

# Generating and Verifying Aerospace Software

Ewen Denney<sup>+</sup>, Bernd Fischer\*, Johann Schumann\*

[fisch@email.arc.nasa.gov](mailto:fisch@email.arc.nasa.gov)

Automated Software Engineering Group

<sup>+</sup>QSS / \*RIACS, NASA Ames Research Center



Funded by CICT ITSR and IS Programs

# GN&C – Guidance, Navigation, and Control

Applications:

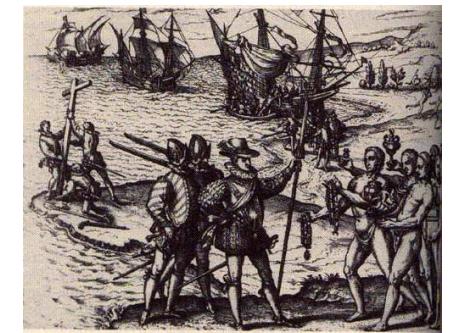
*Spacecraft, aircraft, ships, and (increasingly) cars require methods for the accurate determination of position and attitude*

Equipment:

- compass, clock, triangulation (e.g., star tracker), GPS, INS
- radio navigation (VOR, DME, Radar)

Problems:

- measurements are not precise (“noisy”)
- each measurement contributes partial information
- sensor failures (degradation / transient / permanent)



Overall task:

*Calculate the optimal position estimate using all available information*

# High-Profile GN&C Software Failures



- Mars Polar Lander
  - inconsistent sensor usage
  - crashed into Mars
- Mars Climate Orbiter
  - unit problem
  - crashed into Mars
- Gemini 5
  - modeling error
  - missed landing point by 100 miles
- Soyuz TMA-1
  - “software glitch”
  - missed landing point by 275 miles
- Ariane 5
  - counter overflow
  - self-destructed

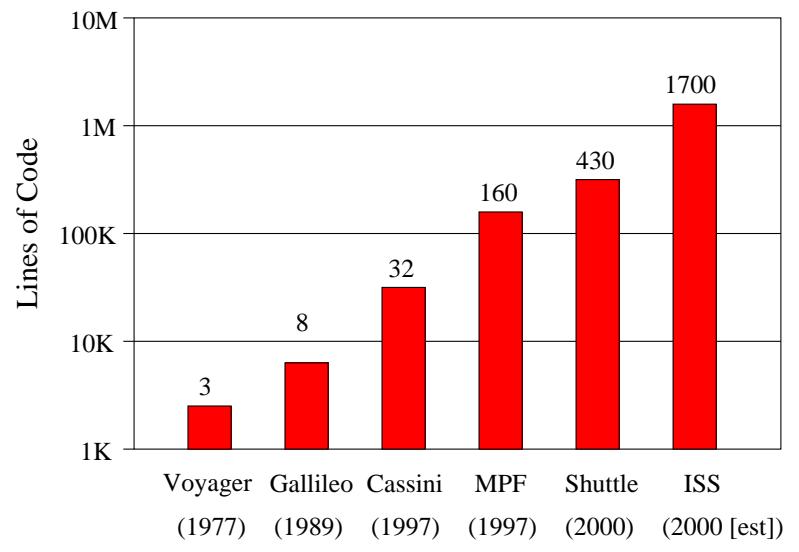
# GN&C Software Development Dilemma

## Problem:

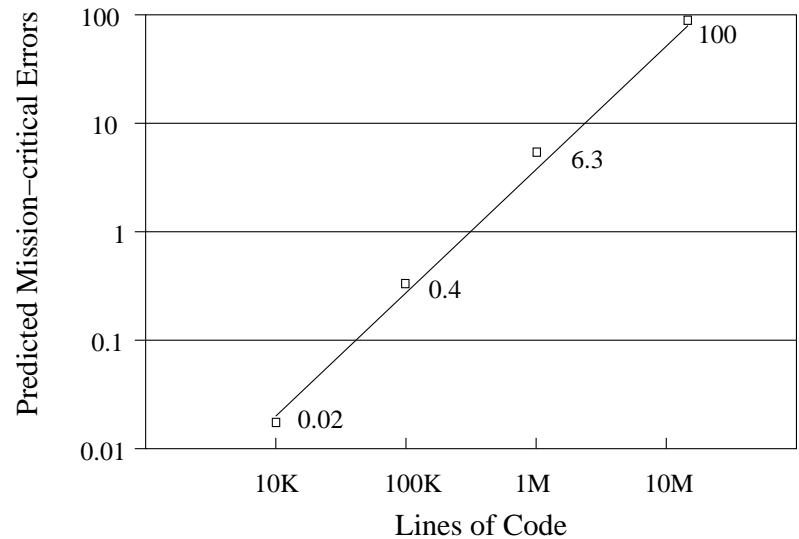
- GN&C software is integral part of all missions but is becoming
  - more important
  - more complex
  - more error-prone
  - more expensive
- Cross-mission reuse of software difficult
- High-profile mission failures

## Solution:

automatic program synthesis  
(i.e., computer writes programs)



Source: Defense Science Board



# Automatic Program Synthesis

**Automatic Program Synthesis:** derivation of efficient, high-quality programs from compact, high-level specifications

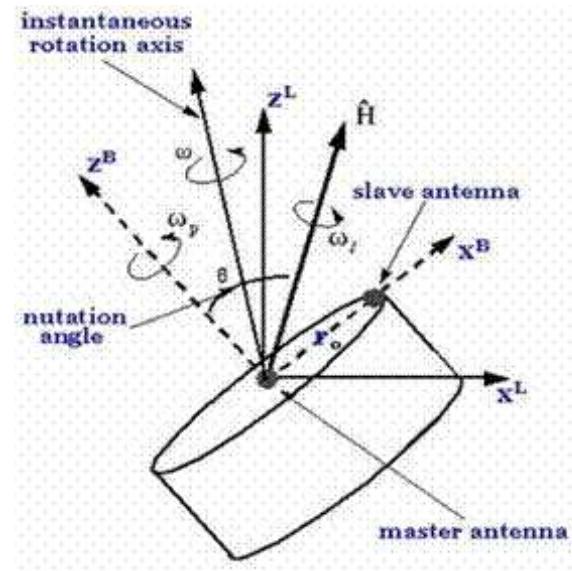
**input:** small specifications ( $\approx 10\text{--}50$  lines)

**output:** optimized, documented C/C++ code (up to 10000 lines)

**Program Synthesis System:** Really smart compiler

## Advantages of Program Synthesis

- Fast turn-around of software revisions
- Fast exploration of design space
- Incorporation of domain knowledge
- Better algorithms, better data types
- Generation of extensive documentation



# Automatic Program Synthesis

**Automatic Program Synthesis:** derivation of efficient, high-quality programs from compact, high-level specifications

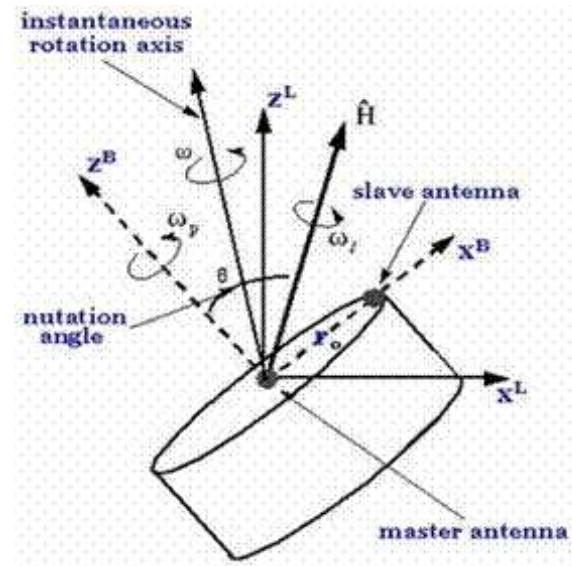
**input:** small specifications ( $\approx 10\text{--}50$  lines)

**output:** optimized, documented C/C++ code (up to 10000 lines)

**Program Synthesis System:** Really smart compiler

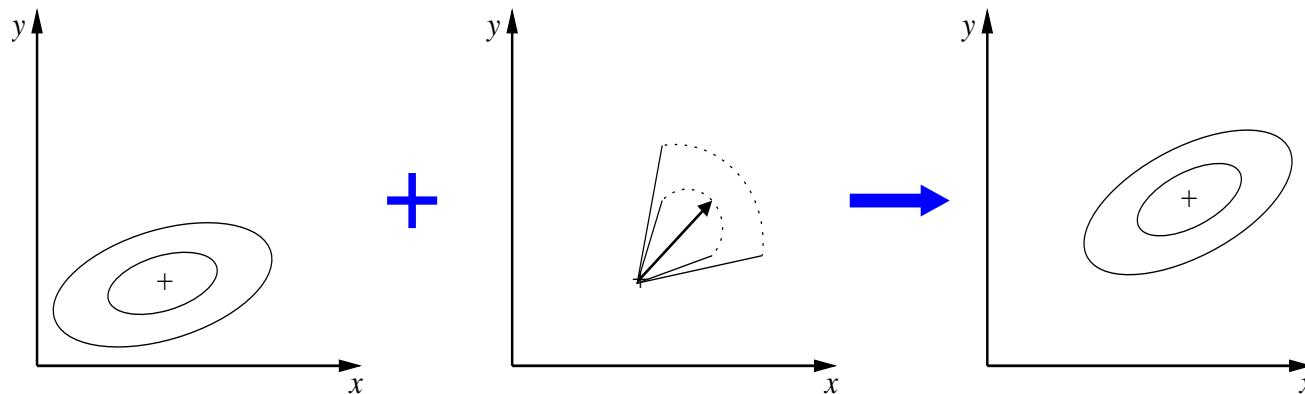
## Disadvantages of Program Synthesis

- Difficult



# Kalman Filters: Idea

- Developed 1960 by R.E. Kalman
- Standard GN&C algorithm
- Basic idea:
  - state estimate is updated every time step
  - state updates are linear with noise
  - measurements have noise
  - each measurement contributes information
  - state and measurement are combined (using statistics)



