixSymbol('X', n, m)
>>> y = MatrixSymbol('y', n, 1)

>>> beta = (X.T*X).I * X.T*y
```



Matrix Inference

If **A** is symmetric positive-definite and **B** is orthogonal:

Question: is $\mathbf{B} \cdot \mathbf{A} \cdot \mathbf{B}^\top$ symmetric and positive-definite?

Answer: Yes.

Question: Could a computer have told us this?

Answer: Probably.

---

sympy.matrices.expressions

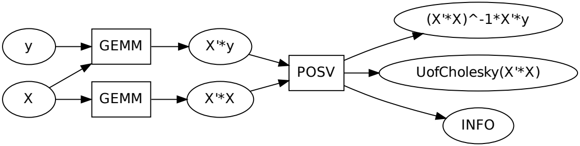
```
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, n)

>>> facts = Q.symmetric(A) & Q.positive_definite(A) & Q.orthogonal(B)
>>> query = Q.symmetric(B*A*B.T) & Q.positive_definite(B*A*B.T)

>>> ask(query, facts)
True
```

Computations

Numeric libraries for dense linear algebra

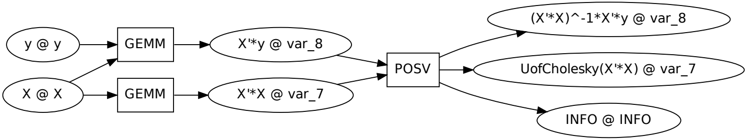
- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- `DSYMM` - **D**ouble precision **SY**mmetric **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- ...
- `DPOSV` - **D**ouble symmetric **PO**sitive definite matrix **S**ol**V**e – $A^{-1}y$
  - `SUBROUTINE DPOSV( UPLO, N, NRHS, A, LDA, B, LDB, INFO )`

http://github.com/mrocklin/computations




```
comp = (GEMM(1, X.T, X, 0, 0)
      + GEMM(1, X.T, y, 0, 0)
      + POSV(X.T*X, X.T*y))

class GEMM(Computation):
    """ Genreral Matrix Multiply """
    inputs    = [alpha, A, B, beta, C]
    outputs   = [alpha*A*B + beta*C]
    inplace   = {0: 4}
    fortran   = ....

class POSV(Computation):
    """ Symmetric Positive Definite Matrix Solve """
    inputs    = [A, y]
    outputs   = [UofCholesky(A), A.I*y]
    inplace   = {0: 0, 1: 1}
    condition = Q.symmetric(A) & Q.positive_definite(A)
    fortran   = ....
```

http://github.com/mrocklin/computations




```
comp = (GEMM(1, X.T, X, 0, 0)
      + GEMM(1, X.T, y, 0, 0)
      + POSV(X.T*X, X.T*y))

class GEMM(Computation):
    """ Genreral Matrix Multiply """
    inputs    = [alpha, A, B, beta, C]
    outputs   = [alpha*A*B + beta*C]
    inplace   = {0: 4}
    fortran   = ....

class POSV(Computation):
    """ Symmetric Positive Definite Matrix Solve """
    inputs    = [A, y]
    outputs   = [UofCholesky(A), A.I*y]
    inplace   = {0: 0, 1: 1}
    condition = Q.symmetric(A) & Q.positive_definite(A)
    fortran   = ....
```

```fortran
subroutine f(X, y, var_7, m, n)
implicit none

integer, intent(in) :: m
integer, intent(in) :: n
real*8, intent(in) :: y(n)          !  y
real*8, intent(in) :: X(n, m)       !  X
real*8, intent(out) :: var_7(m)     !  0 -> X'*y -> (X'*X)^-1*X'*y
real*8 :: var_8(m, m)               !  0 -> X'*X
integer :: INFO                     !  INFO

call dgemm('N', 'N', m, 1, n, 1.0, X, n, y, n, 0.0, var_7, m)
call dgemm('N', 'N', m, m, n, 1.0, X, n, X, n, 0.0, var_8, m)
call dposv('U', m, 1, var_8, m, var_7, m, INFO)

