Gadgetron

Release 4.0

David Hansen, Gradient Software

Jan 28, 2022

CONTENTS:

CHAPTER

ONE

RELEASE NOTES

1.1 4.0

- New *interface* for Gadgets, based on Channels, which makes it significantly easier to write new Gadgets. Old style Gadget interface still supported and are 100% compatible with the new interface.
- Branching chains now supported
- Vastly better support for distributed computing, which now allows for any Gadget which receives and sends standard Gadgetron datatypes to be distributed accross a many Gadgets.
- Improved error handling. Gadgetron should now produce more meaningful errors and single Gadgets can no longer crash the Gadgetron instance.
- Removed ACE. A compatiblity layer has been added to provide a standin for ACE_Message_Block. Importing ACE headers in any shape or form is no longer supported.
- Added support for NumPy-like slicing for hoNDArrays.

CHAPTER

TWO

WRITING A GADGET

A Gadget is a *Node* in the Gadgetron chain, which processes data comming in through an *GenericInputChannel* and sends the processed data to the next *Node* in the chain using an

OutputChannel.

The simplest Gadgets to write are *PureGadget* and *ChannelGadget*.

2.1 PureGadget

A *PureGadget* is a Gadget which processes Messages one at a time, and holds no state. Examples could be a Gadget which removes oversampling on *Acquisitions*, or one which takes an *Image* and performs autoscaling.

A PureGadget inheritss from *PureGadget<OUTPUT,INPUT>*, where OUTPUT and INPUT are the output type and input type of the Gadget.

AutoScaleGadget.h

```
#ifndef AUTOSCALEGADGET_H_
#define AUTOSCALEGADGET_H_

#include "Gadget.h"
#include "hoNDArray.h"
#include "gadgetron_mricore_export.h"

#include <ismrmrd/ismrmrd.h>

namespace Gadgetron{

  class EXPORTGADGETSMRICORE AutoScaleGadget:
    public Gadget2<ISMRMRD::ImageHeader,hoNDArray< float > >
  {
  public:
    GADGET_DECLARE(AutoScaleGadget);

    AutoScaleGadget();
    virtual ~AutoScaleGadget();

  protected:
    GADGET_PROPERTY(max_value, float, "Maximum value (after scaling)", 2048);

    virtual int process(GadgetContainerMessage<ISMRMRD::ImageHeader>* m1,
```

(continues on next page)

(continued from previous page)

```
                        GadgetContainerMessage< hoNDArray< float > >* m2);
    virtual int process_config(ACE_Message_Block *mb);

    unsigned int histogram_bins_;
    std::vector<size_t> histogram_;
    float current_scale_;
    float max_value_;
  };
}

#endif /* AUTOSCALEGADGET_H_ */
```

The **NODE_PROPERTY** macro defines a variable on the AutoScaleGadget which can be set from the XML file defining the chain.

AutoScaleGadget.cpp

```
/*
 * AutoScaleGadget.cpp
 *
 *  Created on: Dec 19, 2011
 *      Author: Michael S. Hansen
 */

#include "AutoScaleGadget.h"

namespace Gadgetron{

AutoScaleGadget::AutoScaleGadget()
        : histogram_bins_(100)
        , current_scale_(1.0)
        , max_value_(2048)
{
}

AutoScaleGadget::~AutoScaleGadget() {
        // TODO Auto-generated destructor stub
}

int AutoScaleGadget::process_config(ACE_Message_Block* mb) {
        max_value_ = max_value.value();
        return GADGET_OK;
}


int AutoScaleGadget::process(GadgetContainerMessage<ISMRMRD::ImageHeader> *m1,␣
↪GadgetContainerMessage<hoNDArray<float> > *m2)
{
        if (m1->getObjectPtr()->image_type == ISMRMRD::ISMRMRD_IMTYPE_MAGNITUDE) { //
↪Only scale magnitude images for now
                float max = 0.0f;
                float* d = m2->getObjectPtr()->get_data_ptr();
                for (unsigned long int i = 0; i < m2->getObjectPtr()->get_number_of_
↪elements(); i++) {
```