# Kalman Filters: Mathematical Model

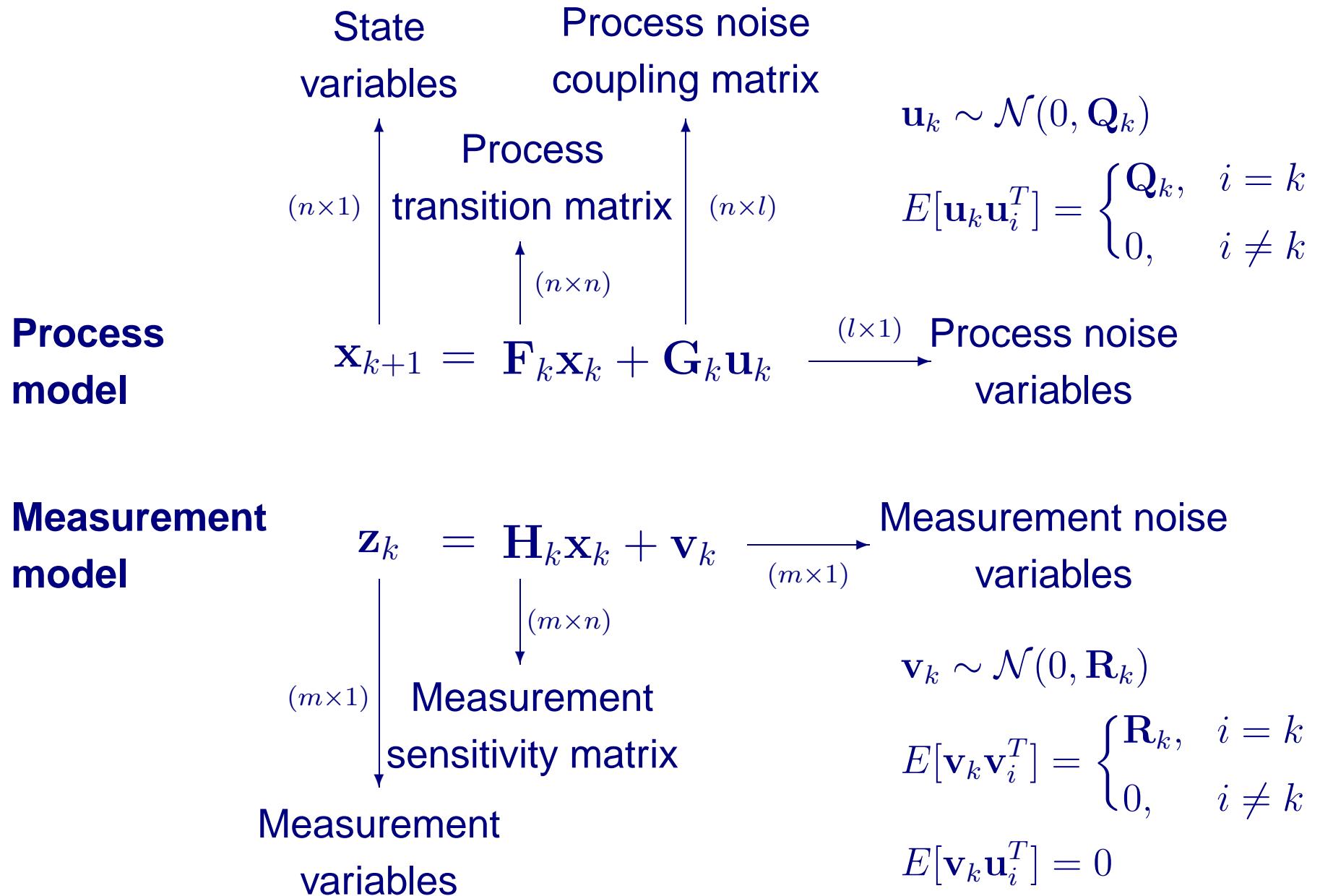
**Process  
model**

$$\mathbf{x}_{k+1} = \mathbf{F}_k \mathbf{x}_k + \mathbf{G}_k \mathbf{u}_k$$

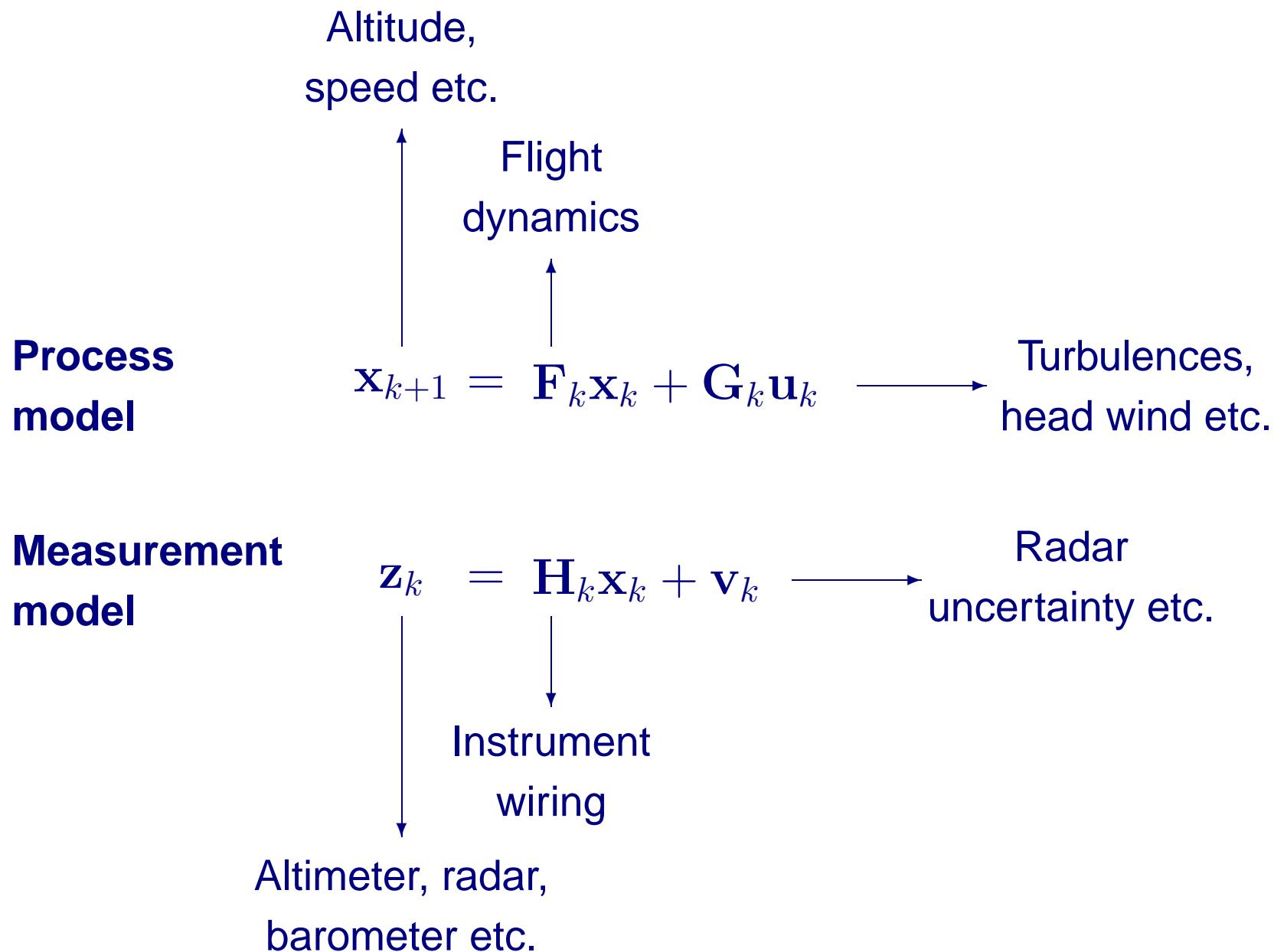
**Measurement  
model**

$$\mathbf{z}_k = \mathbf{H}_k \mathbf{x}_k + \mathbf{v}_k$$

# Kalman Filters: Mathematical Model



# Kalman Filters: Mathematical Model



# Kalman Filters: Algorithm

1. Initialization: initialize all vectors and matrices
2. Measurement update: *read and process measurement z*

$$\mathbf{K} = \mathbf{P}^{-} \mathbf{H}^T (\mathbf{H} \mathbf{P}^{-} \mathbf{H}^T + \mathbf{R})^{-1} \xleftarrow{(n \times m)} \text{Kalman-Gain}$$

$$\hat{\mathbf{x}}^{+} = \hat{\mathbf{x}}^{-} + \mathbf{K}(\mathbf{z} - \mathbf{H}\hat{\mathbf{x}}^{-}) \xleftarrow{(n \times 1)} \text{(posterior) state estimate}$$

$$\mathbf{P}^{+} = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}^{-} \xleftarrow{(n \times n)} \text{covariance matrix ("errorbar")}$$

3. Temporal Update: *estimate one step ahead in time*

$$\hat{\mathbf{x}}^{-} = \mathbf{F}\hat{\mathbf{x}}^{+}$$

$$\mathbf{P}^{-} = \mathbf{F}\mathbf{P}^{+}\mathbf{F}^T + \mathbf{Q}$$

4. Go to 2

Notation:

$\mathbf{M}^T \quad \hat{\quad} \text{matrix transposition}$

$\mathbf{M}^{-} / \mathbf{M}^{+} \quad \hat{\quad} \text{prior / posterior value (wrt. measurement update)}$