RETURN
END
```

Logic Programming

TODO

- `http://github.com/mrocklin/term`
  An interface for terms
  Composable with legacy code via Monkey patching
  Supports pattern matching via Unification
- `http://github.com/logpy/logpy`
  Implements miniKanren, a logic programming language
- `http://github.com/logpy/strategies`
  Partially implements Stratego, a control flow programming language

Automation

Have

```
(X.T*X).I*X.T*y
Q.fullrank(X)
```

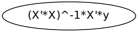
Want

```
comp = (GEMM(1, X.T, X, 0, 0)
      + GEMM(1, X.T, y, 0, 0)
      + POSV(X.T*X, X.T*y))
```

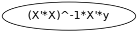
Accomplish Using Pattern Matching

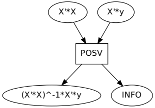
```
# Source Expression,  Target Computation,         Condition
(alpha*A*B + beta*C, SYMM(alpha, A, B, beta, C), Q.symmetric(A) | Q.symmetric(B
(alpha*A*B + beta*C, GEMM(alpha, A, B, beta, C), True),
(A.I*B,              POSV(A, B),        Q.symmetric(A) & Q.positive_definite(A)
(A.I*B,              GESV(A, B),        True),
(alpha*A + B,        AXPY(alpha, A, B), True),
```

Pattern Matching done with LogPy, a composable Logic Programming library

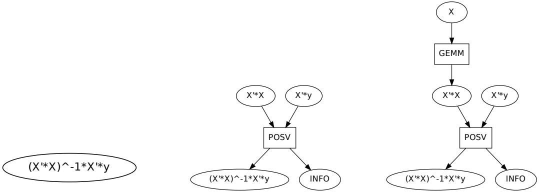



```
# Source Expression,  Target Computation,        Condition
(alpha*A*B + beta*C, SYMM(alpha, A, B, beta, C), Q.symmetric(A) | Q.symmetric(B
(alpha*A*B + beta*C, GEMM(alpha, A, B, beta, C), True),
(A.I*B,              POSV(A, B),         Q.symmetric(A) & Q.positive_definite(A)
(A.I*B,              GESV(A, B),         True),
(alpha*A + B,        AXPY(alpha, A, B), True),
```

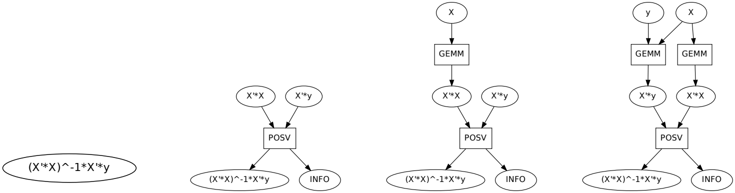





```
# Source Expression,  Target Computation,        Condition
(alpha*A*B + beta*C, SYMM(alpha, A, B, beta, C), Q.symmetric(A) | Q.symmetric(B
(alpha*A*B + beta*C, GEMM(alpha, A, B, beta, C), True),
(A.I*B,              POSV(A, B),        Q.symmetric(A) & Q.positive_definite(A)
(A.I*B,              GESV(A, B),        True),
(alpha*A + B,        AXPY(alpha, A, B), True),
```



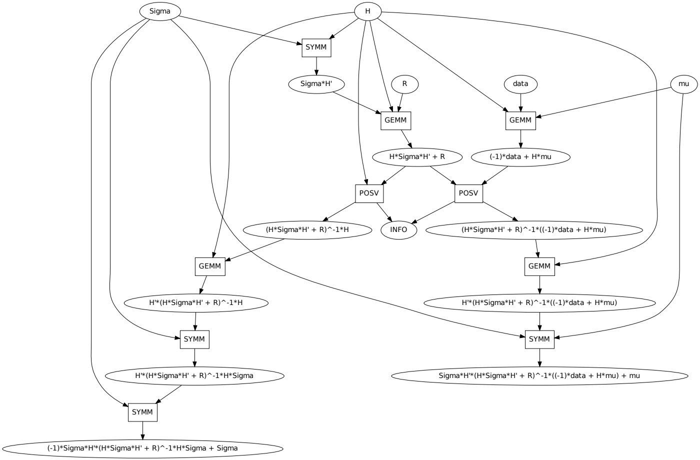