(continues on next page)

(continued from previous page)

```
                        if (d[i] > max) max = d[i];
                }

                if (histogram_.size() != histogram_bins_) {
                        histogram_ = std::vector<size_t>(histogram_bins_);
                }

                for (size_t i = 0; i < histogram_bins_; i++) {
                        histogram_[i] = 0;
                }

                for (unsigned long int i = 0; i < m2->getObjectPtr()->get_number_of_
↪elements(); i++) {
                        size_t bin = static_cast<size_t>(std::floor((d[i]/max)*histogram_
↪bins_));
                        if (bin >= histogram_bins_) {
                                bin = histogram_bins_-1;
                        }
                        histogram_[bin]++;
                }

                //Find 99th percentile
                long long cumsum = 0;
                size_t counter = 0;
                while (cumsum < (0.99*m2->getObjectPtr()->get_number_of_elements())) {
                        cumsum += (long long)(histogram_[counter++]);
                }
                max = (counter+1)*(max/histogram_bins_);
                GDEBUG("Max: %f\n",max);

                current_scale_ = max_value_/max;

                for (unsigned long int i = 0; i < m2->getObjectPtr()->get_number_of_
↪elements(); i++) {
                        d[i] *= current_scale_;
                }
        }

        if (this->next()->putq(m1) < 0) {
                GDEBUG("Failed to pass on data to next Gadget\n");
                return GADGET_FAIL;
        }

        return GADGET_OK;
}

GADGET_FACTORY_DECLARE(AutoScaleGadget)

}
```

Note the **GADGETRON_GADGET_EXPORT** declaration, which produces the code causing the AutoScaleGadget to be loadable by Gadgetron.

2.2 ChannelGadget

PureGadget can't hold any state between different messages and must send one message per input. This makes it easier to reason about and implement, but is also limiting in cases where we want to accumulate multiple messages for processing. In this case, *ChannelGadget* should be used. If we want to create a Gadget which takes several *Acquisitions* and reconstruct them, we inherit from *ChannelGadget<Acquisition>*.

```
using namespace Gadgetron;
using namespace Gadgetron::Core;
class SimpleRecon : public ChannelGadget<Acquisition>{
    public:
        void process(InputChannel<Acquisition>& in, OutputChannel& out) override {
            ...
        }
}
```

We can take the messages from the channel either by calling `InputChannel::pop()` directly, or by using it in a for loop.

Channels are ranges, meaning they can be used directly with for loops and with algorithm from standard library, such as `std::transform()` and `std::accumulate()`.

```
void process(InputChannel<Acquisition>& in, OutputChannel& out) override {
    for (auto acquisition : in ) {

        auto& header = std::get<ISMRMRD::AcquisitionHeader>(acquisition);
        auto& data = std::get<hoNDArray<std::complex<float>>>(acquisition);
        auto& trajectory = std::get<optional<hoNDArray<float>>>(acquisition);
        //Gather acquisitions here
    }
}
```

Or if you're using C++17, this would be

```
void process(InputChannel<Acquisition>& in, OutputChannel& out) override {
    for (auto [header, data, trajectory] : in ) {
        //Gather acquisitions here
    }
}
```

We want to gather acquisitions until we have enough for a (possibly undersampled) image. The AcquisitionHeader has the ISMRMRD::_ACQ_LAST_IN_ENCODE_STEP1 flag which we can use as a trigger. By importing channel_algorithms.h, we can write

```
void process(InputChannel<Acquisition>& in, OutputChannel& out) override {

    auto split_condition = [](auto& message){
      return std::get<ISMRMRD::AcquisitionHeader>(message).isFlagSet(ISMRMRD::ISMRMRD_
↪ACQ_LAST_IN_ENCODE_STEP1);
    };

    for (auto acquisitions : buffer(in,split_condition)) {
        for (auto [header, data, trajectory] : acquisitions ) {
        //Gather acquisitions here
```

(continues on next page)