$\hat{\mathbf{x}} \quad \hat{\quad} \text{estimate}$

# Kalman Filters: Algorithm

1. Initialization: initialize all vectors and matrices
2. Measurement update: *read and process measurement z*

$$\mathbf{K} = \mathbf{P}^{-} \mathbf{H}^T (\mathbf{H} \mathbf{P}^{-} \mathbf{H}^T + \mathbf{R})^{-1}$$

$$\hat{\mathbf{x}}^{+} = \hat{\mathbf{x}}^{-} + \mathbf{K}(\mathbf{z} - \mathbf{H}\hat{\mathbf{x}}^{-})$$

$$\mathbf{P}^{+} = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}^{-}$$

3. Temporal Update: *estimate one step ahead in time*

$$\hat{\mathbf{x}}^{-} = \mathbf{F}\hat{\mathbf{x}}^{+}$$

$$\mathbf{P}^{-} = \mathbf{F}\mathbf{P}^{+}\mathbf{F}^T + \mathbf{Q}$$

4. Go to 2

- ⇒ time-varying case more complicated
- ⇒ continuous case more complicated
- ⇒ numerous variations

# AutoFilter

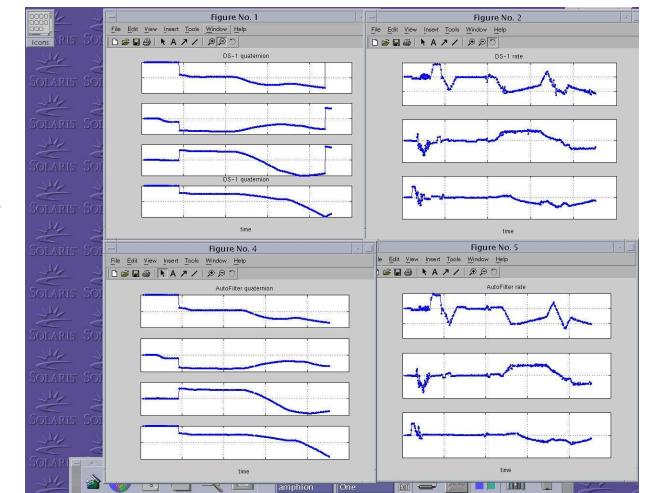
AutoFilter in a nutshell:

- program synthesis system for state estimation
- built on generic program synthesis framework
- domain-specific modeling language
- generates C/C++ code



Case study: Deep Space One (with JPL)

- ARC & JPL extracted mathematical model
  - ARC generated C code linking to JPL libraries
  - JPL tested code against original DS1 code  
(Autonomy testbed)
- ⇒ generated code passed all tests



# AutoFilter: DS1-Specification

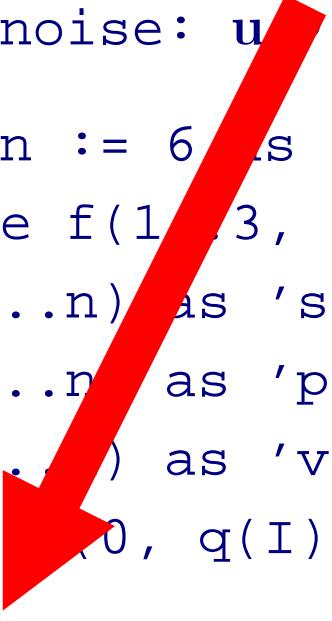
```
% Process model:  $\dot{\mathbf{x}} = \mathbf{F}_t \mathbf{x} + \mathbf{u}$ 
% Process noise:  $\mathbf{u} \sim \mathcal{N}(0, \mathbf{q} \cdot \mathbf{I}) \Leftrightarrow u_i \sim \mathcal{N}(0, q_i)$ 

const nat n := 6 as 'Number of state variables'.
data double f(1..3, time) as 'gyro readings'.
double x(1..n) as 'state variable vector'.
double u(1..n) as 'process noise vector'.
double q(1..n) as 'variance of process noise'.
u(I) ~ gauss(0, q(I)).

equations process_eqs are [
    dot x(1) := (hat x(4) - x(4)) - u(1)
                + x(2) * (f(3,t) - hat x(6))
                - x(3) * (f(2,t) - hat x(5)),
    ...
    dot x(6) := u(6)
].
```

## AutoFilter: DS1-Specification

```
% Process model:  $\dot{\mathbf{x}} = \mathbf{F}_t \mathbf{x} + \mathbf{u}$ 
% Process noise:  $\mathbf{u} \sim \mathcal{N}(0, \mathbf{q} \cdot \mathbf{I}) \Leftrightarrow u_i \sim \mathcal{N}(0, q_i)$ 

const nat n := 6 as 'Number of state variables'.
data double f(1..3, time) as 'gyro readings'.
double x(1..n) as 'state variable vector'.
double u(1..n) as 'process noise vector'.
double q(1..n) as 'variance of process noise'.
u(I) ~ gau(0, q(I)).  
  
equations process_eqs are [
    dot x(1) := (hat x(4) - x(4)) - u(1)
        + x(2) * (f(3,t) - hat x(6))
        - x(3) * (f(2,t) - hat x(5)),
    ...
    dot x(6) := u(6)
].
```

Specification  $\hat{=}$  Math + Declarations

# AutoFilter: DS1-Specification

```
const double delta := 1/400 as 'Sampling Interval'.
units delta in seconds.

estimator ds1_filter.

ds1_filter::process_model      ::= process_eqs.
ds1_filter::measurement_model ::= measurement_eqs.
ds1_filter::steps              ::= 24000.
ds1_filter::update_interval    ::= delta.
ds1_filter::initials          ::= xinit(_).

output ds1_filter.
```

Specification of desired software architecture:

- parameters and initial values
- interface adaptation
- units, coordinate systems, and frames