```
# Source Expression,  Target Computation,        Condition
(alpha*A*B + beta*C, SYMM(alpha, A, B, beta, C), Q.symmetric(A) | Q.symmetric(B
(alpha*A*B + beta*C, GEMM(alpha, A, B, beta, C), True),
(A.I*B,              POSV(A, B),        Q.symmetric(A) & Q.positive_definite(A)
(A.I*B,              GESV(A, B),        True),
(alpha*A + B,        AXPY(alpha, A, B), True),
```




```
# Source Expression,  Target Computation,        Condition
(alpha*A*B + beta*C, SYMM(alpha, A, B, beta, C), Q.symmetric(A) | Q.symmetric(B
(alpha*A*B + beta*C, GEMM(alpha, A, B, beta, C), True),
(A.I*B,              POSV(A, B),         Q.symmetric(A) & Q.positive_definite(A)
(A.I*B,              GESV(A, B),         True),
(alpha*A + B,        AXPY(alpha, A, B), True),
```

Kalman Filter

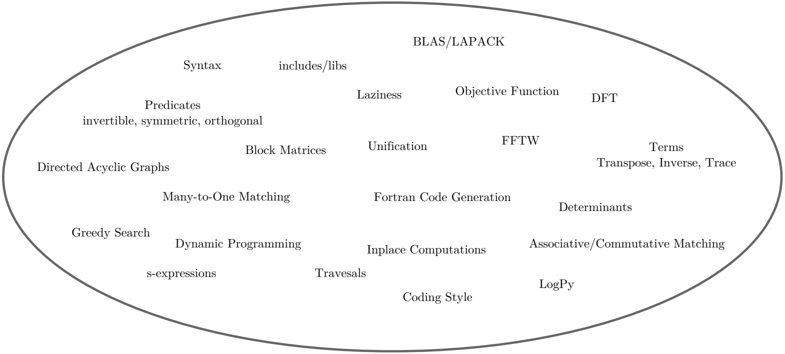
```
newmu       = mu + Sigma*H.T * (R + H*Sigma*H.T).I * (H*mu - data)
newSigma    = Sigma - Sigma*H.T * (R + H*Sigma*H.T).I * H * Sigma

assumptions = [Q.positive_definite(Sigma), Q.symmetric(Sigma),
               Q.positive_definite(R), Q.symmetric(R), Q.fullrank(H)]

f = fortran_function([mu, Sigma, H, R, data], [newmu, newSigma], *assumptions)
```

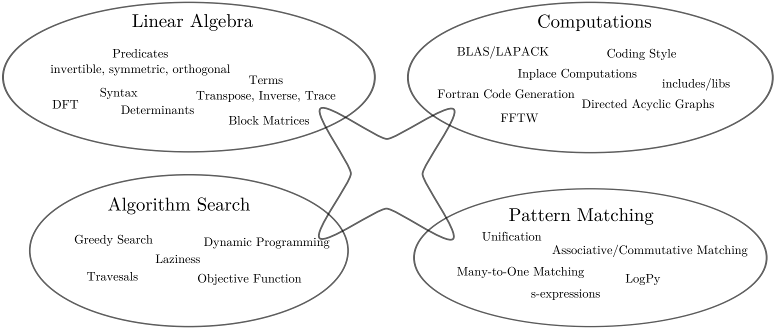


Kalman Filter

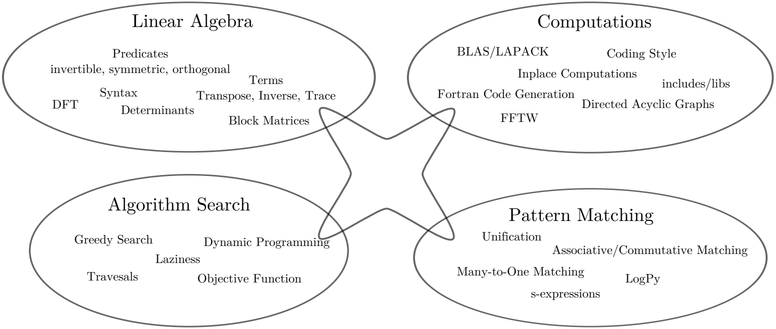
```
subroutine f(mu, Sigma, H, R, data, mu_2, Sigma_2)
  implicit none