(continued from previous page)

```
        }
}
```

```
#include <gadgetron/Gadget.h>
#include <gadgetron/hoNDFFT.h>
#include <gadgetron/mri_core_utility.h>
#include <gadgetron/ChannelAlgorithms.h>
#include <gadgetron/log.h>
#include <gadgetron/mri_core_coil_map_estimation.h>
using namespace Gadgetron;
using namespace Gadgetron::Core;
using namespace Gadgetron::Core::Algorithm;

class SimpleRecon : public ChannelGadget<Acquisition> {

    public:
        SimpleRecon(const Context& context, const GadgetProperties& params) :␣
↪ChannelGadget<Acquisition>(params), header{context.header} {

        }

        void process(InputChannel<Acquisition>& in, OutputChannel& out){

            auto recon_size = header.encoding[0].encodedSpace.matrixSize;

            ISMRMRD::AcquisitionHeader saved_header;

            auto split_condition = [](auto& message){
            return std::get<ISMRMRD::AcquisitionHeader>(message).
↪isFlagSet(ISMRMRD::ISMRMRD_ACQ_LAST_IN_ENCODE_STEP1);
            };

            for (auto acquisitions : buffer(in,split_condition)) {

               auto data = hoNDArray<std::complex<float>>(recon_size.x,recon_size.y,
↪recon_size.z,header.acquisitionSystemInformation->receiverChannels.get());
               for ( auto [acq_header, acq_data, trajectories] : acquisitions){
                    saved_header = acq_header;
                    data(slice,acq_header.idx.kspace_encode_step_1,0,slice) = acq_data;
                }

                hoNDFFT<float>::instance()->fft2c(data);

                auto coil_map = coil_map_Inati(data);
                data = coil_combine(data,coil_map,3);

                auto image_header = image_header_from_acquisition(saved_header,header,
↪data);

                out.push(image_header,data);
            }
        }
```

(continues on next page)

(continued from previous page)

```
    private:
        const ISMRMRD::IsmrmrdHeader header;
};

GADGETRON_GADGET_EXPORT(SimpleRecon)
```

CHAPTER

THREE

TYPE MATCHING

PureGadget, *ChannelGadget* as well as several other classes in Gadgetron have a template arguement defining what type of data they accept. In the simplest case, this is simply a list of types. For instance

```
ChannelGadget<ISMRMRD::AcquisitionHeader, hoNDArray<std::complex<float>>>
```

will match any message starting with a `AcquisitionHeader` followed by a `hoNDArray`. If the message has more parts, these will simply be discarded. Also note that this is equivalent to

```
ChannelGadget<tuple<ISMRMRD::AcquisitionHeader, hoNDArray<std::complex<float>>>>
```

3.1 Optional

If we want to include an element that will only appear some times, we can define it as optional. For instance, acquisitions can have a trajectory attached to them. This would look like

```
ChannelGadget<ISMRMRD::AcquisitionHeader, hoNDArray<std::complex<float>>, optional
↪<hoNDArray<float>>>
```

and in fact Types.h defines *Acquisition* as

```
using Acquisition = tuple<ISMRMRD::AcquisitionHeader,  hoNDArray<std::complex<float>>,
↪optional<hoNDArray<float>>>;
```

3.2 Variant

What if you need to create a ChannelGadget that accepts multiple types? For instance, one which receives both Acquisition and Waveform. In this case we can use a variant.

In order to work with the data, you call `Core::visit`, which is modelled from std::visit.

For instance, a toy example which counts the number of data points in all waveforms and acquisitions could look like

CHAPTER

FOUR

API

4.1 Core

4.1.1 Nodes and Gadgets

class `Gadgetron::Core::Node`

Node is the base class for everything in a Gadgetron chain, including Gadgets and TypedChannelGadgets

Subclassed by *Gadgetron::Core::GenericChannelGadget*, Gadgetron::LegacyGadgetNode

Public Functions

virtual void **process**(*GenericInputChannel* &in, *OutputChannel* &out) = 0

The function which processes the data comming from the *InputChannel*. Conceptually a coroutine.