# AutoFilter: DS1-Code

```

//-----  

// Code file generated by AutoFilter V0.0.1  

// (c) 1999-2004 ASE Group, NASA Ames Research Center  

// Problem: IMU + SRU; nonlinear w/ quaternions  

// Source: examples/certification/quaternion.ab  

// Command: ./autofilter  

//          examples/certification/quaternion.ab  

// Generated: Wed Sep 15 18:16:41 2004  

//-----  

#include "autobayes.h"  

//-----  

// Octave Function: quaternion_dsl  

//-----  

DEFUN_DLD(quaternion_dsl,input_args,output_args, "")  

{  

    octave_value_list retval;  

    if (input_args.length () != 7 || output_args != 1){  

        octave_stdout << "error"  

        return retval;  

    }  

    //-- Input declarations -----  

    // standard deviation of measurement noise  

octave_value_arg_rho = input_args(0);  

if (!arg_rho.is_real_matrix() || arg_rho.columns() != 1){  

    gripe_wrong_type_arg("rho", (const string &) "ColumnVector expected");  

    return retval;  

}  

ColumnVector rho = (ColumnVector)(arg_rho.vector_value());  

    // standard deviation of process noise  

octave_value_arg_sigma = input_args(1);  

if (!arg_sigma.is_real_matrix() || arg_sigma.columns() != 1){  

    gripe_wrong_type_arg("sigma", (const string &) "ColumnVector expected");  

    return retval;  

}  

ColumnVector sigma = (ColumnVector)(arg_sigma.vector_value());  

    // gyro readings  

octave_value_arg_u = input_args(2);  

if (!arg_u.is_real_matrix()){  

    gripe_wrong_type_arg("u", (const string &) "Matrix expected");  

    return retval;  

}  

Matrix u = (Matrix)(arg_u.matrix_value());  

    // initial state variable values  

octave_value_arg_xinit = input_args(3);  

if (!arg_xinit.is_real_matrix() || arg_xinit.columns() != 1){  

    gripe_wrong_type_arg("xinit", (const string &) "ColumnVector expected");  

    return retval;  

}  

ColumnVector xinit = (ColumnVector)(arg_xinit.vector_value());  

    // initial value means  

octave_value_arg_xinit_mean = input_args(4);  

if (!arg_xinit_mean.is_real_matrix() || arg_xinit_mean.columns() != 1){  

    gripe_wrong_type_arg("xinit_mean", (const string &) "ColumnVector expected");  

    return retval;  

}  

ColumnVector xinit_mean = (ColumnVector)(arg_xinit_mean.vector_value());  

    // initial state noise  

octave_value_arg_xinit_noise = input_args(5);  

if (!arg_xinit_noise.is_real_matrix() || arg_xinit_noise.columns() != 1){  

    gripe_wrong_type_arg("xinit_noise", (const string &) "ColumnVector expected");  

    return retval;  

}  

ColumnVector xinit_noise = (ColumnVector)(arg_xinit_noise.vector_value());  

    // SRU measurements  

octave_value_arg_z = input_args(6);  

if (!arg_z.is_real_matrix()){  

    gripe_wrong_type_arg("z", (const string &) "Matrix expected");  

    return retval;  

}  

Matrix z = (Matrix)(arg_z.matrix_value());  

    //-- Constant declarations -----  

    // Number of measurements  

int n_measvars = 3;  

    // Number of state variables  

int n_statevars = 6;  

    // Number of iteration steps  

int n_steps = 999;  

    // Sampling Interval  

double t = (double)(1) / (double)(400);  

    //-- Output declarations -----  

    // Output vector  

Matrix xhat_dsl_filter(6, n_steps);  

    // Local declarations -----  

double dsl_filter;  

    // process noise vector  

ColumnVector etta(n_statevars);  

double measurement_model;  

double process_model;  

    // measurement noise vector  

ColumnVector v_m(measvars);  

    // state variables vector (error values)  

ColumnVector x(n_statevars);
    // Loop variable  

int pv41;  

    // Vector to hold predicted measurements on each iteration  

Matrix zpred_dsl_filter(m_measvars, 1);  

    // Declare State Transition matrix  

Matrix phi_dsl_filter(n_statevars, n_statevars);  

    // Declare Driving Input  

Matrix dv_dsl_filter(n_statevars, 1);  

    // Declare process noise covariance matrix  

Matrix q_dsl_filter(n_statevars, n_statevars);  

    // Declare measurement noise covariance matrix  

Matrix r_dsl_filter(m_measvars, m_measvars);  

    // Declare measurement matrix  

Matrix h_dsl_filter(m_measvars, n_statevars);  

    // Declare process covariance matrix prior  

Matrix pinminus_dsl_filter(n_statevars, n_statevars);  

    // Declare prior state estimate  

Matrix xhatmin_dsl_filter(n_statevars, 1);  

    // Declare posterior state estimate  

Matrix xhat1_dsl_filter(n_statevars, 1);  

    // Declare Kalman Gain  

Matrix gain_dsl_filter(n_statevars, m_measvars);  

    // Declare identity matrix  

Matrix id_dsl_filter(n_statevars, n_statevars);  

    // Vector to hold measurements on each iteration  

Matrix zhat_dsl_filter(m_measvars, 1);  

    // Initialize Driver Input  

dv_dsl_filter(0, 0) = xinit(3);  

dv_dsl_filter(1, 0) = xinit(4);  

dv_dsl_filter(2, 0) = xinit(5);  

dv_dsl_filter(3, 0) = 0;  

dv_dsl_filter(4, 0) = 0;  

dv_dsl_filter(5, 0) = 0;  

    // Initialize process covariance noise matrix  

q_dsl_filter(0, 0) = t * sigma(0);  

q_dsl_filter(0, 1) = 0;  

q_dsl_filter(0, 2) = 0;  

q_dsl_filter(0, 3) = 0;  

q_dsl_filter(0, 4) = 0;  

q_dsl_filter(0, 5) = 0;  

q_dsl_filter(1, 0) = 0;  

q_dsl_filter(1, 1) = t * sigma(1);  

q_dsl_filter(1, 2) = 0;  

q_dsl_filter(1, 3) = 0;  

q_dsl_filter(1, 4) = 0;  

q_dsl_filter(1, 5) = 0;  

q_dsl_filter(2, 0) = 0;  

q_dsl_filter(2, 1) = 0;  

q_dsl_filter(2, 2) = t * sigma(2);  

q_dsl_filter(2, 3) = 0;  

q_dsl_filter(2, 4) = 0;  

q_dsl_filter(2, 5) = 0;  

q_dsl_filter(3, 0) = 0;  

q_dsl_filter(3, 1) = 0;  

q_dsl_filter(3, 2) = 0;  

q_dsl_filter(3, 3) = t * sigma(3);  

q_dsl_filter(3, 4) = 0;  

q_dsl_filter(3, 5) = 0;  

q_dsl_filter(4, 0) = 0;  

q_dsl_filter(4, 1) = 0;  

q_dsl_filter(4, 2) = 0;  

q_dsl_filter(4, 3) = 0;  

q_dsl_filter(4, 4) = t * sigma(4);  

q_dsl_filter(4, 5) = 0;  

q_dsl_filter(5, 0) = 0;  

q_dsl_filter(5, 1) = 0;  

q_dsl_filter(5, 2) = 0;  

q_dsl_filter(5, 3) = 0;  

q_dsl_filter(5, 4) = 0;  

q_dsl_filter(5, 5) = t * sigma(5);  

    // Initialize measurement covariance noise matrix  

r_dsl_filter(0, 0) = rho(0);  

r_dsl_filter(0, 1) = 0;  

r_dsl_filter(0, 2) = 0;  

r_dsl_filter(0, 3) = 0;  

r_dsl_filter(0, 4) = 0;  

r_dsl_filter(0, 5) = 0;  

r_dsl_filter(1, 0) = 0;  

r_dsl_filter(1, 1) = rho(1);  

r_dsl_filter(1, 2) = 0;  

r_dsl_filter(1, 3) = 0;  

r_dsl_filter(1, 4) = 0;  

r_dsl_filter(1, 5) = 0;  

r_dsl_filter(2, 0) = 0;  

r_dsl_filter(2, 1) = rho(2);  

r_dsl_filter(2, 2) = rho(2);  

    // Set initial estimate  

xhatmin_dsl_filter(0, 0) = xinit_mean(0);  

xhatmin_dsl_filter(1, 0) = xinit_mean(1);  

xhatmin_dsl_filter(2, 0) = xinit_mean(2);  

xhatmin_dsl_filter(3, 0) = xinit_mean(3);  

xhatmin_dsl_filter(4, 0) = xinit_mean(4);  

xhatmin_dsl_filter(5, 0) = xinit_mean(5);  

    // Set initial process covariance matrix  

pinminus_dsl_filter(0, 0) = xinit_noise(0);  

pinminus_dsl_filter(0, 1) = 0;  

pinminus_dsl_filter(0, 2) = 0;  

pinminus_dsl_filter(0, 3) = 0;  

pinminus_dsl_filter(0, 4) = 0;  

pinminus_dsl_filter(0, 5) = 0;  

pinminus_dsl_filter(1, 0) = 0;  

pinminus_dsl_filter(1, 1) = xinit_noise(1);  

pinminus_dsl_filter(1, 2) = 0;  

pinminus_dsl_filter(1, 3) = 0;  

pinminus_dsl_filter(1, 4) = 0;  

pinminus_dsl_filter(1, 5) = 0;  

pinminus_dsl_filter(2, 0) = 0;  

pinminus_dsl_filter(2, 1) = xinit_noise(2);  

pinminus_dsl_filter(2, 2) = 0;  

pinminus_dsl_filter(2, 3) = 0;  

pinminus_dsl_filter(2, 4) = 0;  

pinminus_dsl_filter(2, 5) = 0;
    // Kalman Filter Loop  

    // Extended Kalman Filter  

for( pV6 = 0;pV6 <= n_steps - 1;pV6++ )  

{
    // Initialize identity matrix  

    // Definition of ID matrix  

for( pV49 = 0;pV49 <= n_statevars - 1;pV49++ )  

    for( pV50 = 0;pV50 <= n_statevars - 1;pV50++ )  

        if ( pV49 == pV50 )  

            id_dsl_filter(pV49, pV50) = 1;  

        else  

            id_dsl_filter(pV49, pV50) = 0;
    zpred_dsl_filter(0, 0) = xhatmin_dsl_filter(0, 0);  

zpred_dsl_filter(1, 0) = xhatmin_dsl_filter(1, 0);  

zpred_dsl_filter(2, 0) = xhatmin_dsl_filter(2, 0);  

zhat_dsl_filter(0, 0) = z(0,pV5);  

zhat_dsl_filter(1, 0) = z(1,pV5);  

zhat_dsl_filter(2, 0) = z(2,pV5);  

    // Update loop dependent quantities  

if ( pV5 > 0 )  

{
    phi_dsl_filter(2, 1) = t * (xhatmin_dsl_filter(3, 0) - u(0, pV5));  

phi_dsl_filter(2, 0) = t * (u(1, pV5) - xhatmin_dsl_filter(4, 0));  

phi_dsl_filter(1, 2) = t * (u(0, pV5) - xhatmin_dsl_filter(3, 0));  

phi_dsl_filter(1, 0) = t * (xhatmin_dsl_filter(5, 0) - u(2, pV5));  

phi_dsl_filter(0, 2) = t * (xhatmin_dsl_filter(4, 0) - u(1, pV5));  

phi_dsl_filter(0, 1) = t * (u(2, pV5) - xhatmin_dsl_filter(5, 0));  

}
else
{
    // Update loop dependent quantities  

if ( pV5 > 0 )
{
    dv_dsl_filter(2, 0) = xhatmin_dsl_filter(5, 0);  

dv_dsl_filter(1, 0) = xhatmin_dsl_filter(4, 0);  

dv_dsl_filter(0, 0) = xhatmin_dsl_filter(3, 0);  

}
else
{
    gain_dsl_filter = pinminus_dsl_filter *  

        ((h_dsl_filter).transpose()) *  

        (r_dsl_filter +  

        h_dsl_filter *  

        (pinminus_dsl_filter *  

        (h_dsl_filter).transpose())).inverse();  

xhat1_dsl_filter = xhatmin_dsl_filter +  

    gain_dsl_filter *  

        ((zhat_dsl_filter - zpred_dsl_filter));  

pplus_dsl_filter = (id_dsl_filter - gain_dsl_filter * h_dsl_filter) *  

    pinminus_dsl_filter;  

xhatmin_dsl_filter = dv_dsl_filter + phi_dsl_filter * xhat1_dsl_filter;  

pinminus_dsl_filter = q_dsl_filter +  

    (pplus_dsl_filter * (phi_dsl_filter).transpose());  

    // Populate Output Vector  

for( pV39 = 0;pV39 <= n_statevars - 1;pV39++ )  

    xhat_dsl_filter(pV39, pV6) = xhat1_dsl_filter(pV39, 0);
}
retval.resize(1);  

retval(0) = xhat_dsl_filter;  

return retval;
} //-- End of code -----

```

# AutoFilter: DS1-Code

• 434 lines C++-Code

• deeply nested loops

• fully documented ( $\approx$  50% comments)