  real(kind=8), intent(in) :: mu(:)
  real(kind=8), intent(in) :: Sigma(:,:)
  real(kind=8), intent(in) :: H(:,:)
  real(kind=8) :: R(:,:)
  real(kind=8) :: data(:)
  real(kind=8), intent(out) :: mu_2(:)
  real(kind=8), intent(out) :: Sigma_2(:,:)

! Variable Declarations !   -- Cut for space --

  call dcopy(n**2, Sigma, 1, Sigma_2, 1)
  call dgemm('N', 'N', k, 1, n, 1.0d+0, H, k, mu, n, -1.0d+0, data, k)
  call dcopy(n, mu, 1, mu_2, 1)
  call dcopy(k*n, H, 1, H_2, 1)
  call dsymm('R', 'U', k, n, 1.0d+0, Sigma, n, H, k, 0.0d+0, var_17, k)
  call dgemm('N', 'T', k, k, n, 1.0d+0, H, k, var_17, k, 1.0d+0, R, k)
  call dposv('U', k, n, R, k, H_2, k, INFO)
  call dpotrs('U', k, 1, R, k, data, k, INFO)
  call dgemm('T', 'N', n, 1, k, 1.0d+0, H, k, data, k, 0.0d+0, var_12, n)
  call dsymm('L', 'U', n, 1, 1.0d+0, Sigma, n, var_12, n, 1.0d+0, mu_2, n)
  call dgemm('T', 'N', n, n, k, 1.0d+0, H, k, H_2, k, 0.0d+0, var_19, n)
  call dsymm('R', 'U', n, n, 1.0d+0, Sigma, n, var_19, n, 0.0d+0, var_18, n)
  call dsymm('L', 'U', n, n, -1.0d+0, Sigma_2, n, var_18, n, 1.0d+0, Sigma_2, n
  return
```

Software Design


BLAS/LAPACK
Syntax
includes/libs
Laziness
Objective Function
DFT
Predicates
invertible, symmetric, orthogonal
Block Matrices
Unification
FFTW
Terms
Transpose, Inverse, Trace
Directed Acyclic Graphs
Many-to-One Matching
Fortran Code Generation
Determinants
Greedy Search
Dynamic Programming
Inplace Computations
Associative/Commutative Matching
s-expressions
Travesals
LogPy
Coding Style

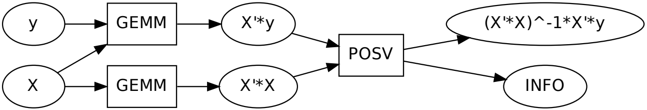

Linear Algebra
Predicates
invertible, symmetric, orthogonal
Terms
Syntax
Transpose, Inverse, Trace
DFT
Determinants
Block Matrices
Computations
BLAS/LAPACK
Coding Style
Inplace Computations
includes/libs
Fortran Code Generation
Directed Acyclic Graphs
FFTW
Algorithm Search
Greedy Search
Dynamic Programming
Laziness
Travesals
Objective Function
Pattern Matching
Unification
Associative/Commutative Matching
Many-to-One Matching
LogPy
s-expressions

`git@github.com/sympy/sympy.git` `git@github.com/mrocklin/computations.git`

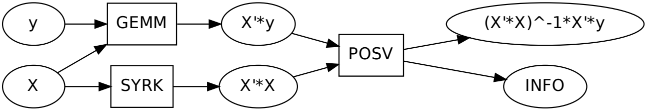


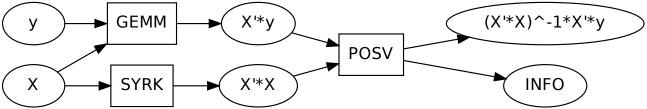
`git@github.com/logpy/strategies.git` `git@github.com/logpy/logpy.git`