Parameters

- **in** – Channel from which messages are received from upstream
- **out** – Channel in which messages are sent on downstream

class **GenericChannelGadget** : public Gadgetron::Core::*Node*, public Gadgetron::Core::PropertyMixin

Subclassed by *Gadgetron::Core::ChannelGadget< TYPELIST >*, *Gadgetron::Core::GenericPureGadget*

template<class ...**TYPELIST**>

class `Gadgetron::Core::ChannelGadget` : public Gadgetron::Core::*GenericChannelGadget*

A *Node* providing typed access to input data. Messages not matching the TYPELIST are simply passed to the next *Node* in the chain. Should be the first choice for writing new Gadgets.

tparam TYPELIST The type(s) of the messages to be received

Public Functions

inline virtual void **process**(*GenericInputChannel* &in, *OutputChannel* &out) final
The function which processes the data comming from the *InputChannel*. Conceptually a coroutine.

Parameters

- **in** – Channel from which messages are received from upstream
- **out** – Channel in which messages are sent on downstream

virtual void **process**(*InputChannel*<*TYPELIST*...> &in, *OutputChannel* &out) = 0
The process function to be implemented when inheriting from this class.

Parameters

- **in** – A channel of the types specified in TYPELIST
- **out** – Channel of output

class `Gadgetron::Core::GenericPureGadget` : public Gadgetron::Core::*GenericChannelGadget*
Subclassed by *Gadgetron::Core::PureGadget< RETURN, INPUT >*

Public Functions

inline virtual void **process**(*GenericInputChannel* &in, *OutputChannel* &out) final
The function which processes the data comming from the *InputChannel*. Conceptually a coroutine.

Parameters

- **in** – Channel from which messages are received from upstream
- **out** – Channel in which messages are sent on downstream

template<class **RETURN**, class **INPUT**>

class **PureGadget** : public Gadgetron::Core::*GenericPureGadget*

4.1.2 Channels

class **MessageChannel** : public Gadgetron::Core::Channel

class `Gadgetron::Core::OutputChannel`
The end of a channel which provides output. Only constructible through make_channel(args)

Public Functions

OutputChannel(*OutputChannel* &&other) noexcept = default

OutputChannel &**operator=**(*OutputChannel* &&other) noexcept = default

template<class ...**ARGS**>
void **push**(*ARGS*&&... ptrs)
Pushes a message of type ARGS to the channel.

void **push_message**(Message)
Pushes a message to the channel.

class Gadgetron::Core::**GenericInputChannel** : public ChannelRange<*GenericInputChannel*>
The end of a channel which provides input

Public Functions

GenericInputChannel(*GenericInputChannel* &&other) noexcept = default

GenericInputChannel &**operator=**(*GenericInputChannel* &&other) noexcept = default

Message **pop**()
Blocks until it can take a message from the channel.

optional<Message> **try_pop**()
Nonblocking method returning a message if one is available, or None otherwise.

template<class ...**TYPELIST**>

class **InputChannel** : public ChannelRange<*InputChannel*<*TYPELIST*...>>

4.1.3 Core Types

using Gadgetron::Core::**Acquisition** = tuple<ISMRMRD::AcquisitionHeader, hoNDArray<std::complex<float>>, *optional*<hoNDArray<float>>>
An Acquisition consists of a data header, the kspace data itself and optionally an array of kspace trajectories.

using Gadgetron::Core::**Image** = tuple<ISMRMRD::ImageHeader, hoNDArray<T>, *optional*<ISMRMRD::MetaContainer>>
An image consists of a header, an array of image data and optionally some metadata.

using Gadgetron::Core::**Waveform** = tuple<ISMRMRD::WaveformHeader, hoNDArray<uint32_t>>
A Waveform consiste of a header, followed by the raw Waveform data. See the MRD documentation page for more details.

using Gadgetron::Core::**optional** = boost::optional<T>

Warning: doxygentypedef: Cannot find typedef “Gadgetron::Core::variant” in doxygen xml output for project “Gadgetron” from directory: doc/xml

Warning: doxygenfunction: Cannot find function “Gadgetron::Core::visit” in doxygen xml output for project “Gadgetron” from directory: doc/xml

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

G