•  $\approx$  1:10 leverage from specification

•  $\approx$  1.0 sec. synthesis time

```

//-----  

// Code file generated by AutoFilter V0.0.1  

// (c) 1999-2004 ASE Group, NASA Ames Research Center  

// Problem: IMU + SRU; nonlinear w/ quaternions  

// Source: examples/certification/quaternion.ab  

// Command: ./autofilter  

//          examples/certification/quaternion_ab  

// Generated: Wed Sep 15 18:18:56 2004  

//-----  

#include "autobeyes.h"  

//-----  

// Octave Function: quaternion_dsl  

//-----  

DEFUN_DLD(quaternion_dsl, input_args, output_args, "")  

{  

    octave_value_list retval;  

    if (input_args.length() == 0)  

        octave_stdout << "error"  

    return retval;  

}  

//-- Input declarations -----  

// standard deviation of measurement noise  

octave_value_arg_rho = input_args(0);  

if (!arg_rho.is_real_matrix() || arg_rho.columns() != 1){  

    gripe_wrong_type_arg("rho", (const string *)&"ColumnVector expected");  

    return retval;  

}  

ColumnVector rho = (ColumnVector)(arg_rho.vector_value());  

// standard deviation of process noise  

octave_value_arg_sigma = input_args(1);  

if (!arg_sigma.is_real_matrix() || arg_sigma.columns() != 1){  

    gripe_wrong_type_arg("sigma", (const string *)&"ColumnVector expected");  

    return retval;  

}  

ColumnVector sigma = (ColumnVector)(arg_sigma.vector_value());  

// gyro readings  

octave_value_arg_u = input_args(2);  

if (!arg_u.is_real_matrix()) {  

    gripe_wrong_type_arg("u", (const string *)&"ColumnVector expected");  

    return retval;  

}  

Matrix u = (Matrix)(arg_u.matrix_value());  

// initial state variable values  

octave_value_arg_xinit = input_args(3);  

if (!arg_xinit.is_real_matrix() || arg_xinit.columns() != 1){  

    gripe_wrong_type_arg("xinit", (const string *)&"ColumnVector expected");  

    return retval;  

}  

ColumnVector xinit = (ColumnVector)(arg_xinit.vector_value());  

// initial value means  

octave_value_arg_xinit_mean = input_args(4);  

if (!arg_xinit_mean.is_real_matrix() || arg_xinit_mean.columns() != 1){  

    gripe_wrong_type_arg("xinit_mean", (const string *)&"ColumnVector expected");  

    return retval;  

}  

ColumnVector xinit_mean = (ColumnVector)(arg_xinit_mean.vector_value());  

// initial state noise  

octave_value_arg_xinit_noise = input_args(5);  

if (!arg_xinit_noise.is_real_matrix() || arg_xinit_noise.columns() != 1){  

    gripe_wrong_type_arg("xinit_noise", (const string *)&"ColumnVector expected");  

    return retval;  

}  

Matrix xinit_noise = (Matrix)(arg_xinit_noise.matrix_value());  

// SRU measurements  

octave_value_arg_z = input_args(6);  

if (!arg_z.is_real_matrix()) {  

    gripe_wrong_type_arg("z", (const string *)&"Matrix expected");  

    return retval;  

}  

Matrix z = (Matrix)(arg_z.matrix_value());  

//-- Constant declarations -----  

// Number of measurements  

int m_measvars = 3;  

// Number of state variables  

int n_statevars = 6;  

// Number of iteration steps  

int n_steps = 999;  

// Sampling Interval  

double t = (double)(1) / (double)(400);  

//-- Output declarations -----  

// Output vector  

Matrix xhat_dsl_filter(6, n_steps);  

//-- Local declarations -----  

double dsl_filter;  

// process noise vector  

ColumnVector etn(n_statevars);  

double measurement_model;  

double process_model;  

// measurement noise vector  

ColumnVector v(m_measvars);  

// State variables vector (error values)  

ColumnVector x(n_statevars);  

//-----  

// Loop variable  

int pv41;  

// Vector to hold predicted measurements on each iteration  

Matrix zpred_dsl_filter(m_measvars, 1);  

// Declare State Transition matrix  

Matrix phi_dsl_filter(n_statevars, n_statevars);  

// Declaring Drivings set  

Matrix dv_dsl_filter(n_statevars, n_statevars, 1);  

// Declaring Process Covariance matrix  

Matrix q_dsl_filter(n_statevars, n_statevars);  

// Declare measurement noise covariance matrix  

Matrix r_dsl_filter(m_measvars, m_measvars);  

// Declare measurement matrix  

Matrix h_dsl_filter(m_measvars, n_statevars);  

// Declare prior state estimate  

Matrix xhatmin_dsl_filter(n_statevars, 1);  

// Declare posterior state estimate  

Matrix xhat_dsl_filter(n_statevars, 1);  

// Declare Kalman Gain  

Matrix id_dsl_filter(n_statevars, n_statevars);  

// Declare process covariance matrix prior  

Matrix pplus_dsl_filter(n_statevars, n_statevars);  

// Declare loop dependent quantities  

Matrix id_dsl_filter(n_statevars, n_statevars);  

// Declare prior state estimate  

Matrix xhatmin_dsl_filter(n_statevars, 1);  

// standard deviation of measurement noise  

octave_value_arg_rho = input_args(0);  

if (!arg_rho.is_real_matrix() || arg_rho.columns() != 1){  

    gripe_wrong_type_arg("rho", (const string *)&"ColumnVector expected");  

    return retval;  

}  

ColumnVector rho = (ColumnVector)(arg_rho.vector_value());  

// standard deviation of process noise  

octave_value_arg_sigma = input_args(1);  

if (!arg_sigma.is_real_matrix() || arg_sigma.columns() != 1){  

    gripe_wrong_type_arg("sigma", (const string *)&"ColumnVector expected");  

    return retval;  

}  

ColumnVector sigma = (ColumnVector)(arg_sigma.vector_value());  

// gyro readings  

octave_value_arg_u = input_args(2);  

if (!arg_u.is_real_matrix()) {  

    gripe_wrong_type_arg("u", (const string *)&"ColumnVector expected");  

    return retval;  

}  

Matrix u = (Matrix)(arg_u.matrix_value());  

// initial state variable values  

octave_value_arg_xinit = input_args(3);  

if (!arg_xinit.is_real_matrix() || arg_xinit.columns() != 1){  

    gripe_wrong_type_arg("xinit", (const string *)&"ColumnVector expected");  

    return retval;  

}  

ColumnVector xinit = (ColumnVector)(arg_xinit.vector_value());  

// initial value means  

octave_value_arg_xinit_mean = input_args(4);  

if (!arg_xinit_mean.is_real_matrix() || arg_xinit_mean.columns() != 1){  

    gripe_wrong_type_arg("xinit_mean", (const string *)&"ColumnVector expected");  

    return retval;  

}  

ColumnVector xinit_mean = (ColumnVector)(arg_xinit_mean.vector_value());  

// initial state noise  

octave_value_arg_xinit_noise = input_args(5);  

if (!arg_xinit_noise.is_real_matrix() || arg_xinit_noise.columns() != 1){  

    gripe_wrong_type_arg("xinit_noise", (const string *)&"ColumnVector expected");  

    return retval;  

}  

Matrix xinit_noise = (Matrix)(arg_xinit_noise.matrix_value());  

// SRU measurements  

octave_value_arg_z = input_args(6);  

if (!arg_z.is_real_matrix()) {  

    gripe_wrong_type_arg("z", (const string *)&"Matrix expected");  

    return retval;  

}  

Matrix z = (Matrix)(arg_z.matrix_value());  

//-- Constant declarations -----  

// Number of measurements  

int m_measvars = 3;  

// Number of state variables  

int n_statevars = 6;  

// Number of iteration steps  

int n_steps = 999;  

// Sampling Interval  

double t = (double)(1) / (double)(400);  

//-- Output declarations -----  

// Output vector  

Matrix xhat_dsl_filter(6, n_steps);  

//-- Local declarations -----  

double dsl_filter;  

// process noise vector  

ColumnVector etn(n_statevars);  

double measurement_model;  

double process_model;  

// measurement noise vector  

ColumnVector v(m_measvars);  

// State variables vector (error values)  

ColumnVector x(n_statevars);  

//-----  

// Loop variable  

int pv41;  

// Vector to hold predicted measurements on each iteration  

Matrix zpred_dsl_filter(m_measvars, 1);  

// Declare State Transition matrix  

Matrix phi_dsl_filter(n_statevars, n_statevars);  

// Declare loop dependent quantities  

Matrix id_dsl_filter(n_statevars, n_statevars);  

// Declaring Drivings set  

Matrix dv_dsl_filter(n_statevars, n_statevars, 1);  

// Declaring Process Covariance matrix  

Matrix q_dsl_filter(n_statevars, n_statevars);  

// Declare measurement noise covariance matrix  

Matrix r_dsl_filter(m_measvars, m_measvars);  

// Declare measurement matrix  

Matrix h_dsl_filter(m_measvars, n_statevars);  

// Declare prior state estimate  

Matrix xhatmin_dsl_filter(n_statevars, 1);  

// Declare posterior state estimate  

Matrix xhat_dsl_filter(n_statevars, 1);  

// Declare Kalman Gain  

Matrix id_dsl_filter(n_statevars, n_statevars);  

// Declare process covariance matrix prior  

Matrix pplus_dsl_filter(n_statevars, n_statevars);  

// Declare loop dependent quantities  

Matrix id_dsl_filter(n_statevars, n_statevars);  

// Initialize Driver Input  

dv_dsl_filter(0, 0) = xinit(3);  

dv_dsl_filter(1, 0) = xinit(4);  

dv_dsl_filter(2, 0) = xinit(5);  

dv_dsl_filter(3, 0) = 0;  

dv_dsl_filter(4, 0) = 0;  

dv_dsl_filter(5, 0) = 0;  

dv_dsl_filter(6, 0) = 0;  

// Initialize process covariance noise matrix  

q_dsl_filter(0, 0) = t * sigma(0);  

q_dsl_filter(0, 1) = 0;  

q_dsl_filter(0, 2) = 0;  

q_dsl_filter(0, 3) = 0;  

q_dsl_filter(0, 4) = 0;  

q_dsl_filter(0, 5) = 0;  

q_dsl_filter(1, 0) = 0;  

q_dsl_filter(1, 1) = t * sigma(1);  

q_dsl_filter(1, 2) = 0;  

q_dsl_filter(1, 3) = 0;  

q_dsl_filter(1, 4) = 0;  

q_dsl_filter(1, 5) = 0;  

q_dsl_filter(2, 0) = 0;  

q_dsl_filter(2, 1) = 0;  

q_dsl_filter(2, 2) = t * sigma(2);  

q_dsl_filter(2, 3) = 0;  

q_dsl_filter(2, 4) = 0;  

q_dsl_filter(2, 5) = 0;  

q_dsl_filter(3, 0) = 0;  

q_dsl_filter(3, 1) = 0;  

q_dsl_filter(3, 2) = 0;  

q_dsl_filter(3, 3) = t * sigma(3);  

q_dsl_filter(3, 4) = 0;  

q_dsl_filter(3, 5) = 0;  

q_dsl_filter(4, 0) = 0;  

q_dsl_filter(4, 1) = 0;  

q_dsl_filter(4, 2) = 0;  

q_dsl_filter(4, 3) = 0;  

q_dsl_filter(4, 4) = t * sigma(4);  

q_dsl_filter(4, 5) = 0;  

q_dsl_filter(5, 0) = 0;  

q_dsl_filter(5, 1) = 0;  

q_dsl_filter(5, 2) = 0;  

q_dsl_filter(5, 3) = 0;  

q_dsl_filter(5, 4) = 0;  

q_dsl_filter(5, 5) = t * sigma(5);  

// Initialize measurement covariance noise matrix  

r_dsl_filter(0, 0) = rho(0, 0);  

r_dsl_filter(0, 1) = 0;  

r_dsl_filter(0, 2) = 0;  

r_dsl_filter(0, 3) = 0;  

r_dsl_filter(0, 4) = 0;  

r_dsl_filter(0, 5) = 0;  

r_dsl_filter(1, 0) = rho(1, 1);  

r_dsl_filter(1, 1) = 0;  

r_dsl_filter(1, 2) = 0;  

r_dsl_filter(1, 3) = 0;  

r_dsl_filter(1, 4) = 0;  

r_dsl_filter(1, 5) = 0;  

r_dsl_filter(2, 0) = rho(2, 2);  

r_dsl_filter(2, 1) = 0;  

r_dsl_filter(2, 2) = 0;  

r_dsl_filter(2, 3) = 0;  

r_dsl_filter(2, 4) = 0;  

r_dsl_filter(2, 5) = 0;  

r_dsl_filter(3, 0) = rho(3, 3);  

r_dsl_filter(3, 1) = 0;  

r_dsl_filter(3, 2) = 0;  

r_dsl_filter(3, 3) = 0;  

r_dsl_filter(3, 4) = 0;  

r_dsl_filter(3, 5) = 0;  

r_dsl_filter(4, 0) = rho(4, 4);  

r_dsl_filter(4, 1) = 0;  

r_dsl_filter(4, 2) = 0;  

r_dsl_filter(4, 3) = 0;  

r_dsl_filter(4, 4) = 0;  

r_dsl_filter(4, 5) = 0;  

r_dsl_filter(5, 0) = rho(5, 5);  

r_dsl_filter(5, 1) = 0;  

r_dsl_filter(5, 2) = 0;  

r_dsl_filter(5, 3) = 0;  

r_dsl_filter(5, 4) = 0;  

r_dsl_filter(5, 5) = 0;  

// Set initial estimate  

xhatmin_dsl_filter(0, 0) = xinit_mean(0);  

xhatmin_dsl_filter(1, 0) = xinit_mean(1);  

xhatmin_dsl_filter(2, 0) = xinit_mean(2);  

xhatmin_dsl_filter(3, 0) = xinit_mean(3);  

xhatmin_dsl_filter(4, 0) = xinit_mean(4);  

xhatmin_dsl_filter(5, 0) = xinit_mean(5);  

// Set initial process covariance matrix  

pminus_dsl_filter(0, 0) = xinit_noise(0);  

pminus_dsl_filter(0, 1) = 0;  

pminus_dsl_filter(0, 2) = 0;  

pminus_dsl_filter(0, 3) = 0;  

pminus_dsl_filter(0, 4) = 0;  

pminus_dsl_filter(0, 5) = 0;  

pminus_dsl_filter(1, 0) = 0;  

pminus_dsl_filter(1, 1) = xinit_noise(1);  

pminus_dsl_filter(1, 2) = 0;  

pminus_dsl_filter(1, 3) = 0;  

pminus_dsl_filter(1, 4) = 0;  

pminus_dsl_filter(1, 5) = 0;  

pminus_dsl_filter(2, 0) = 0;  

pminus_dsl_filter(2, 1) = xinit_noise(2);  

pminus_dsl_filter(2, 2) = 0;  

pminus_dsl_filter(2, 3) = 0;  

pminus_dsl_filter(2, 4) = 0;  

pminus_dsl_filter(2, 5) = 0;  

pminus_dsl_filter(3, 0) = 0;  

pminus_dsl_filter(3, 1) = xinit_noise(3);  

pminus_dsl_filter(3, 2) = 0;  

pminus_dsl_filter(3, 3) = 0;  

pminus_dsl_filter(3, 4) = 0;  

pminus_dsl_filter(3, 5) = 0;  

pminus_dsl_filter(4, 0) = 0;  

pminus_dsl_filter(4, 1) = xinit_noise(4);  

pminus_dsl_filter(4, 2) = 0;  

pminus_dsl_filter(4, 3) = 0;  

pminus_dsl_filter(4, 4) = 0;  

pminus_dsl_filter(4, 5) = 0;  

pminus_dsl_filter(5, 0) = 0;  

pminus_dsl_filter(5, 1) = xinit_noise(5);  

// Kalman Filter Loop  

//-----  

// Extended Kalman Filter  

for (pv5 = 0; pv5 <= n_steps - 1; pv5++)  

{  

    // Initialize identity matrix  

    //-----  

    // Definition of ID matrix  

    for (pv49 = 0; pv49 < n_statevars - 1; pv49++)  

        for (pv50 = 0; pv50 < n_statevars - 1; pv50++)  

            if (pv49 == pv50)  

                id_dsl_filter(pv49, pv50) = 1;  

            else  

                id_dsl_filter(pv49, pv50) = 0;  

    zpred_dsl_filter(0, 0) = xhatmin_dsl_filter(0, 0);  

    zpred_dsl_filter(1, 0) = xhatmin_dsl_filter(1, 0);  

    zpred_dsl_filter(2, 0) = xhatmin_dsl_filter(2, 0);  

    zhat_dsl_filter(0, 0) = z(pv5);  

    zhat_dsl_filter(1, 0) = z(1, pv5);  

    zhat_dsl_filter(2, 0) = z(2, pv5);  

    // Update loop dependent quantities  

    if (pv5 > 0)  

    {  

        phidsl_filter(2, 1) = t * (xhatmin_dsl_filter(3, 0) - u(0, pv5));  

        phidsl_filter(2, 0) = t * (u(1, pv5) - xhatmin_dsl_filter(4, 0));  

        phidsl_filter(1, 2) = t * (u(0, pv5) - xhatmin_dsl_filter(3, 0));  

        phidsl_filter(1, 0) = t * (xhatmin_dsl_filter(5, 0) - u(2, pv5));  

        phidsl_filter(0, 2) = t * (xhatmin_dsl_filter(4, 0) - u(1, pv5));  

        phidsl_filter(0, 1) = t * (u(2, pv5) - xhatmin_dsl_filter(5, 0));  

    }  

    else  

    {  

        // Update loop dependent quantities  

        if (pv5 > 0)  

        {  

            dv_dsl_filter(2, 0) = xhatmin_dsl_filter(5, 0);  

            dv_dsl_filter(1, 0) = xhatmin_dsl_filter(4, 0);  

            dv_dsl_filter(0, 0) = xhatmin_dsl_filter(0, 0);  

        }  

        else  

        {  

            gain_dsl_filter = pminus_dsl_filter *  

                (h_dsl_filter * transpose());  

            (x_dsl_filter +  

            h_dsl_filter *  

            (pminus_dsl_filter *  

            (h_dsl_filter).transpose())).inverse();  

            xhat1_dsl_filter = xhatmin_dsl_filter +  

            gain_dsl_filter *  

            (ch_dsl_filter - zpred_dsl_filter);  

            pplus_dsl_filter = (id_dsl_filter - gain_dsl_filter * h_dsl_filter) *  

            pminus_dsl_filter;  

            xhatmin_dsl_filter = phidsl_filter * phi_dsl_filter *  

            pminus_dsl_filter +  

            phi_dsl_filter *  

            (pplus_dsl_filter * (phi_dsl_filter).transpose());  

        }  

        // Populate Output Vector  

        for (pv39 = 0; pv39 < n_statevars - 1; pv39++)  

            xhat_dsl_filter(pv39, pv5) = xhat_dsl_filter(pv39, 0);  

    }  

    zpred_dsl_filter.resize(1);  

    zpred_dsl_filter(0) = xhat_dsl_filter;  

    return zpred_dsl_filter;  

}  

//-- End of code -----