```
X = MatrixSymbol('X', n, m)
y = MatrixSymbol('y', n, 1)

inputs  = [X, y]
outputs = [(X.T*X).I*X.T*y]
facts   = fullrank(X)
```



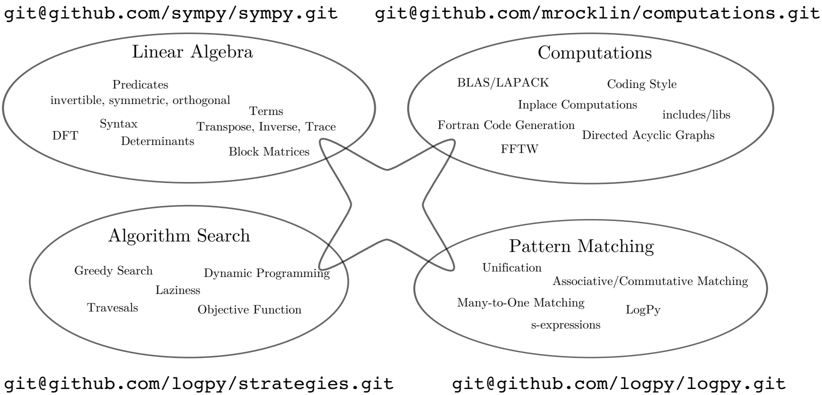
```
X = MatrixSymbol('X', n, m)
y = MatrixSymbol('y', n, 1)

inputs  = [X, y]
outputs = [(X.T*X).I*X.T*y]
facts   = fullrank(X)
```



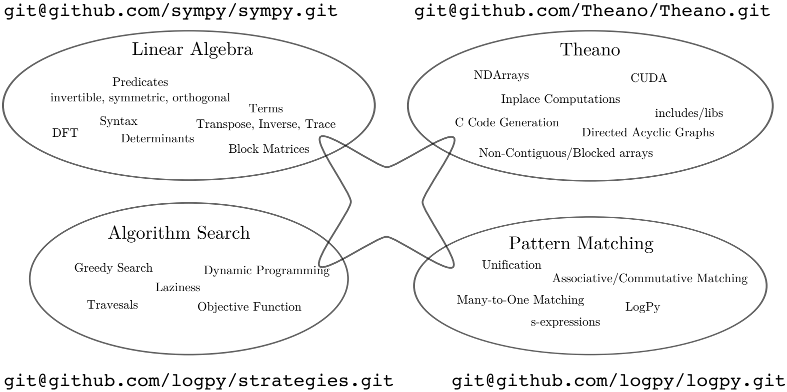



```
class SYRK(Computation):
    """ Symmetric Rank-K Update ‘alpha X’ X + beta Y’ """
    inputs  = (alpha, A, beta, D)
    outputs = (alpha * A * A.T + beta * D,)
    inplace  = {0: 3}
    fortran_template = ("call %(fn)s('%(UPLO)s', '%(TRANS)s', %(N)s, %(K)s, "
                        "%(alpha)s, %(A)s, %(LDA)s, "
                        "%(beta)s, %(D)s, %(LDD)s)")
    ...