```

# Automatic Program Synthesis Dilemma

*Do you trust your code generator?*

- Correctness of generated code depends on correctness of generator
- Correctness of generator difficult to show practically
  - too large
  - too complicated
  - too dynamic

```
const nat n := 6 as 'Number of state variables'.  
data double f(1..3, 1..oo) as 'gyro readings'.  
  
double x(1..n) as 'state variable vector'.  
double u(1..n) as 'process noise vector'.  
double q(1..n) as 'variance of process noise'.  
  
u(I) ~ gauss(0, q(I)).  
  
equations process_eqs are [  
  
dot x(1) := (hat x(4) - x(4)) - u(1)  
+ (f(3,t) - hat x(6)) * x(2)  
- (f(2,t) - hat x(5)) * x(3),  
  
...  
]
```

*Size*



```
// Calculate KH  
  
for(i = 0; i <= 5; i++)  
  
    for(j = 0; j <= 5; j++)  
  
        tmp0 = 0;  
  
        for(k = 0; k <= 2; k++)  
  
            tmp0 += gain[i][k] * h[k][j];  
  
        tmp1[i][j] = tmp0;  
  
// Calculate I-KH  
  
for(i = 0; i <= 5; i++)  
  
    for(j = 0; j <= 5; j++)  
  
        tmp2[i][j] = id[i][j] - tmp1[i][j];  
  
...
```

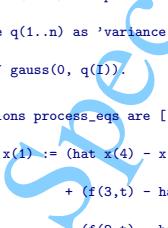
*Code*

# Automatic Program Synthesis Dilemma

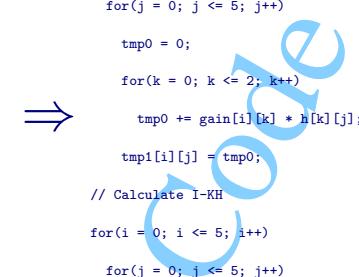
*Do you trust your code generator?*

- Correctness of generated code depends on correctness of generator
- Correctness of generator difficult to show practically
  - too large
  - too complicated
  - too dynamic

```
const nat n := 6 as 'Number of state variables'.  
data double f(1..3, 1..oo) as 'gyro readings'.  
  
double x(1..n) as 'state variable vector'.  
double u(1..n) as 'process noise vector'.  
double q(1..n) as 'variance of process noise'.  
  
u(I) ~ gauss(0, q(I)).  
  
equations process_eqs are [  
  
dot x(1) := (hat x(4) - x(4)) - u(1)  
+ (f(3,t) - hat x(6)) * x(2)  
- (f(2,t) - hat x(5)) * x(3),  
  
...  
]
```



```
// Calculate KH  
  
for(i = 0; i <= 5; i++)  
  
    for(j = 0; j <= 5; j++)  
  
        tmp0 = 0;  
  
        for(k = 0; k <= 2; k++)  
  
            tmp0 += gain[i][k] * h[k][j];  
  
        tmp1[i][j] = tmp0;  
  
// Calculate I-KH  
  
for(i = 0; i <= 5; i++)  
  
    for(j = 0; j <= 5; j++)  
  
        tmp2[i][j] = id[i][j] - tmp1[i][j];  
  
...
```



*So what?*

- Certify generated programs, not the generator
- Extend generator to “mark up” generated programs
- Use program verification techniques
- Focus on safety properties

# Program Verification

- Extend program by formal specification
  - description of its purpose

```
s = 0; t = 0;  
for (i = 0; i <= n; i++)  
  
{  
    s += x[i];  
    t += x[i] * x[i];  
}  
a = s / (n-1);  
d = (t - s*s/(n-1)) / (n-1);
```

# Program Verification

- Extend program by formal specification
  - description of its purpose
  - uses logic and mathematics
  - short and precise

```
// pre: n > 0
s = 0; t = 0;
for (i = 0; i <= n; i++)
    // inv: s = sum(0, i, x)
{
    s += x[i];
    t += x[i] * x[i];
}
a = s / (n-1);
d = (t - s*s/(n-1)) / (n-1);
// post: a = avg(x) ∧ d = dev(x)
```

# Program Verification

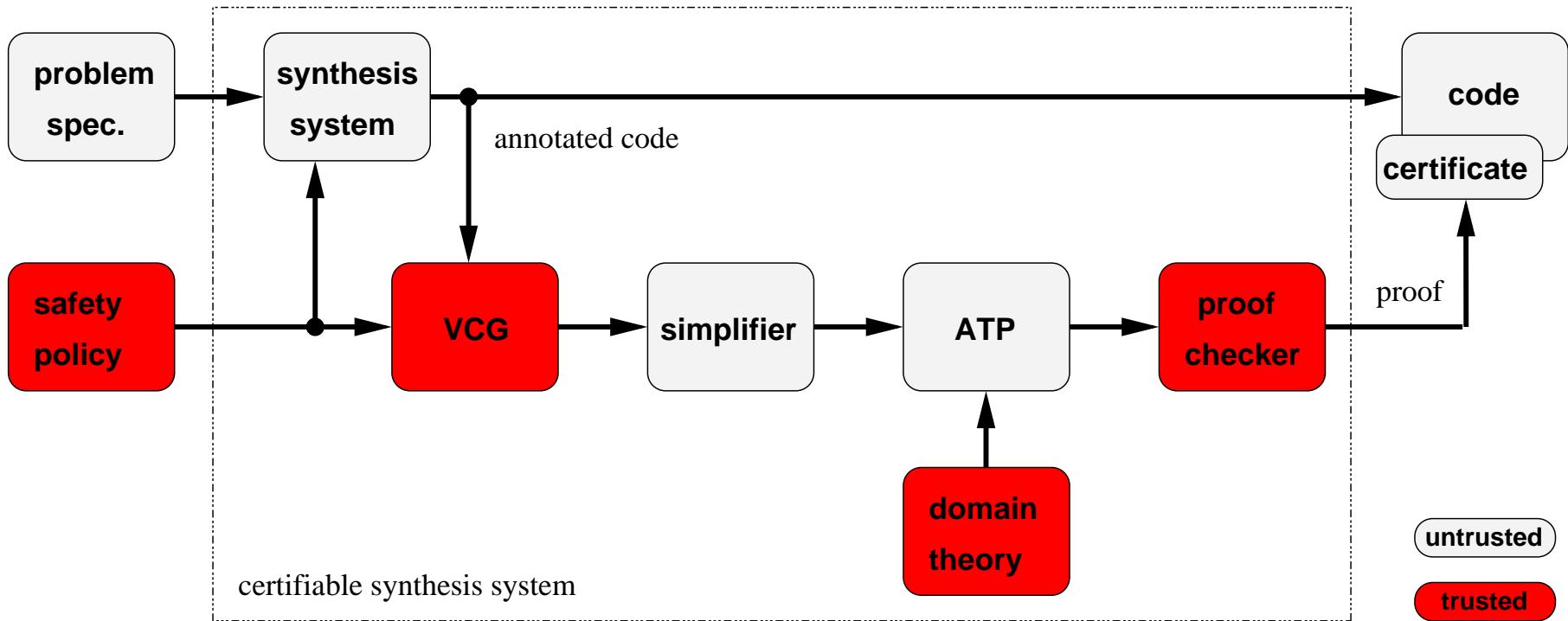
- Extend program by formal specification
    - description of its purpose
    - uses logic and mathematics
    - short and precise
  - Compute verification conditions (VCs)
    - logical formulae
    - derived from program and specification
    - can be automated by tools (VCG)
  - Prove VCs as theorems
    - paper-and-pencil (but tedious – cf. Pythagoras' Theorem)
    - can be supported by tools (“theorem provers”, ATP)
    - undecidable: no tool can prove everything
- ⇒ Correctness (wrt. specification) : $\Leftrightarrow$  all VCs are proven

# Program Verification

- Extend program by formal specification
    - description of its purpose
    - uses logic and mathematics
    - short and precise
  - Compute verification conditions (VCs)
    - logical formulae
    - derived from program and specification
    - can be automated by tools (VCG)
  - Prove VCs as theorems
    - paper-and-pencil (but tedious – cf. Pythagoras' Theorem)
    - can be supported by tools (“theorem provers”, ATP)
    - undecidable: no tool can prove everything
- ⇒ Correctness (wrt. specification) : $\Leftrightarrow$  all VCs are proven

*Testing can only show the presence of bugs, not their absence!*

# Certifiable Synthesis Architecture



- *Trusted vs. untrusted components:*
  - trusted components *must be correct*
  - untrusted components *may contain errors*
- Trusted component base minimized
  - trusted components simpler
  - untrusted components larger

## Example: Initialization Safety

Intuitively:

*Each variable shall have a defined value before it is used.*

```
void acs_nav( ){
    int i;
    while (...){
        i = i + 1; ...
    }
}
```

Compilers enforce this to some extent but have trouble with

- inputs
- arrays

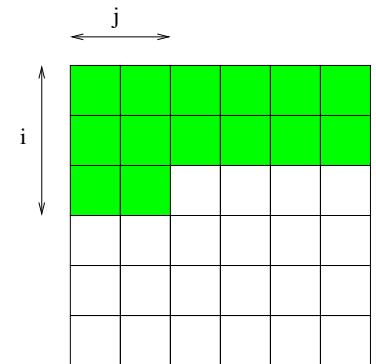
But scientific computation is largely array-based:

- DS1 error log lists several occurrences of incorrectly initialized arrays
- safety analysis can be arbitrarily complex

# Initialization: Annotated Code

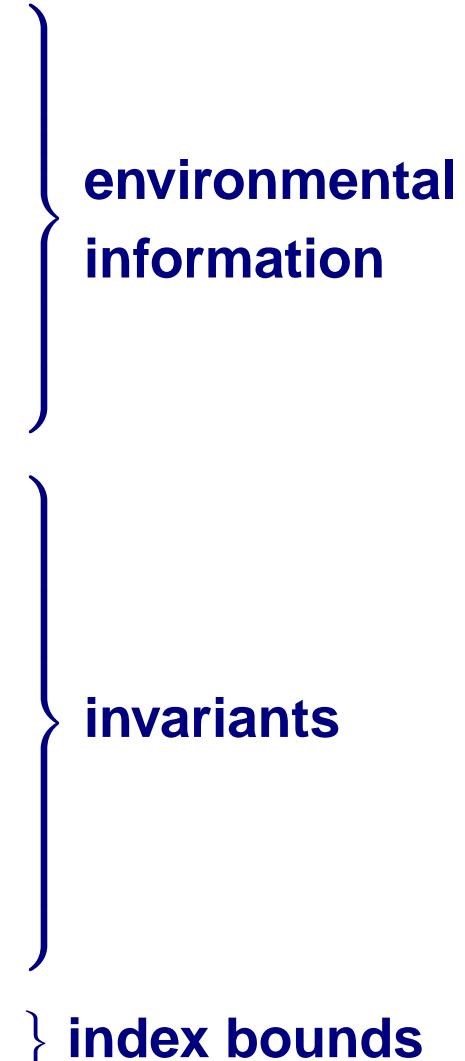
```
// Calculate I - KH

for(i = 0; i <= 5; i++)
/*{ inv forall x,y:int . 0<=x<=i-1 && 0<=y<=5 =>
    tmp2_init[x][y]==init
}*/
for(j = 0; j <= 5; j++)
/*{ inv forall x,y:int . 0<=x<=5 && 0<=y<=5 =>
    (x<i => tmp2_init[x][y]==init) &&
    (x==i && y<j => tmp2_init[x][y]==init)
}*/
    tmp2[i][j] = id[i][j] - tmp1[i][j];
/*{ post forall x,y:int . 0<=x<=i && 0<=y<=5 =>
    tmp2_init[x][y]==init
}*/
/*{ post forall x,y:int . 0<=x<=5 && 0<=y<=5 =>
    tmp2_init[x][y]==init
}*/
```