  (alpha*A*A.T + beta*D, SYRK(alpha, A, beta, D), True),
  (A*A.T,                SYRK(1.0, A, 0.0, 0),    True),
```

Separation promotes Comparison and Experimentation

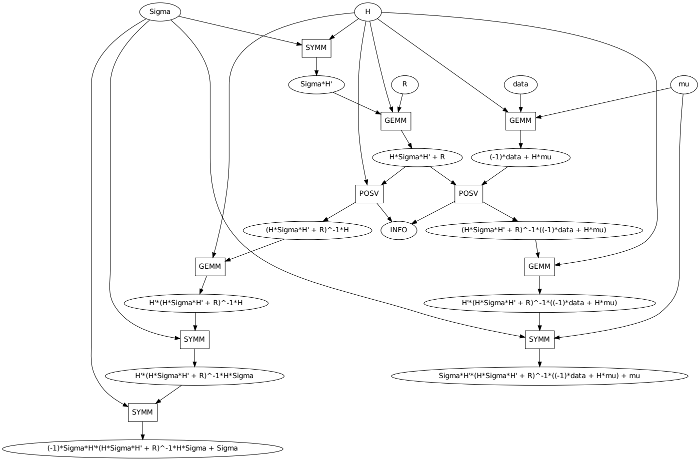


Separation promotes Comparison and Experimentation



Kalman Filter - Theano v. Fortran

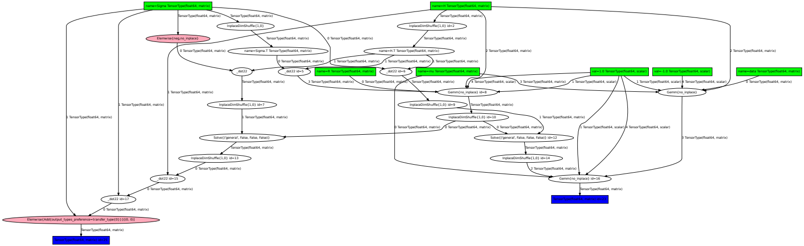
```
newmu       = mu + Sigma*H.T * (R + H*Sigma*H.T).I * (H*mu - data)
newSigma    = Sigma - Sigma*H.T * (R + H*Sigma*H.T).I * H * Sigma

assumptions = [Q.positive_definite(Sigma), Q.symmetric(Sigma),
               Q.positive_definite(R), Q.symmetric(R), Q.fullrank(H)]

f = fortran_function([mu, Sigma, H, R, data], [newmu, newSigma], *assumptions)
```



Kalman Filter - Theano v. Fortran

```
newmu       = mu + Sigma*H.T * (R + H*Sigma*H.T).I * (H*mu - data)
newSigma    = Sigma - Sigma*H.T * (R + H*Sigma*H.T).I * H * Sigma
```

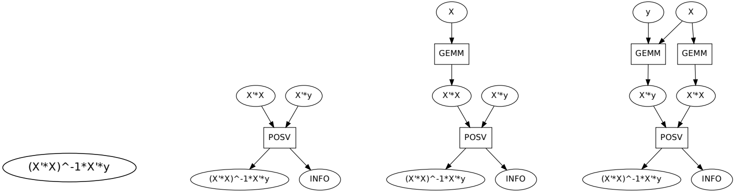
```
f = theano_function( [mu, Sigma, H, R, data], [newmu, newSigma])
```



Final Thoughts

- Modularity is good!
  - Cater to single-field experts
  - Eases comparison and evolution
  - This project might die but the parts will survive
- Intermediate Representations are Good!
  - Fortran code doesn't depend on Python
  - Readability encourages development
  - Extensibility (lets generate CUDA)
- Read more! `http://matthewrocklin.com/blog`
- Listen more!
  - Dynamics with SymPy Mechanics, *Jason Moore*,
    Room 204 - 2:10pm
  - SymPy Gamma and SymPy Live: Python and Mathematics Online, *David Li*
    Room 203 - 3:50pm

Multiple Results



Multiple Results