# Initialization: Verification Conditions

```
:  
  
(forall x,y:int . 0<=x<=5 && 0<=y<=5 =>  
  id_init[x][y]==init)  
&&  
(forall x,y:int . 0<=x<=5 && 0<=y<=5 =>  
  tmp1_init[x][y]==init)  
:  
(forall x,j:int . 0<=x<=i-1 && 0<=j<=5 =>  
  tmp2_init[x][j]==init)  
&&  
(forall x,y:int . 0<=y<=5 && 0<=x<=n_p-1 =>  
  (x<i => tmp2_init[x][y]==init &&  
   (y<j && x==i => tmp2_init[x][y]==init)))  
:  
( 0<=i<=5 ) && ( 0<=j<=5 )  
=>  
( id_init[i][j]==init && tmp1_init[i][j]==init ) } safety obligation
```



# Empirical Results

Array bounds checking – different algorithms:

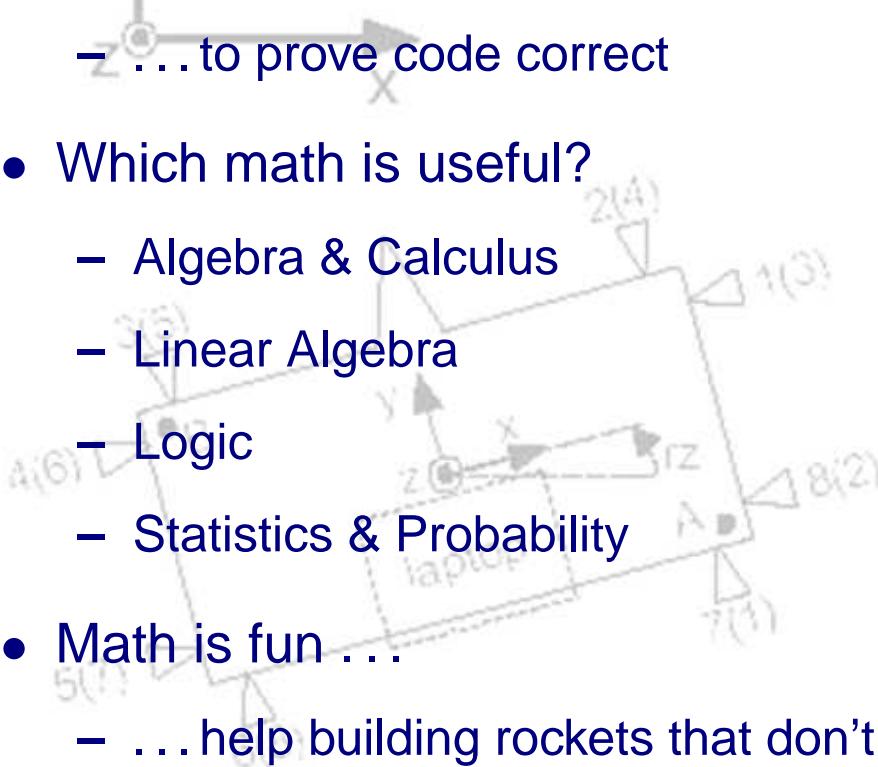
Algorithm	Spec	Code	Anns	#VCs	$T_{\text{proof}}$
EM	17	164	205	5	1.3s
Simplex	25	420	681	33	2m 1s
Extended KF	48	253	16	1	0.04s

Extended Kalman Filter – different safety policies:

Policy	Spec	Code	Anns	#VCs	$T_{\text{proof}}$
array	48	253	16	1	0.04s
init	48	253	165	72	1m 21s
symm	48	253	254	857	12m 57s

# Conclusions

- Math is useful ...
  - ... to describe “real-life” systems
  - ... to describe algorithms
  - ... to generate code
  - ... to prove code correct
- Which math is useful?
  - Algebra & Calculus
  - Linear Algebra
  - Logic
  - Statistics & Probability
- Math is fun ...
  - ... help building rockets that don’t explode!



$$\begin{aligned}
 \text{fwd.vframe} &= 326 \cdot T \\
 \text{accel.Cframe} &= f2a \cdot T \cdot \text{fwd.vframe} \\
 \text{accel.frame} &= \text{accel.Cframe} + CM32 \cdot \text{vframe} \\
 \text{real.frame} &= TxX \cdot \text{real.frame}
 \end{aligned}$$

T is handle offset center CM  
for now, assume CM is centered  
TxX relates from vehicle to world

$$x_0 = \text{accel.frame} \cdot TxX \cdot f2a \cdot t2f \cdot T$$

$$\begin{aligned}
 TxX &= \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} & f2a &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
 t2f &= \begin{bmatrix} -F & 0 & 0 & F & F & 0 & 0 & -F \\ 0 & -F & -F & 0 & 0 & F & F & 0 \\ F_3 & F_1 & F_1 & F_2 & F_2 & F_1 & F_1 & -F_3 \end{bmatrix} & F: & \text{Force per thruster (N)} \\
 & & F_x: & \text{Fx short moment arm (Nm)} \\
 & & F_l: & \text{Fx long } " " " "
 \end{aligned}$$

$$\begin{aligned}
 \begin{bmatrix} p_{x,1}(k+1) \\ p_{y,1}(k+1) \\ p_{z,1}(k+1) \\ v_{x,1}(k+1) \\ v_{y,1}(k+1) \\ v_{z,1}(k+1) \\ x(k+1) \end{bmatrix} &= \begin{bmatrix} 1 & & & T_2 & & & \\ & 1 & & & T_2 & & \\ & & 1 & & & T_2 & \\ & & & 1 & & & T_2 \\ & & & & 1 & & & T_2 \\ & & & & & 1 & & & T_2 \\ & & & & & & 1 & & & T_2 \end{bmatrix} \cdot \begin{bmatrix} p_{x,1}(k) \\ p_{y,1}(k) \\ p_{z,1}(k) \\ v_{x,1}(k) \\ v_{y,1}(k) \\ v_{z,1}(k) \\ x(k) \end{bmatrix} + \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \end{bmatrix} \cdot \begin{bmatrix} a_{x,1}(k) \\ a_{y,1}(k) \\ a_{z,1}(k) \\ u_{x,1}(k) \\ u_{y,1}(k) \\ u_{z,1}(k) \\ w(k) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
 X(k+1) &= A \cdot X(k) + B \cdot U(k) + W(k)
 \end{aligned}$$

$$T_{12} = \frac{T_1^2}{2}$$

$$\begin{aligned}
 \begin{bmatrix} 0.5 \cdot x & A \cdot \sin L \\ y & A \\ z & B \\ 1 & B \end{bmatrix} &= \begin{bmatrix} p_x \cdot A \cdot \sin L \\ p_y \cdot A \cdot \sin L \\ p_z \cdot A \cdot \sin L \\ 1 \end{bmatrix} + \begin{bmatrix} l_1 x A \cdot \cos -l_2 y A \cdot \sin \\ l_2 x A \cdot \sin + l_1 y A \cdot \cos \\ l_1 z B \cdot \cos - l_2 y B \cdot \sin \\ l_2 z B \cdot \sin + l_1 y B \cdot \cos \end{bmatrix} \\
 &\quad \cos \text{ is } \cos(\text{pre.} \cdot c(k)) \\
 &\quad \sin \text{ is } \sin(\text{pre.} \cdot s(k))
 \end{aligned}$$

$$\text{ROT} = \begin{bmatrix} \cos & -\sin \\ \sin & \cos \end{bmatrix}$$

$$\begin{bmatrix} 0.5 \cdot x & A \cdot \sin L \\ y & A \\ z & B \\ 1 & B \end{bmatrix} = \begin{bmatrix} 0.5 \cdot x \\ y \\ z \\ 1 \end{bmatrix} + \begin{bmatrix} \text{ROT} & 0 \\ 0 & \text{ROT} \end{bmatrix} \cdot \begin{bmatrix} l_1 A \\ l_2 A \\ l_1 B \\ l_2 B \end{bmatrix}$$

$$\begin{aligned}
 \begin{bmatrix} 0.5 \cdot x & A \cdot \sin L \\ y & A \\ z & B \\ 1 & B \end{bmatrix} &= \begin{bmatrix} 0.5 \cdot x \\ y \\ z \\ 1 \end{bmatrix} + \left( \begin{bmatrix} p_x \cdot A \\ p_y \cdot A \\ p_z \cdot A \\ 1 \end{bmatrix} + \begin{bmatrix} \text{ROT} & 0 \\ 0 & \text{ROT} \end{bmatrix} \cdot \begin{bmatrix} l_1 A \\ l_2 A \\ l_1 B \\ l_2 B \end{bmatrix} \right) + \begin{bmatrix} E_{1A} \\ E_{2A} \\ E_{3A} \\ E_{1B} \\ E_{2B} \\ E_{3B} \end{bmatrix} \\
 &\quad - \begin{bmatrix} B_A \\ B_B \end{bmatrix}
 \end{aligned}$$